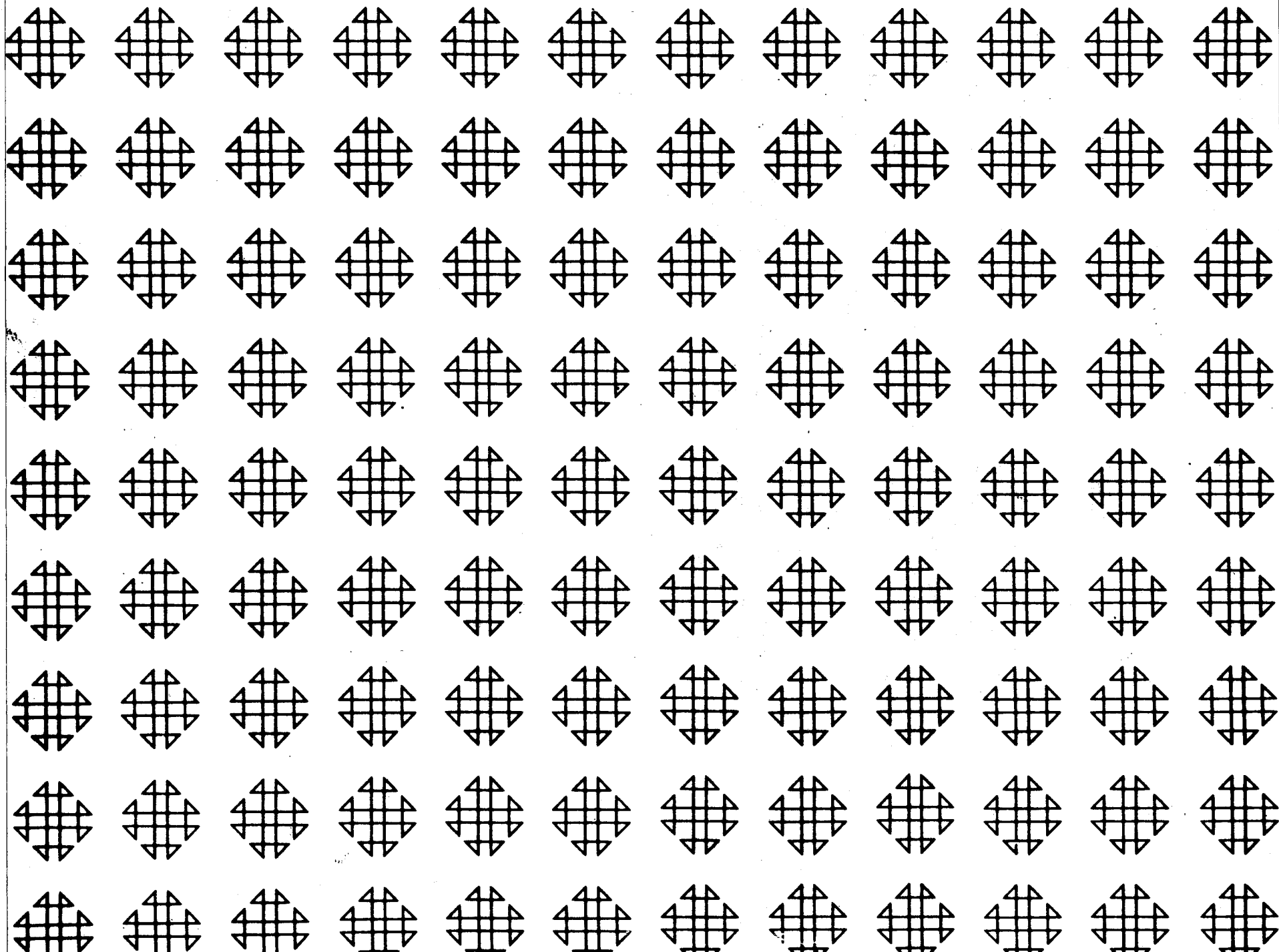


Data Language /1 360D-01.6.007

DISCLAIMER

This program and its documentation have been contributed to the Program Information Department by an IBM employee and are provided by the IBM Corporation as part of its service to customers. The program and its documentation are essentially in the author's original form and have not been subjected to any formal testing. IBM makes no warranty expressed or implied as to the documentation, function, or performance of this program and the user of the program is expected to make the final evaluation as to the usefulness of the program in his own environment. There is no committed maintenance for the program.

Questions concerning the use of the program should be directed to the author or other designated party. Any changes to the program will be announced in the appropriate Catalog of Programs; however, the changes will not be distributed automatically to users. When such an announcement occurs, users should order only the material (documentation, machine readable or both) as indicated in the appropriate Catalog of Programs.

Program Contribution Form

Type III (IBM Employee)

IBM Corporation
Program Information Department (PID)
40 Saw Mill River Road
Hawthorne, New York 10532, U.S.A.
Attention: Program Control Desk

1 PROGRAM ORDER NUMBER (TO BE FILLED IN BY PID): 360D-01.6.007

2 SYSTEM TYPE (MACHINE): 360

3 SEARCH KEY: DATA BASE ORGANIZATION FOR OS/360 /SUPPORTING/ HIERARCHICAL SEGMENTED FILE STRUCTURES

4 AUTHORS' NAME(S) (IF DIFFERENT THEN SUBMITTER'S)

5 SUBMITTER'S NAME (DIRECT TECHNICAL INQUIRIES TO): Vern L. Watts

6 SUBMITTER'S ADDRESS: IBM Corporation
1930 Century Park West - LA Dev. Ctr.
Los Angeles, California 90067

7 TITLE OF PROGRAM: Data Language/I

8 PRIM.SUB. CODE: 0116

9 SECONDARY SUBJECT CODES:

10 OPERATING OR MONITOR SYSTEM REQUIRED: OS/360

11 TYPE OF SUBMITTAL PLEASE CHECK (√): 1 INITIAL PROGRAM ABSTRACT | 2 √ INITIAL SUBMISSION OF PROGRAM | 3 RESUBMITTAL OF UNANNOUNCED PROG. | 4 CORRECTION TO PROGRAM NUMBER

12 DATE OF SUBMITTAL: 9-20-68

13 ABSTRACT (PLEASE LIMIT TO 150 WORDS. DESCRIBE PROGRAM AND PURPOSE. CLEARLY IDENTIFY MACHINE CONFIGURATION AND SOURCE LANGUAGE.)

Data Language/I is a file management-data base interface that supports hierarchical segmented structures under control of Operating System/360. It will operate under PCP, MFT-II (Multiprogramming with a Fixed Number of Tasks, Version II), or MVT (Multiprogramming with a Variable Number of Tasks) and supports programs written in COBOL, PL/I, and Assembler Language. The minimum configuration is the System/360 Model 40, 128K, with decimal arithmetic feature and device requirements to support Operating System/360. Source Language of this program is Assembler Language.

Data Language/I provides application independence from access methods, from physical storage organization, and from the characteristics of devices on which the application's data is stored. This independence is provided by a common source linkage program and by data base descriptions external to the application program. These external data base descriptions describe the logical data organization of data bases to Data Language/I and make it possible to physically reorganize established data bases in a timely manner without modification to application programs. Reduction in application program maintenance should result.

PLEASE ATTACH ADDITIONAL PAGES IF NECESSARY → TOTAL PAGES ATTACHED

120-1424-3

PERMISSION TO PUBLISH: "I HEREBY GIVE IBM PERMISSION TO REPRINT, REPRODUCE AND DISTRIBUTE THIS PROGRAM TO ANYONE."

14 Vern L. Watts 11-19-68
SIGNATURE OF SUBMITTER AND DATE

CONTENTS

SECTION 1. DATA LANGUAGE/I PROGRAM DESCRIPTION---------- 1i
WHAT IS DATA LANGUAGE/I?---------- 1i
DATA LANGUAGE/I VS OPERATING SYSTEM/360 DATA MANAGEMENT---------- 3
Data Language/I - Data Base---------- 3
APPLICATION DEVELOPMENT AND STRUCTURING OF DATA LANGUAGE/I---------- 6
DATA LANGUAGE/I---------- 12
Data Language/I Major Features---------- 12
Data Language/I Rules---------- 14
DATA BASE ORGANIZATION---------- 17
TYPES OF DATA BASES---------- 17
Simple Hierarchical Files---------- 17
Complex Hierarchical Files---------- 18
STRUCTURE OF DATA BASES---------- 21
Data Language/I - Data Base Organization---------- 22
DATA BASE PROCESSING---------- 34
Introduction---------- 34
Data Base Creation---------- 35
Data Base Retrievals---------- 37
Data Base Updates---------- 38
Data Base Deletions---------- 39
Data Base Insertions---------- 39
Program Specification Block (PSB)---------- 39

Data Base Segment Sensitivity---------- 40
Data Base Segment Definition---------- 41
APPLICATION PROGRAM STRUCTURE---------- 42
BATCH PROGRAM STRUCTURE---------- 42
COBOL Batch Program Structure---------- 42
PL/I Batch Program Structure---------- 46
Assembler Language Batch Program Structure---------- 50
THE LANGUAGE INTERFACE---------- 50
Parameter List Contents---------- 51
SEGMENT SEARCH ARGUMENTS (SSA)---------- 53
Segment Name---------- 54
Segment Qualification Statement---------- 54
Examples of SSA Usage---------- 55
SEGMENT INPUT/OUTPUT AREAS---------- 57
PROGRAM COMMUNICATION BLOCK (PCB) FORMAT---------- 57
PCB for a Data Base---------- 57
COBOL---------- 59
PL/I---------- 60
ENTRY TO APPLICATION PROGRAMS---------- 61
COBOL---------- 61
PL/I---------- 61
DATA LANGUAGE/I DATA BASE CALLS---------- 61
The GET UNIQUE Call (GUbb) - Data Base---------- 62
Status Codes for GET UNIQUE Calls---------- 63
The GET NEXT Call (GNbb) - Data Base---------- 64
Status Codes for GET NEXT Calls - Data Base---------- 65

Definition of Cross Hierarchical Boundary---------- 66
The GET NEXT WITHIN PARENT Call (GNPb) - Data Base---- 66
Status Codes for GET NEXT WITHIN PARENT Calls--------- 66
The GET HOLD Calls - Data Base------------------------ 68
Status Codes for GET HOLD Calls----------------------- 69
The INSERT Call (ISRT) - Data Base-------------------- 69
Status Codes for INSERT Calls------------------------- 71
The DELETE Call (DLET) - Data Base-------------------- 74
The REPLACE Call (REPL) - Data Base-------------------- 75
Status Codes for DELETE/REPLACE Calls----------------- 76
TERMINATION OF AN APPLICATION PROGRAM----------------- 77
PROCESSING REGION ABENDS------------------------------ 78
Commonly Encountered Operating System/360 System Abends 80
DATA BASE DESCRIPTIONS (DBD)-------------------------- 80
DBD CONTROL CARD REQUIREMENTS------------------------- 83
PRINT Control Card------------------------------------ 84
DBD Control Card-------------------------------------- 84
DMAN Control Card------------------------------------- 85
SEGM Control Card------------------------------------- 88
FLDK Control Card------------------------------------- 90
FLD Control Card-------------------------------------- 91
DBDGEN Control Card----------------------------------- 92
FINISH Control Card----------------------------------- 92
END Control Card-------------------------------------- 92
DBDGEN Examples--------------------------------------- 92

DESCRIPTION OF DBD GENERATION OUTPUT------------------- 96
Control Card Listing---------------------------------- 96
Diagnostics--- 96
Assembly Listing-------------------------------------- 97
Load Module--- 97
DBD Generation Error Conditions----------------------- 97
PROGRAM SPECIFICATION BLOCK (PSB) GENERATION---------- 101
PSB Requirements-------------------------------------- 101
PCB Control Card - Data Language/I Data Base PCB------ 101
SENSEG Control Card - Sensitive Segments-------------- 103
PSBGEN Control Card----------------------------------- 105
DESCRIPTION OF PSB GENEPATION OUTPUT------------------ 106
Control Card Listing---------------------------------- 106
Diagnostics--- 106
Assembly Listing-------------------------------------- 106
Load Module--- 106
PSB Generation Error Conditions----------------------- 107
SECTION 2. SYSTEMS OPERATION OF DATA LANGUAGE/I------- 109
SYSTEM DISTRIBUTION, HANDLING, AND MAINTENANCE------- 109
SYSTEM DISTRIBUTION----------------------------------- 109
Nine-Track Tape--------------------------------------- 110
Seven-Track Tape-------------------------------------- 110
SYSTEM HANDLING--------------------------------------- 111
PLANNING-- 113
Batch Processing Region System Flow------------------ 113

Impact of MFT-II and MVT on Data Language/I---------- 116
ESTIMATING STORAGE REQUIREMENTS---------- 117
DATA BASE SPACE ALLOCATION, FORMAT, CREATION, AND REORGANIZATION---------- 117
Data Language/I Data Base Space Allocation---------- 117
Allocation Problem Example---------- 118
ISAM PRIME AREA---------- 118
INDEX AREA---------- 123
OSAM OVERFLOW---------- 124
EXAMPLE CONCLUSION---------- 125
ISAM DCB Option Codes---------- 126
OSAM DCB Option Codes---------- 126
Data Language/I Record Format---------- 126
TYPE 1 RECORD FORMAT---------- 129
TYPE 2 RECORD FORMAT---------- 130
TYPE 3 RECORD FORMAT---------- 131
TYPE 4 RECORD FORMAT---------- 132
TYPE 5 RECORD FORMAT---------- 133
EXAMPLES OF TYPES OF DATA LANGUAGE/I LOGICAL RECORDS- 135
Variable Length Data Base Record Processing---------- 138
PSB Generation---------- 139
DBD Generation---------- 139
Data Base Creation---------- 141
Single and Multiple Data Set Groups---------- 141
Considerations of HISAM and HSAM---------- 142
Data Base Reorganization---------- 143

SYSTEM DEFINITION FOR DATA LANGUAGE/I---------- 143
System Definition---------- 143
System Definition Macro-Instructions---------- 144
DLICTRL Macro---------- 145
MACLIB Macro---------- 146
RESLIB Macro---------- 147
PGMLIB Macro---------- 148
PSBLIB Macro---------- 149
DBDLIB Macro---------- 150
PROCLIB Macro---------- 151
DLIGEN Macro---------- 152
Supervisor Call Routine---------- 154
OSAM Channel End Appendage---------- 154
System Libraries and Procedures---------- 154
LIBRARIES---------- 161
Operating System/360 Link Pack Modules---------- 162
IMPLEMENTATION---------- 164
DATA BASE---------- 164
DBD Generation---------- 164
Description of Data Language/I Segment Insertion------ 164
ROOT SEGMENT INSERTION---------- 165
DEPENDENT SEGMENT INSERTION---------- 166
Data Base Integrity Through the Use of OSAM---------- 169
PSB Generation---------- 172
JCL - Batch Processing Region---------- 173

System Definition---------- 174

Data Set Allocation---------- 175

Inclusion of SVC Routines in Operating System/360 Nucleus---------- 175

Procedure Section---------- 177

OSAM Appendage to SYS1.SVCLIB---------- 178

Tape Key

3 Files

DLI.GENLIB

Record Count = 981

DLI.LOAD

Record Count = 90

DLI.SOURCE

Record Count =1860

All Files: Logical Record = 80 Characters

Blocking Factor = 10

EBDIC

Tape: 9 Track

800 BPI

Unlabeled

EBDIC

WHAT IS DATA LANGUAGE/I?

The traditional limitation of every data processing application has been the organization of the data to be manipulated. The structure of each data record and its manner and medium of storage have affected application design and programming, and a great deal of effort has been expended to free the data organization from the physical restriction of the data storage medium.

It is the purpose of Data Language/I to allow the application program to gain a high degree of independence from the input/output software systems and storage devices that are required for storage and manipulation of the data. As seen in Figure 1, Data Language/I provides a "wall" or separation between the application program and the data bases. An application program has two distinct interfaces with Data Language/I: (1) A data base description, a mapping or transformation relating the logical data structure and physical storage structure of the data base given as a definition external to the application program; and (2) a common source program linkage (referred to later in this manual as the application program language interface), which allows data input/output requests during the execution of the application program.

A second, and possibly more important, purpose of Data Language/I is to provide a medium through which a programmer can have access to large data files not specifically built and organized for his use. This should lead to the ability to combine common data into a single data base rather than maintain redundant data. Data Language/I relieves the application program of the necessity of knowing the physical location of its data in the data bases. The application program requests input/output data base operations of Data Language/I using the logical data relationships of the application. Data Language/I translates or maps this logical data relationship to the physical storage of the data. In this manner, the physical storage of data may be changed and, if the logical data relationships are retained, the application programs need not be modified.

The availability of noncustomized data brings into full meaning the concept of a data base. In this context, the ability to create and access large data files having multiple uses and eliminating redundant information takes on real meaning.

The following are the significant capabilities available to the Data Language/I user:

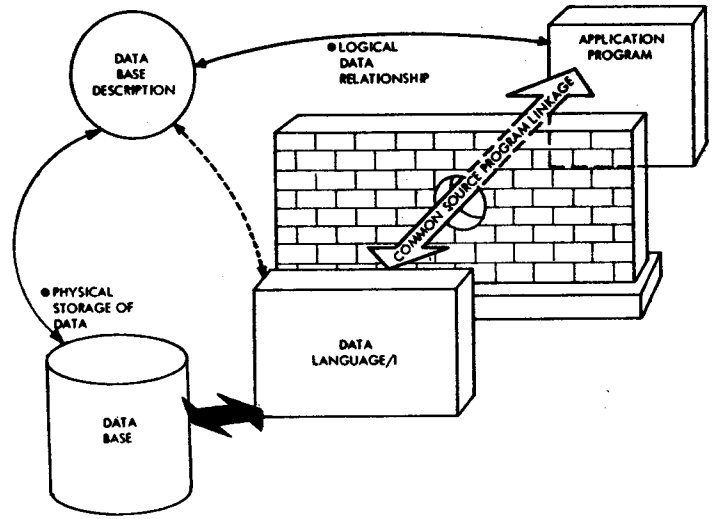


Figure 1. Data Language/I relationship between application program and data base

1. A common source program interface is provided between the application program and the data base.
2. A data base description provides a mapping from the logical data relationships to the physical storage of the data. This description is maintained external to the application program.
3. The ability for a program to define the portions of a data base to which it wishes to be "sensitive" (that is, to have access) without considering the total data base

structure. This permits the organization of nonsensitive data to be changed or added to without affecting application programs.

4. Data Language/I use of Operating System/360 fixed-length ISAM with an improved capability for data insertion and overflow control.

5. A hierarchical data element relationship is introduced between the various portions of a data base. This permits the handling of variable-length data structures in a fixed-length manner. Simplification of data handling should be experienced in COBOL or PL/I application programs.

6. A utility program is provided for use in describing and storing a definition of the data base structure. (See "Data Base Description".)

7. A utility program is provided for use in describing the application program's data base "sensitivity and usage". (See "Program Specification Block".)

DATA LANGUAGE/I VS OPERATING SYSTEM/360 DATA MANAGEMENT

This portion of the manual shows the relationship of Data Language/I to Operating System/360 Data Management, lists the difference between the two, and defines the terminology associated with each.

Data Language/I - Data Base

The data base concept is introduced by Data Language/I. In order to define the term data base, its relationship to the Operating System/360 Data Management data set should first be defined. The SRL publication, IBM System/360 Operating System Supervisor and Data Management Services, says, "Any information that is a named, organized collection of logically related records can be classified as a data set.... A data set may be...a file of data records used by a processing program". The data set is the major unit of data storage and retrieval in Operating System/360. Figure 2 shows the Operating System/360 Data Management data set structure to be made up of physical records which are further broken down into logical records. The only relationship between the physical and logical data structure provided by Operating System/360 is one or more logical records within a physical record.

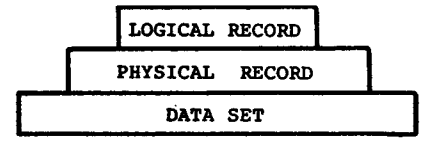


Figure 2. OS/360 data management data set structure

Data Language/I, in order to accommodate variable-length application records (data base records), provides the capability of a logical record within a physical record or a logical record spanning one or more physical records.

A data base may be considered similar to a data set because it is an organized collection of data entered and maintained in some logical sequence to facilitate later inquiry and processing.

In the application data sense, the data base is composed of data base records (Figure 3). The data base record is the logical record of the application. A data base record is a collection (of variable number) of fixed-length data elements, called "segments", hierarchically related to a single occurrence of a root segment. A segment is a portion of a data base record containing one or more logically related data fields. A root segment is the highest hierarchical segment in the data base record. Each data base record must have only one

root segment. The root segment comprises data which applies to all users in the processing of the data base record. A dependent segment is a segment that relies on at least the root segment for its full hierarchical meaning. It is therefore always at a lower hierarchical level than the root segment. A dependent segment may also be dependent on other dependent segments for its full meaning. In order to process the segments in a data base, it is only necessary for the user to be aware of those segments which comprise the data base record and the relationship of these segments to each other (that is, the logical data base record structure of segments). There can be 255 segment-types within a data base and 15 levels of segment hierarchy within a data base record.

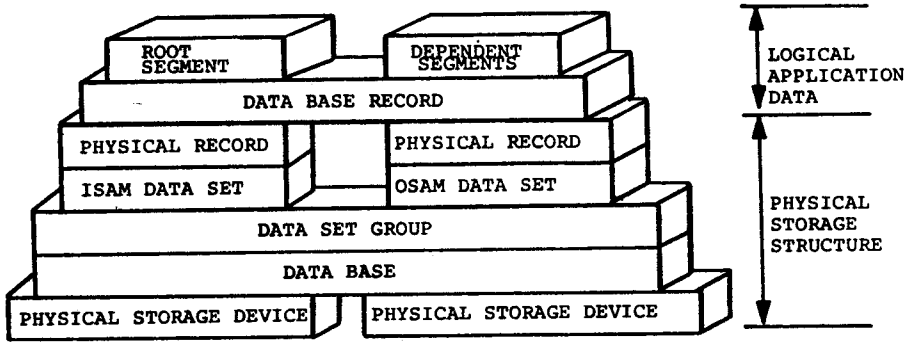


Figure 3. Data Language/I data base structure

Referring to Figure 3, it can be seen that each data base, in the physical sense, is composed of at least one data set group. Each data set group consisting of one or more data sets is dependent upon the organization of the data base for exact definition. The data base/data set group concept represents the Data Language/I expansion of the Operating System/360 data set concept in the storage of data. The data set group concept

allows Data Language/I to accommodate variable-length logical records even with the constraints of available storage devices. The user of a data base (that is, application program) is insensitive to the number of data set groups which comprise the data base (the physical structure of the data base) and which may change periodically based upon the processing and storage requirements of the data base. The user should view the data base as a collection of data base records, not of data set groups.

The descriptive information of logical data base record-segment relationships and the physical device and data set group description used by Data Language/I are stored apart from the data base and application program in a Data Base Description (DBD). The DBD is built using the data base description generation utility program and must be completed before data base creation or application program execution.

APPLICATION DEVELOPMENT AND STRUCTURING OF DATA LANGUAGE/I

When the user of the Data Language/I system initiates the definition of an application program to operate with Data Language/I, the following must be performed:

1. The definition of each Data Language/I data base in terms of its hierarchical structure and storage, and the creation or load of data into each data base in the batch processing environment using the capabilities of Data Language/I

2. The definition and construction of all batch processing programs and the control blocks which define how a program intends to use a data base

A detailed description of the events which must occur before execution of the Data Language/I control program with the user's application program and data bases follows. Figures 4 through 7 describe the component functions which are the user's responsibility. All block numbers refer to Figure 8.

1. The user must create a Data Base Description (DBD, Block 4) for each data base associated with an application program. He then uses the Data Base Description of the data base as input to the DBD generation utility program (Block 5) in the Data Language/I batch environment (Block 6) (or Operating System/360 batch environment). The resultant DBD is stored as a member of an OS/360 partitioned data set called the DBD library (Block 9). See Figure 4.

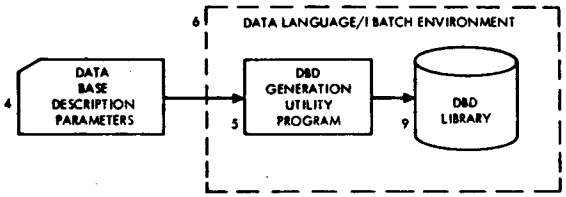


Figure 4. Data base description

2. Next, the user must create three items: First, he must create a data base creation (load) program (Block 1) for each data base. Second, he must create the description of each load program's data base requirements (Application Program Description, Block 8) according to the parameters of the Program Specification Block (PSB) generation utility program (Block 7). Third, he must use the data base creation program description (PSB, Block 8) as input to the PSB generation utility program (Block 7) in the Data Language/I batch environment (Block 6) (or Operating System/360 batch environment). The resultant PSB is stored as a member of an Operating System/360 partitioned data set called the PSB library (Block 11). See Figure 5.

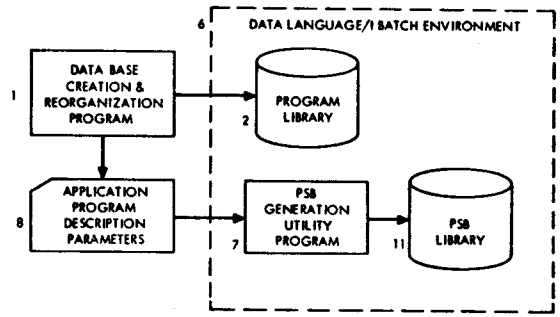


Figure 5. Data base creation

3. The user's data base load program may now be executed either for loading or for reorganizing the data base. The program requires the Data Base Description (DBD) for that particular data base, and the Program Specification Block (PSB) associated with the data base load program. See Figure 6.

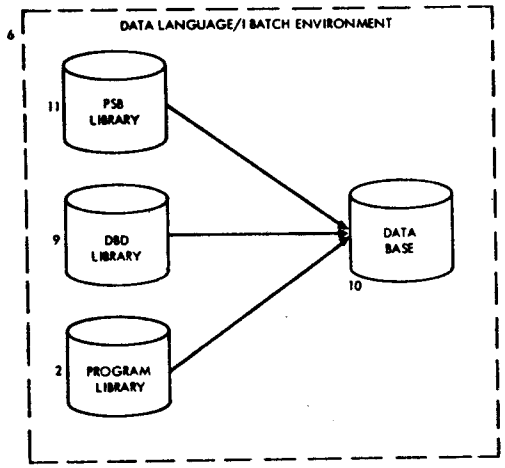


Figure 6. Creation or reorganization into batch environment

4. Once all data bases have been created, the user must create a PSB (Blocks 8 and 7) for each processing program. The user must place all programs which use Data Language/I into a user program library. The name of each PSB is identical to the name of the program with which it is associated. See Figure 7.

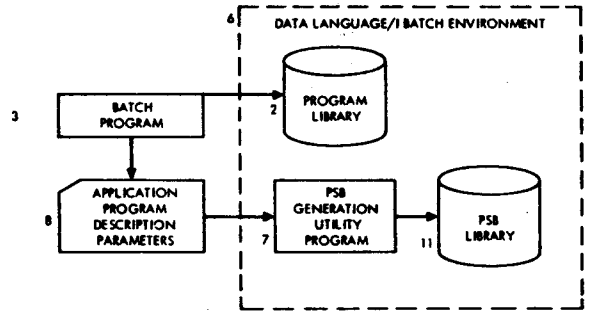


Figure 7. Storing in library

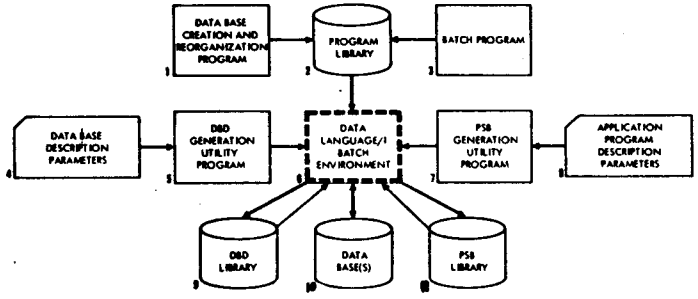


Figure 8. Events for IMS/360 use

DATA LANGUAGE/I

Data Language/I builds upon the facilities of two of the basic access methods of Operating System/360 Data Management. Basic Sequential Access Method (BSAM) has been adopted to offer the basic ability for processing data bases which have been stored sequentially on either 2301 drum, 2302 disk file, 2311 disk packs, 2314 disk facility, 2321 data cell, or 2400 magnetic tapes. In addition, Queued Indexed Sequential Access Method (QISAM) has been adopted to provide the capability of holding indexed sequential data bases on either 2302 disk file, 2311 disk packs, 2314 disk facility, or 2321 data cell.

To complement the facilities of QISAM, a new access method, called Overflow Sequential Access Method (OSAM), has been implemented. OSAM was developed to facilitate the sequential addition of fixed-length physical blocks on a multivolume direct access data set, and concurrently provide the capability to directly access and update existing blocks on the data set. The capability is used for handling the overflow data from ISAM records under Data Language/I.

Data Language/I Major Features

Data Language/I has many outstanding features:

1. It provides for a common source program interface between the application programs and the data bases they reference. This interface takes the form of a CALL statement. CALL statements are provided for both PL/I and COBOL. The format of the CALL statement and the features provided are almost identical regardless of which compiler language and data base are used. This results in a significant reduction in programmer training. It also removes from the application program any Operating System/360 data management definition.

2. Data Language/I also provides for the data descriptions associated with the data base to be retained as members of an Operating System/360 partitioned data set independent of application programs which use the associated data bases. This partitioned data set is known as the Data Base Description Library. By holding a data base description separate from the program, an application program is relatively independent of the organization of the data base. Thus, moderate changes in the data organization are possible without affecting the programs produced and maintained by an application programmer.

3. Prior to the advent of improved data base management, several programmers could use portions of a combined data base only if there were positive and continuous coordination between the several users. Furthermore, a change which affected any one of the users more than likely affected them all. Data Language/I offers a unique capability which allows a programmer to state which portions of a combined data base he wishes to be "sensitive" to. Within the constraint of a single logical sequence (sort order), the physical organization of a data base may be changed, or data may be added only by modifying the programs which are sensitive to the changed elements. Of course, if changes are required for data elements common to every program, they must all be modified. However, if changes are made to those elements unique to one (or perhaps a few) of the programs, only that fraction of the programs affected need be modified.

4. At the present time, Operating System/360 Data Management allows the programmer to describe data only if the total number of characters in a logical data record is less than one physical track of the direct access storage device selected. Thus the programmer must ultimately be aware of the characteristics of the device he is using for information storage. Data Language/I allows logical data base records to span one or more physical tracks, if necessary. This provides a functional capability through Data Language/I otherwise available only through custom programming.

5. The general trend is toward combining files which share common data elements. The common elements (including the sort key) are called the root segments, the remaining data elements pertaining to individual application programs are called dependent segments. By storing the root segment only once for any logical record, requirements are reduced. The data is logically represented as a hierarchy of segments, with the root segment at the highest level. Access to lower segments is accomplished by qualification from the higher levels in the hierarchy. To allow expeditious accessing of segments, sensitivity codes were introduced. Each application program indicates through these sensitivity codes those dependent segments which it is prepared to process. Thus, it is possible to have many different segments relating to a single root segment, yet, through the mechanism of the sensitivity code, to retain simple, uncluttered application programs which relate only to a single root-dependent segment combination.

6. Another feature is that of design compatibility. The facilities provided within Data Language/I support two logical data structures. The data base designer first describes the root segment, which consists of a field containing the highest level sort key and may contain one or more data fields always associated with the sort key and common to the root. If no further structure is provided, the data base thus created represents the simplest case of the simple data base with one segment type.

   If a more complex organization is required, the data base designer describes one or more dependent segment types which are logically dependent on the root segment. If a program is sensitive to a single root-dependent segment combination, it need not be complicated with code that concerns the other dependent segments. Although the application program will not be logically affected by segments to which it is insensitive, its execution time may be increased, since all segments are logically stored following their related root. This inefficiency when using dependent segments can be alleviated or remedied through the use of multiple secondary data set groups.

Data Language/I Rules

The following rules govern operation under Data Language/I:

1. There shall be only one root segment per data base record. This implies that there shall be only one sort key and, hence, only one sort order per data base.

2. The total length in bytes of any segment key field or identifier shall be equal to or less than 255 bytes.

3. Each segment type may be composed of one or more fields; however, there may be only one key field within a segment-type. The key fields of the segments determine the sort order of the segments within the data base.

4. The total number of the dependent segment-types under a root segment must not exceed 254.

5. Each segment, be it root or dependent, must be a single fixed length. The length may vary from segment-type to segment-type, irrespective of segment level, but a single named segment shall have a fixed length.

6. Up to 14 levels of dependency plus the root segment may be described in any single data base record.

7. A data base may consist of a single or multiple data set groups. The primary data set group contains at least the root segment. The secondary data set groups must all start with second level dependent segments. There can be only one primary data set group within a data base. A maximum of nine secondary data set groups can be defined in a data base.

The general structure of the batch processing region is as follows:

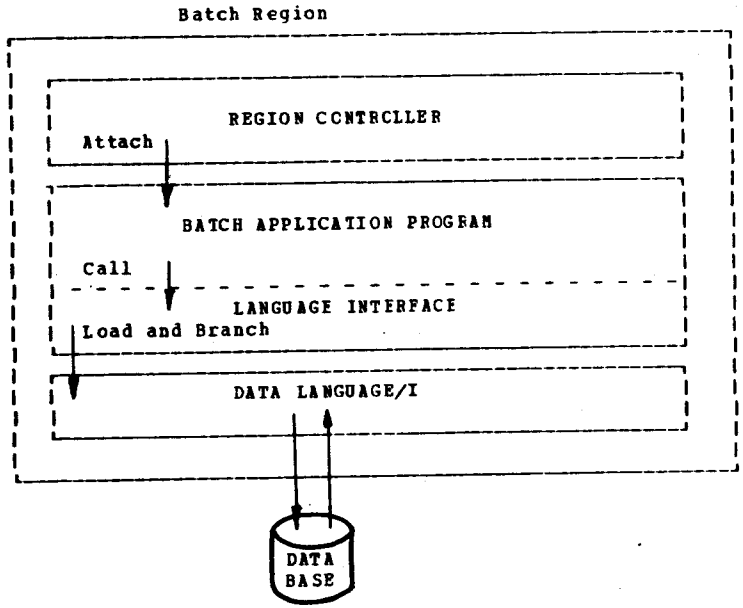


Figure 9. Organization and control flow in batch region

In the batch environment, the region controller performs the following operations:

1. Causes Data Language/I function capability to be loaded into the region
2. Causes all necessary blocks, tables, and Data Language/I control modules to be loaded into the batch region
3. Returns control to the batch program to start processing

In addition to Data Language/I data base requests, data management input/output operations may be performed within either type of batch region.

DATA BASE ORGANIZATION

TYPES OF DATA BASES

The organization of a data base is related directly to the hierarchical relationship of its segments. The segment of data is fundamental to Data Language/I and allows the structuring of any data base into either a simple or a complex hierarchical relationship. Neither the simplicity nor the complexity influences the type of access method which is used, although it may alter the desirability of one over the other. The following discussion is to assist the programmer to conceptualize his data base even though he need only consider those segments to which he is "sensitive".

The occurrence of dependent data on root information causes a dependency to exist in a data base. If the dependent information can be stratified, or if a dependent segment has segments dependent upon it, the file begins to assume a hierarchical relationship. The term "levels of information" is introduced to describe how far removed from the root segment a dependent segment is. The root segment is defined as containing the key and level-one information. The first dependent segment carries level-two information. A dependent segment which was in turn dependent upon the level-two dependent segment would be called a level-three segment, etc. Data Language/I allows 15 of these levels to be defined, along with many dependent segment-types within each of these levels. However, a table has been set aside with 255 entries in it. The root segment takes up one entry position; therefore, there may not be more than 254 other segment-types in the entire data base.

Simple Hierarchical Files

The simplest of all files consists of only a single segment-type. In Data Language/I terminology, this file consists solely of root segments and no dependent segments. Each segment in the file would have a fixed-length key field, and one or more fixed-length data fields accompanying that key. All of the fields would always be present. The file would be stored and retained in an order based on the values of the keys when taken as simple binary values. Further, the simple file would have fixed definitions for each field.

The simple file is the most common file in existence and in fact requires no hierarchical consideration. Even though the simple data base lacks the true hierarchical structure, it may be handled by Data Language/I and as such becomes the simplest form. This file structure produces fixed-length root segments

within logical data management records and these logical data management records are collected to form physical records.

Complex Hierarchical Files

The first instance of data file complexity usually occurs when control totals are appended to our simple file. If our simple file described an inventory of parts, each with its part number (key), a fixed-length alphameric description, a fixed-length quantity-on-hand field, and a fixed-length unit price field, we might like to insert some total records indicating the total dollar volume of a series of parts in a specific category and the total quantity on hand irdependent of part number. This would be done in one of two ways, depending on the type of storage media used to retain the files. See Figure 10.

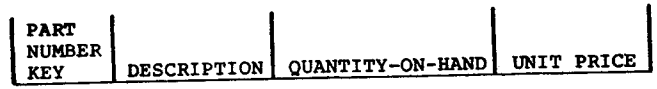
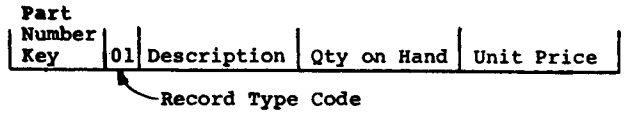


Figure 10. Simple physical file layout

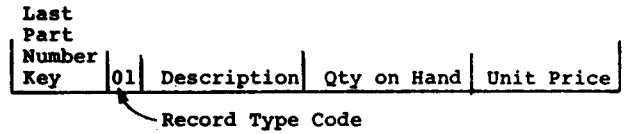
If the file were retained on tape, it would be necessary to rewrite it in its entirety whenever a single inventory item was used to fill an order. Given this situation, it is quite logical to embed the control totals at the appropriate point following the sequence of data they summarized. Thus, the detail information for a category would be read, and, after all the transactions to the category had been processed, the control totals for that category could also be processed, updated, and written out on the new tape drive. This is both traditional and practical, since a spare tape drive to hold the total separately is expensive, and since the entire tape must be rewritten during every processing cycle anyway. Therefore, it makes no sense to build a second simple file which would require a separate drive of its own just for the control totals.

Alternatively, a one-character field segment-type could be added to each entry and appended to the least significant end of the key. All of the detail data pertaining to a single type of line item in the inventory would be awarded one segment-type code, say the numeric value one. The control totals would

assume the key of the highest line item they summarized and have a record-type code field equal to any number greater than one. Thus, if the file is sorted or sequence-checked on the augmented key, it is in fact in numeric sequence on that key, even though the control totals may be embedded in the data. After a record has been read from the device and delivered to core storage, a simple test based on the type code field could allow the program to process detailed data or know that it was dealing with a summary segment. See Figure 11.



LAST PART NUMBER OF A PARTICULAR CLASS



CLASS TOTALS RECORD

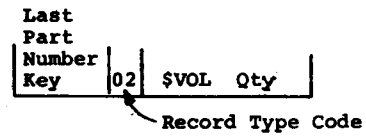


Figure 11 . Complex physical file layout

The "total record" is a simple case of the occurrence of a segment-type code field. Another instance of the segment-type code field will occur should two similar files be merged to allow common information to be held once and information unique to two different purposes to be subservient to it. Frequently,

files occur which have many fields in common. If these files also enjoy a common key and common sort order, they can be combined. The common information (root segment) is held once and the unique information (dependent segments) can be held in a compact form, since the key and related fields need be specified only once for each pair of records.

When several files have been combined to gain improved storage utilization or ease of processing, a second-order effect occurs. Not every programmer requires access, nor is even authorized to access, every different dependent segment-type. To solve this problem and eliminate the superfluous information (in the eyes of a programmer) that he does not need to handle, sensitivity codes are introduced. Each segment-type is assigned a unique name, and Data Language/I allows a programmer to state the names of the segments in the data base he wishes to "see", using an area within the Program Specification Block (PSB), called the Program Communication Block (PCB). At execution time, the Data Base Description (DBD) relates those segment names to the numeric segment-type codes stored with the data. A block of data is then read into core storage by Data Language/I. While Data Language/I has control, the segment-type code fields embedded in the segments of data just read are compared against the type codes of the segments to which the PSB has declared it is sensitive. If a block of data is obtained which contains no segment whose type codes match the programmer's sensitivity list, another read is initiated.

Internally, at execution time, Data Language/I keeps an identification table of segment names, type codes, and levels. Data Language/I returns status codes, level numbers, segment names, and segment keys to the program through the PCB to indicate the relationship to the hierarchy of the segment just obtained. This relationship is obtained by entering the identification table with the name of the segment just retrieved and comparing its level and type with the level and type previously obtained.

With these tools, data bases can be constructed from combined complex files made up of several different segment-types, all of which contain a limited header and control information and a minimum amount of redundant data. It should be noted that Data Language/I reads and sometimes rewrites segments to which the programmer is not sensitive.

STRUCTURE OF DATA BASES

The independence of the application program from the access methods, physical storage organization, and characteristics of the devices on which the data is stored is provided through a common source program linkage (consisting of a list of parameters which are addresses of PCB's, I/O functions, and segment identifiers) and a data base description. This common source program linkage and data base description allow the application program the ability to request Data Language/I to:

- Reference a unique segment (Note 1) (GET UNIQUE)
- Retrieve the next sequential segment (GET NEXT)
- Replace the data in an existing segment (REPLACE)
- Delete the data in an existing segment (DELETE)
- Insert a new segment (INSERT)

Note 1: "Segment" refers to a fixed-length data element containing one or more logically related data fields.

(The above calls are described in a later section of this manual.)

In the COBOL language, this common source program linkage uses the ENTER LINKAGE and the CALL verb to perform the input/output functions listed above. Application programs written in PL/I or Assembler language use similar statements to reference Data Language/I. Because of this approach to data reference, input/output operations and associated control blocks are not compiled into the application program. This removes dependency upon the currently available access methods and physical storage organizations.

Each data base description is created from user-provided statements of the logical and physical structure of each data base. These statements are input to an offline utility program. The result of the utility program is the creation and storage of a Data Base Description in the user-defined Data Base Description Library. This Data Base Description provides Data Language/I with a "mapping" from the logical structure of the data base used in the application program, to the physical organization of the data used by Operating System/360 data management. The logical data structure can be "remapped" into a different physical organization and this can be achieved

without program modification. Integration of other application data can also be added to this data base and still not cause a change to the original application programs. The concept of a data base description reduces application program maintenance caused by changes in the data requirements of the application.

Data Language/I provides for elimination of redundant data while providing integration or sharing of common data. The majority of the data utilized by any company has many interrelationships and hence many redundancies. For example, Manufacturing and Engineering have many pieces of data which would be useful to Quality Control; similarly, Purchasing and Accounting. If analysis of the number of types of segments shows that all the data cannot be placed in a single common data base, Data Language/I allows the user the additional capability of physically structuring the data over more than one data base. Before Data Language/I, personnel responsible for application programs frequently were not able, nor did they have the time, to integrate other data with their own to eliminate redundancies without the necessity of a major rewrite of the application programs involved.

Another capability of Data Language/I protects each application of a multiapplication data base through the concept of "sensitive" segments. When operating against a Data Language/I data base, only the data segments that are predefined as sensitive are available for use in this application. Each application using the data base can be sensitive to its unique subset of "sensitive" segments. Where an application program has defined "sensitivity" to a subset of segments within a data base record, modification and addition of nonsensitive segments do not affect the processing capability of the program. In addition, any application program can be restricted to "read only" operations against its sensitive segments.

Data Language/I - Data Base Organization

The data base structure is best described by providing an example. Figure 12 depicts the hierarchical relationship for a company data base made up of Engineering data, Inventory data, and Purchasing data which could be typical of any company. All this is based on part master (part number) data.

A data base is composed of data base records. A data base record is a collection (a variable number) of hierarchically related, fixed-length data elements, called "segments". A root segment is the highest hierarchical segment in the data base record. A dependent segment is a segment that relies on at least the root segment for its full hierarchical meaning. It

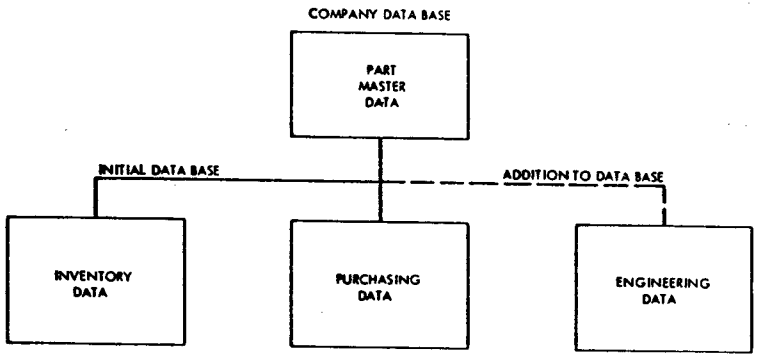


Figure 12 . Company data base hierarchical data relationship

is therefore always at a lower hierarchical level than the root segment. There can be 255 segment-types within a data base and 15 levels of segment hierarchy within a data base record.

Details of the segment of this data base example will be shown for the Inventory and Purchasing data contained in the initial Company data base segment structure. See Figures 13 and 14. This logical structure may be physically stored in either of Data Language/I's organizations:

- Hierarchical Sequential: The Operating System/360 Basic Sequential Access Method (BSAM) is used to implement the Hierarchical Sequential organization. Storage medium may be tape or direct access storage. See Figure 15.

- Hierarchical Indexed Sequential: The Operating System/360 Indexed Sequential Access Method (ISAM) and a unique access method of Data Language/I, called Overflow Sequential Access Method (OSAM), are used to enhance the capabilities of ISAM and to implement the Hierarchical Indexed Sequential organization. The storage medium must be direct access storage. See Figure 16.

After the initial data base details shown in Figures 13 through 17, the addition of engineering data in Figures 18 through 22 may be accomplished. As illustrated, the data base segments may be organized or reorganized into the Hierarchical Indexed Sequential organization or the Hierarchical Sequential organization. Note that, even with the addition of engineering

data, the expansion of the data base may be accomplished without altering the existing processing programs which reference the data base.

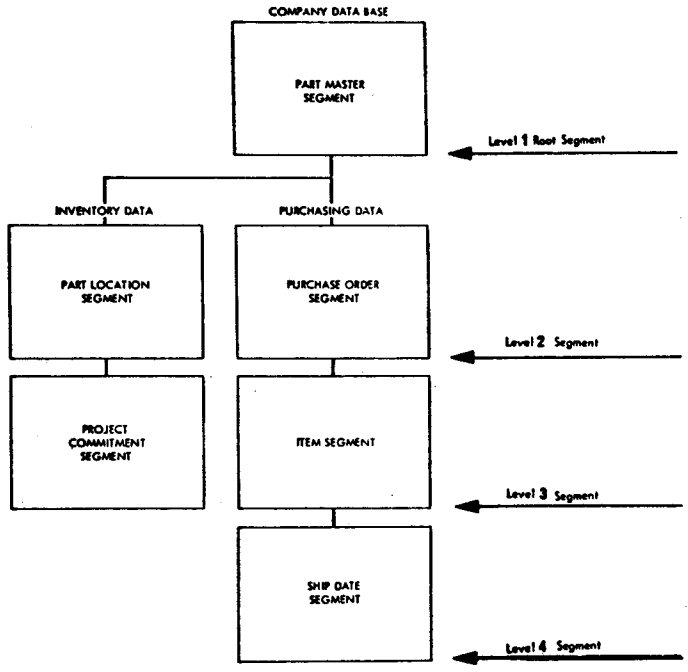


Figure 13. Company data base segment Logical Hierarchical relationship - Inventory and Purchasing data

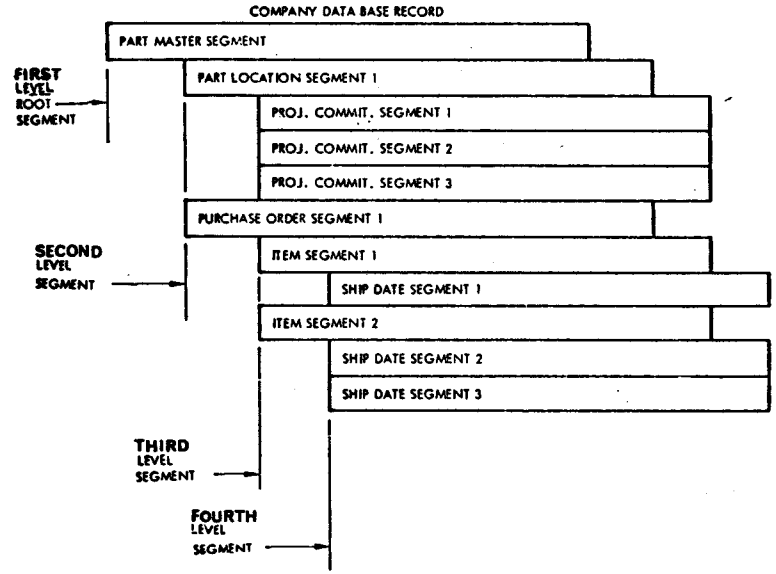


Figure 14. Company data base record segment level structure

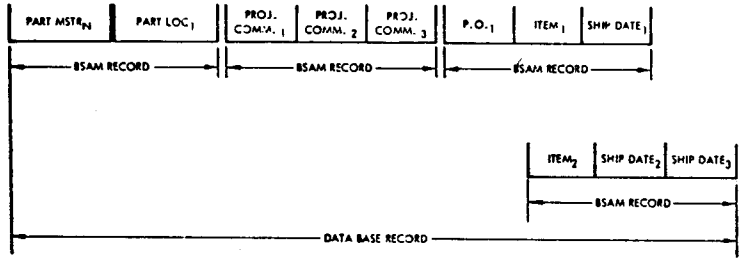


Figure 15. The Nth company data base record stored in the Hierarchical Sequential organization

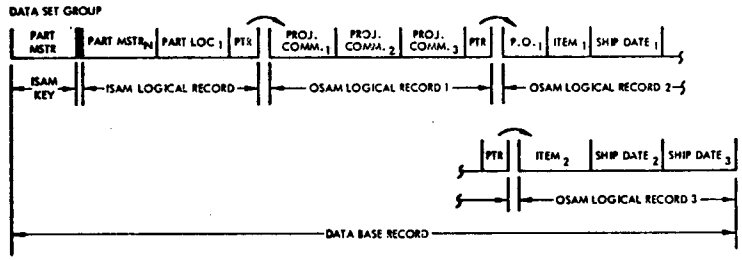


Figure 16. The Nth data base record in Hierarchical Sequential organization - single data set group

The highest level (level one) segment or root segment is the Part Master Segment. All segments immediately subordinate to the root segment are called second level dependent segments: Part Location Segment and Purchase Order Segment. Third level dependent segments are related to the second level dependent segments. In this example, Project Commitment Segment 1 is related to Part Location Segment 1, and Item Segment 1 is related to Purchase Order Segment 1. Fourth level dependent segments are related to the third level dependents. All the segments in Figure 14 constitute a data base record.

If the Hierarchical Sequential organization is chosen for the data base, Figure 16, each segment type is fixed length within a data base record and is stored in physical sequence according to its hierarchical relationship.

Figure 16 represents the Nth data base record depicted by the data base in Figure 13. The Part Master root segment has one occurrence of the second level dependent Part Location segment-type. For inventory purposes, this part has only one storage location. The second level Part Location segment-type has three occurrences of the third level Project Commitment segment-type. There are three projects in this Company that have commitments against the inventory of this part.

11-31-71

There is one second level Purchase Order segment-type, the root segment of which is the Part Master. This second level segment-type has two occurrences of the third level dependent segment-type: Item Segment 1 and Item Segment 2. They in turn have subordinate or fourth level dependent segment-types: Ship Date Segment. The Item segments are the purchased components for this particular part (or assembly). Each has a particular shipping date.

All data base records are stored sequentially in sort sequence of the root segments. The only direct data reference provided with the Hierarchical Sequential organization is to the first root segment in the first data record of the data base. All subsequent reference is sequential.

If the Hierarchical Indexed Sequential organization is chosen, direct reference is provided to each root segment (and therefore to each data base record) within a data base. When the data base is created or reorganized, the key of each root segment is an ISAM logical record key. As many segments (the root and its dependents) are stored as will fit within the ISAM logical record. If storage for additional segments within the data base record is required, a relative block pointer is placed in the ISAM logical record. This pointer relates the

ISAM record to one or more OSAM records which contain the remaining segments of the data base record (Figure 16).

When the data base is created or reorganized, each data base record starts as an ISAM logical record and may overflow into one or more OSAM logical records. Ncte in Figure 16 that the two data sets, ISAM and OSAM, represent a data set group. Reference to segments within the data base record is sequential.

As shown in Figure 16, the ISAM logical record consists of two segments: the Part Master root segment and the second level Part Location dependent segment. At the end of the ISAM logical record is a pointer to the first OSAM record. The first OSAM logical record consists of three second level Project Commitment dependent Segments 1, 2, and 3. The second and third OSAM logical records contain the remainder of the data base record.

An additional capability of the Hierarchical Indexed Sequential organization is to prcvide direct access to all root segments and to all or some first level dependent segment-types. This capability is provided through the use of multiple ISAM and OSAM data sets (multiple data set groups) (See Figure 17).

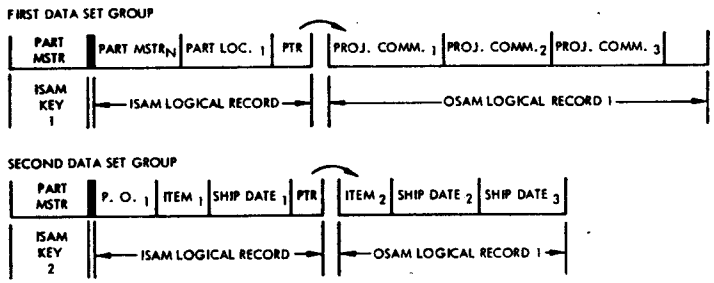


Figure 17.. The Nth data base record in Hierarchical Indexed Sequential organization - multiple data set groups

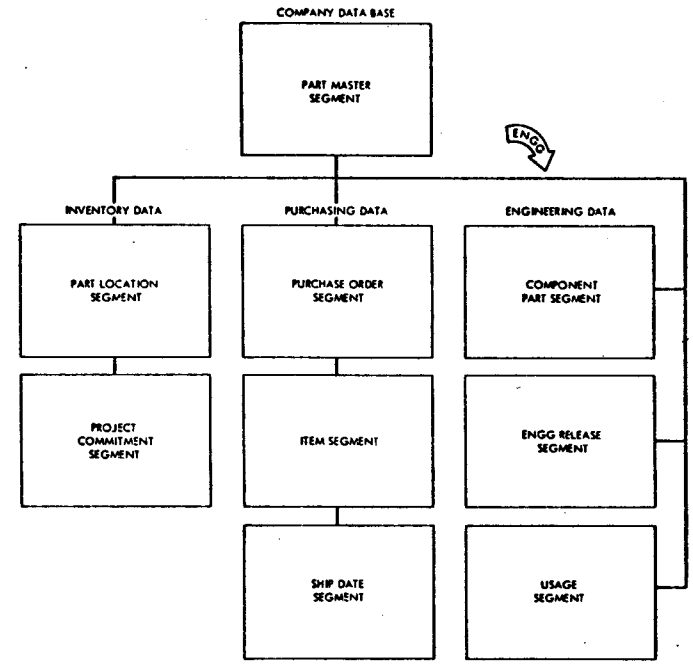


Figure 18. Company data base segment Logical Hierarchical relationship - Inventory and Purchasing data - Engineering data added

The Part Master root segment, the Part Location dependent Segment 1, and the Project Commitment dependent Segments 1, 2, and 3 are contained within one data set group. The Purchase Order second level dependent segment-type and its remaining dependent segments are contained within a second data set group. This allows direct reference to the first Purchase Order segment-type within each data base record as well as to the root segment: Part Master. A maximum of ten data set groups is permitted in the Hierarchical Indexed Sequential organization.

An example is now depicted which illustrates an addition to the Company data base of Engineering data. It is assumed that the Inventory and Purchasing applications are not changing. The addition to the Company data base is the integration of the Engineering application as shown in Figure 18.

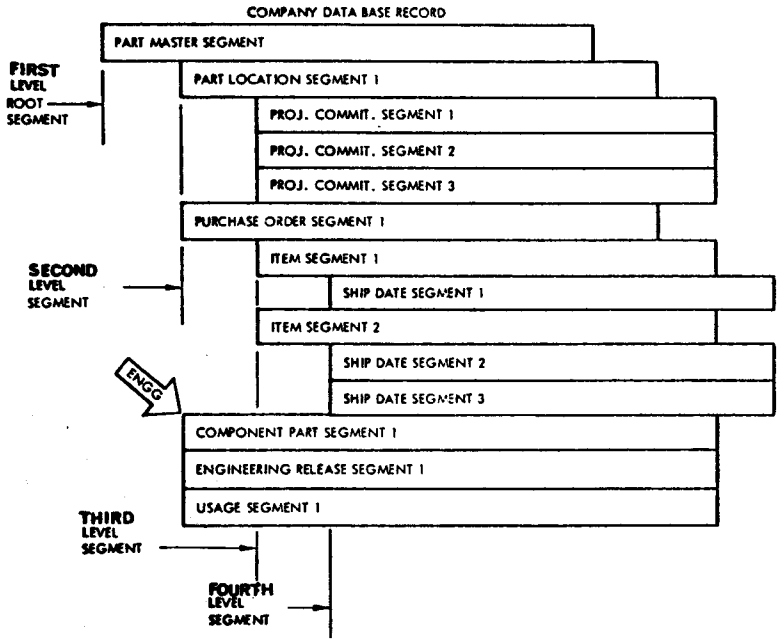


Figure 19 . Company data base record segment level structure - Engineering data added

The responsible user personnel may extend the Company data base description and insert the Engineering data segment structure. Then the existing company data base and the engineering data segments are merged to create the new company

data base. Figure 19 depicts a segment level picture of the data base record.

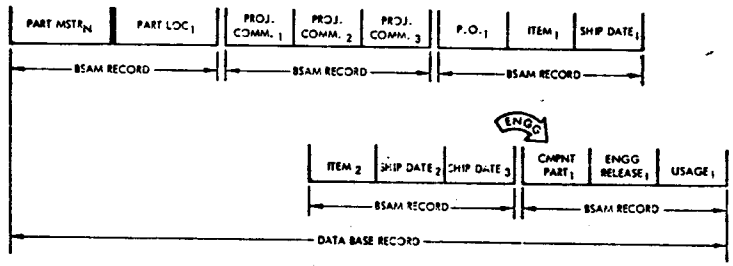


Figure 20 . The Nth company data base record stored in the Hierarchical Sequential organization - Engineering data added

Figures 20, 21, and 22 illustrate how the new data base may be physically stored. Note in Figure 22 a third data set group is added. This is accomplished without disturbing either the Inventory or Purchasing application data.

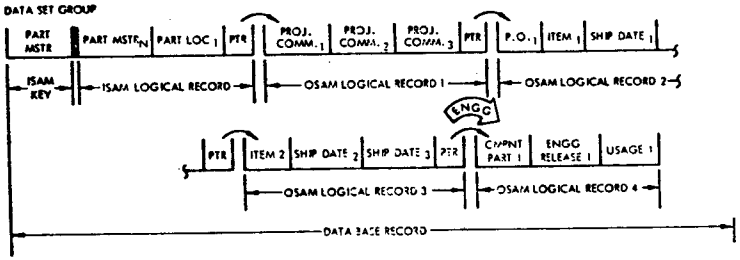


Figure 21. The Nth data base record in Hierarchical Indexed Sequential organization - single data set group - Engineering data added

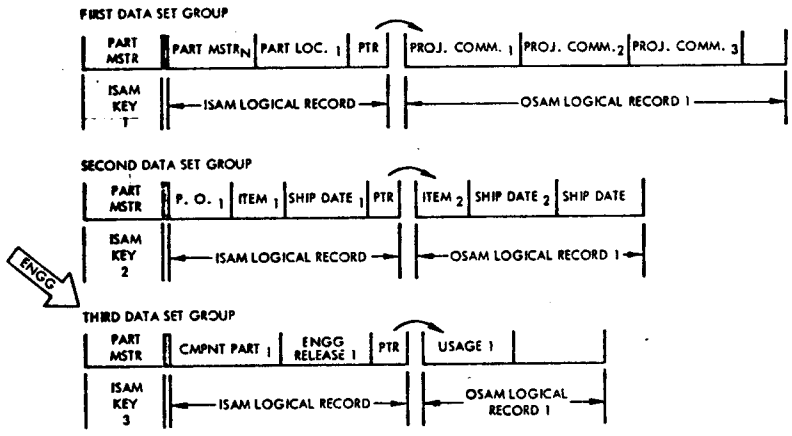


Figure 22. The Nth data base record in Hierarchical Indexed Sequential - organization - multiple data set groups - Engineering data added

DATA BASE PROCESSING

Introduction

Data base processing with Data Language/I is accomplished with the data storage organizations just described and a set of input/output functional requests used by application programs.

An input/output functional request is composed of a CALL statement with a parameter list. The parameter list provides the information, which is assembled by the application program, to describe a particular input/output function and the element of data operated upon. The element of data operated upon by any Data Language/I input/output request is termed a segment. One and only one segment may be operated upon with a single input/output request or CALL.

A segment is composed of one or more data fields, one of which is considered the key field. Each particular segment-type has a fixed length and format definition.

The parameters contained within any input/output functional request include the addresses of:

- The input/output function
- The definition of the data base to be operated upon
- The segment input/output area into or out of which the segment of data is moved
- The identifiers used to describe the segment of data to be operated upon

The input/output functions provided by Data Language/I are GET UNIQUE, GET NEXT, GET NEXT WITHIN PARENT, DELETE, REPLACE, and INSERT. Remember that each of these functions, within a single request, operates upon only one segment of data. Thus, GET UNIQUE causes the retrieval of a specific segment described by the identifiers in the CALL into the defined segment input/output area. The INSERT operation causes the segment residing in the segment input/output area and described by the segment identifiers to be added to a data base. The identifiers in a functional request used to describe the segment of data to be operated upon are called segment search arguments (SSA). A segment search argument includes the one- to eight-character symbolic name of the segment-type, the one- to eight-character symbolic name of the segment key field, an algebraic operator, and the value of the desired key field. Let us again consider Figure 19. The generic name of the part master segment might be PARTMAST and its key field name might

be PARTNUMB (part number). Thus the segment search argument for GET UNIQUE of the part master segment with part number equal to 12345 would look like:

PARTMAST(PARTNUMB =12345)

The portion of the segment search argument within the parentheses is called a qualification statement.

For unique retrieval or addition of a root segment, only one segment search argument must be provided. However, the unique retrieval or insert of a dependent segment requires multiple segment search arguments to be provided in the functional request. Each segment search argument in the list describes a segment upon which the dependent segment to be operated upon is dependent. The SSA's for a given Data Language/I call must be in proper hierarchical relationship. If the generic name of a purchase order segment-type is PURCHASE, its key field name is PURCHNO, and there is a purchase number XYZ for part 12345, unique retrieval is accomplished by two segment search arguments included within the parameter list of the Data Language/I CALL:

PARTMAST(PARTNUMB=12345)

PURCHASE(PURCHNO=XYZ)

The definition of the data base to be operated upon is provided in each Data Language/I CALL by a control block called a Program Communication Block (PCB). All PCB's used by a particular application program for data base operations are contained within the PSB for that program. At execution time, the 'base' addresses of the PCB's are passed to the application program. Each PCB contains the one- to eight-character name of the DBD associated with the data base.

Data Base Creation

A data base is created by an application program issuing Data Language/I calls to insert data base records presorted by the key field of the root segment. When a data base record is composed of more than the root segment, all segments within the data base record must be presorted by their hierarchical relationship and key field value. Consider the process of inserting the segments of a company data base record described in Figure 19. First, the Part Master (root) segment is inserted. The Part location segment (first level dependent) is inserted next. Then the three Project Commitment segments

sorted by key field value are inserted. This continues with the Purchase Order Segment, Item Segment 1, Ship Date Segment 1, Item Segment 2, etc., until all segments are inserted. If this data base record represented the segments of data associated with part number X, the segments to be inserted into the data base next would be those associated with part number X + 1.

The INSERT function is used to create or load (recreate or reorganize) a data base. Prior to the execution of a Data Language/I CALL to cause segment insertion, the segment to be inserted must be moved into a segment input/output area and the proper list of segment search arguments must be assembled. Let us assume we are creating the company data base and we are about to load the segments of data associated with part number 12345. The first three segments to be loaded would be part master, part location 1, and project commitment 1. The associated segment search arguments and input/output work area contents for these three Data Language/I INSERT CALLS are:

PART MASTER SEGMENT INSERTION

SSA - PARTMAST
1

Work Area - (containing part master segment)

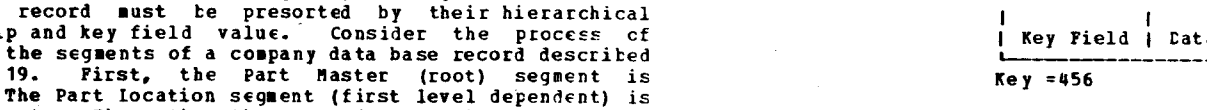
Key =12345

PART LOCATION 1 SEGMENT INSERTION

SSA - PARTMAST(PARTNUMB =12345)
1

SSA - PARTLOC
2

Work Area - (containing part location segment)



Key =456

PROJECT COMMITMENT 1 SEGMENT INSERTION

```
SSA -  PARTMAST(PARTNUMB =12345)
   1

SSA -  PARTLOC(LOCATION =456)
   2

SSA -  COMMIT
   3

Work Area - (containing project commitment segment)

        |           |           |           |
        | Key Field | Data Field| Data Field|
        |___________|___________|___________|

        Key =6185
```

Notice that the segment search arguments of a Data Language/I CALL for inserting a segment into a data base must describe the complete hierarchical path to the segment. Also notice that the last segment search argument within each INSERT CALL does not and must not include the qualification statement portion. The qualification information is taken from the image of the segment in the input/output work area.

Data Base Retrievals

The retrieval of segments within a data base is accomplished by the three GET functions: GET UNIQUE, GET NEXT, and GET NEXT WITHIN PARENT. GET UNIQUE provides for the retrieval of a specific segment by direct reference into the data base. GET NEXT provides for sequential segment retrieval. Usually the GET NEXT function is used after a GET UNIQUE or GET NEXT which has provided "positioning" to a unique segment within the data base. However, a GET NEXT may be used without positioning being supplied by a previous GET UNIQUE or GET NEXT. If Data Language/I has no position established within a data base when a GET NEXT call is issued, the request is satisfied by proceeding from the beginning of the data base. The GET NEXT WITHIN PARENT allows sequential retrieval of all segments subordinate to a parent segment. An example using Figure 19 would be all item and ship date segments within the company data base for a given part and purchase order. The parent segment is a unique purchase order segment and parentage must have been previously established with a GET UNIQUE or GET NEXT request.

Once all the items and ship date segments for a given part and purchase order have been retrieved by a succession of GET NEXT WITHIN PARENT requests, an indication is returned to the application program. This indication provides definition of the end of subordinate segments for the particular part and purchase order.

In addition to direct retrieval of a unique segment and sequential retrieval of segments, an ability to sequentially skip from one segment to another of a common type is provided. Let us assume that we wish to retrieve all purchase order segments within a particular part master segment. However, we do not wish to retrieve the segments subordinate to each purchase order segment (that is, item and ship data segments). The first purchase order segment would be retrieved with a call where the function equals GET UNIQUE. The segment search arguments would be:

```
SSA -  PARTMAST(PARTNO =12345)
   1

SSA -  PURCHASE
   2
```

The remainder of the purchase order segments would be retrieved with Data Language/I calls where the I/O function parameter equaled GET NEXT. However, the Data Language/I calls would have a segment search argument

```
SSA -  PURCHASE
   1
```

In summary, the segment search arguments for all GET UNIQUE calls must start with reference to the root segment level. The GET NEXT calls may be used with or without segment search arguments. The segment search arguments for a GET NEXT call may start at any segment level. All SSA's within a single Data Language/I call must be in proper hierarchical order (that is, SSA for root first, SSA for first level dependent second, etc.).

Data Base Updates

The updating of data within a segment of a data base is performed through the REPLACE input/output function. Before a Data Language/I call to replace a segment may be executed, the segment to be updated must be retrieved through a CALL with a GET function. The GET functions which may be specified are those previously discussed, but must include the addition of a HOLD definition (GET HOLD UNIQUE, GET HOLD NEXT, or GET HOLD NEXT WITHIN PARENT). The REPLACE function must then be

executed in the next CALL against the data base. Any intervening CALLS against the same data base cause the rejection of the subsequent REPLACE call. No SSA's are permitted with the REPLACE function. The key field of the segment to be updated through the REPLACE function CALL must not be modified.

Data Base Deletions

The deletion of an entire segment (all fields) within a data base is performed through the DELETE input/output function. Before a Data Language/I call to delete a segment may be executed, the segment to be deleted must be retrieved through a GET HOLD call. The DELETE function must be executed as the next CALL against the data base or the DELETE function is rejected. No SSA's are permitted with the DELETE function.

Data Base Insertions

The addition or insertion of a new segment (all fields) into an existing data base is performed through the INSERT input/output function. The techniques used for performing an INSERT function to add a segment to an existing data base are identical to those used with the INSERT function when creating a new data base. Remember that the addition of a dependent level segment is not permitted unless all parent segments in the complete hierarchical patch already exist in the data base. An example referring to Figure 19 would be the addition of an item segment subordinate to a particular purchase order segment. The purchase order segment must already exist in the data base or be added before any item segments subordinate to that purchase order segment may be added.

Program Specification Block (PSB)

Associated with every batch processing program is a Program Specification Block (PSB). This control block describes the use of data bases by the associated application program. The PSB is constructed by the Application Programmer and placed in a partitioned data set generically termed the PSB library. A utility program is supplied to assist in the generation of each PSB. Each PSB is composed of one or more sub-blocks called Program Communication Blocks (PCB's). There exists one PCB for each data base with which the batch program intends to interface. Each data base PCB describes the segments which the associated program is sensitive to and the mode of processing the associated program will utilize on the data base. Processing modes include data base creation, retrieval, deletion, update, and addition. The PCB also includes the

symbolic name of the data base with which it is associated. The PSB and its PCB's exist external to their associated batch program. However, these blocks are used by the program in executing Data Language/I calls. The addresses to these blocks are passed to the associated batch program upon entry to the program. The address of a PCB associated with a given data base is subsequently used by the program when issuing a Data Language/I call. The PCB address becomes a parameter in the Data Language/I call.

Data Base Segment Sensitivity

The preceding paragraph described the use of PCB's to enable an application program to execute Data Language/I calls. The PCB contains the one- to eight-character name of the associated data base. You must recognize that the PCB supplies to Data Language/I the logical definition of the data base upon which the requested input/output operation is to be performed (the data base name). The PCB also describes the processing mode the associated application processing program intends to use upon the data base. However, the most important elements of data which the PCB supplies to Data Language/I are the names of the segments of data within the data base upon which the application processing program intends to operate. These represent the segments of data to which this application processing program is sensitive. Only the segments of data named in the PCB may be retrieved, updated, deleted, or added to the data base.

This concept of segment sensitivity has considerable importance with regard to the impact that the addition of new segment types into an existing data base has upon existing application programs. A data base can be expanded with new segment-types by the dumping of the data base, the generation of a new data base description incorporating the new segment-types and the old, and the reloading of the data base with the new data base description. The new data base now contains the old and the new segment-types. The existing application programs which operated upon the old segments in the data base are sensitive only to the old segment-types. The new segment-types appear 'invisible' to the existing application processing programs. No modification is required of the existing application processing programs to operate upon the expanded data base. Of course the existing application programs, since they are insensitive to the new segment-types, cannot operate upon the new segment-types. Presumably new application processing programs would be incorporated to operate with the existing programs to maintain the new segment-types. A significant benefit of the concept of sensitivity is the evolutionary expansion of the data contained within a data base

with minimal impact upon existing application processing programs.

Data Base Segment Definition

Each segment-type within a data base is defined at Data Base Description generation. The characteristics of the segment - length, fields, key field, etc. - are defined. It may often be a considerable task to determine the best structure of hierarchically related segments to use in defining applications data. Several guidelines might be appropriate.

1. Each Data Language/I data base has one root segment. The key field of the root segment is the primary sort key of the data base.
2. The structure (fields) within a root segment should represent data which occurs once per data base record.
3. First level dependent segments may occur zero to n times per root segment. The data within a dependent segment-type should occur zero to n times for one occurrence of the root segment data. Each dependent segment-type represents a lower level sequence of data.
4. Although a data base has only one sort sequence, other sort sequences or cross-reference relationships may be required. These can be accomplished with other data bases of other sort sequences.

APPLICATION PROGRAM STRUCTURE

Application programs which execute using Data Language/I may be written in any one of the following Operating System/360 programming languages: Assembler Language, COBOL, or PL/I. It is intended, however, that the programmer be able to benefit from the power of high level languages for processing and the power of Data Language/I for data manipulation. Thus this discussion is oriented toward COBOL or PL/I.

The structural requirements put upon any application program can be grouped into major areas and must be considered by every programmer.

- entry
- exit
- calls
- parameters for calls
- segment I/O area
- segment search arguments

The names listed below and shown in the previous example are standard and must be used by the programmer.

COBOL	PL/I	ASSEMBLER	STATEMENT USED WITH
DLITCBL	DLITPLI	ANY	ENTRY
CBLTDLI	PLITDLI	CBLTDLI	CALL

BATCH PROGRAM STRUCTURE

COBOL Batch Program Structure

Figure 23 illustrates in outline form all the fundamental parts in the structure of a batch program. Care should be taken to ensure that each item is considered when designing a batch program.

```
REF
NO.
      ENVIRONMENT DIVISION
          •
          •
      DATA DIVISION
      WORKING STORAGE SECTION
 1        77 FUNC-DB-IN    PICTURE XXXX VALUE'GU  '.
          77 FUNC-DB-OUT   PICTURE XXXX VALUE'REPL'.
          77 FUNC-DB-NEXT  PICTURE XXXX VALUE'GHN '.
          •
 2        01  SSA-NAME
 3        01  MAST-SEG-IO-AREA
          01  DET-SEG-IN-AREA
      LINKAGE SECTION
 4        01  DB-PCB-MAST
          01  DB-PCB-DETAIL

      PROCEDURE DIVISION

 5    ENTRY 'DLITCBL' USING DB-PCB-MAST,DB-PCB-DETAIL.
          :
 6    CALL 'CBLTDLI' USING FUNC-DB-IN, DB-PCB-DETAIL,
               DET-SEG-IN-AREA,SSA-NAME.
          :
 7    CALL 'CBLTDLI' USING FUNC-DB-IN, DB-PCB-MAST,
               MAST-SEG-IO-AREA, SSA-NAME.

 8    CALL 'CBLTDLI' USING FUNC-DB-NEXT,DB-PCB-MAST,
               MAST-SEG-IO-AREA.
          :
 9    CALL 'CBLTDLI' USING FUNC-DB-OUT, DB-PCB-MAST
               MAST-SEG-IO-AREA.
          •
          •
10    RETURN

11    COBOL - LANGUAGE INTERFACE
```

Figure 23. COBOL batch program structure

Figure 23 is a general illustration of the significant parts in the design of a COBOL batch program which might retrieve data from a detail file to update a master data base.

The following explanation relates to the reference numbers along the left side of Figure 23.

1. A 77 level or 01 level working storage entry defines each of the CALL functions used by the batch program. Each picture clause is defined as four alphameric characters and has a value assigned for each function (for example, 'GUbb').

2. An 01 level working storage entry defines each segment search arguments used by a application program. An example of SSA definitions, with lower case b's representing blanks, is:

```
01  SSA-NAME  02 SEG-NAME  PICTURE X(8) VALUE 'ROOTbbbb'.
    02 SEG-QUAL  PICTURE X  VALUE '('.  02 SEG-KEYNAME
    PICTURE X(8)  VALUE 'KEYbbbbb'.  02 SEG-OPERATOR
    PICTURE XX  VALUE 'b='.  02 SEG-KEY VALUE  PICTURE
    X(6)  VALUE 'vvvvvv'.  02 SEG-END-CHAR  PICTURE X
    VALUE ')'.
```

When the above COBOL syntax is decoded, it will be in a data string as follows:

```
ROOTbbbb(KEYbbbbb=vvvvvv)
```

3. An 01 level working storage entry defines the program segment I/O area.

4. An 01 level linkage section entry describes the PCB entry for every input or output data base. It is through this linkage that a COBOL program may access the status codes after a Data Language/I CALL.

5. This is the standard entry point in the procedure division of a batch program. After the region controller has loaded and completed the PSB and one or more DBD's for the program into the batch region, it gives control to this entry point. The PSB contains all the PCB's used by the program. There must be a PCB name for every PCB that is used by the batch program.

```
ENTER LINKAGE.
CALL
~~ENTRY~~ 'DLITCBL' USING pcbname-1,....pcbname-n.

ENTER COBOL.
```

6. and 7. These are typical CALLs used to retrieve data from a data base using a qualified search argument.

```
ENTER LINKAGE.

CALL 'CBLTDLI' USING function, pcbname,
        segment-I/O-area,
        segment-search-argument.

ENTER COBOL.
```

8. This is a typical CALL used to retrieve data from a data base using an unqualified search. This CALL is also a HOLD call for a subsequent delete or replace.

```
ENTER LINKAGE.

CALL 'CBLTDLI' USING function, pcbname,
        segment-I/O-area.
ENTER COBOL.
```

9. This CALL is used to replace data from a batch program onto a data base.

10. This RETURN causes the batch program to return control to the Region Controller. The format is:

```
ENTER LINKAGE.

RETURN.

ENTER COBOL.
```

11. A language interface is provided for all programming languages. This module is link-edited to the batch program and provides a common interface to Data Language/I.

PL/I Batch Program Structure

```
REP
NO.
      /* ---------------------------------------- */
      /*              ENTRY POINT                 */
      /* ---------------------------------------- */
1     DLITPLI:PROCEDURE (DB_PCB_MAST,DB_PCB_DETAIL)
          OPTIONS(MAIN);
      /* ---------------------------------------- */
      /*          DESCRIPTIVE STATEMENTS          */
      /* ---------------------------------------- */
2     DECLARE FUNC_DB_IN CHARACTER(4) INITIAL('GUbb');
      DECLARE FUNC_DB_OUT CHARACTER(4) INITIAL('REPL');
      DECLARE FUNC_DB_IN CHARACTER(4) INITIAL('GHNb');
        •
3     DECLARE SSA_NAME CHARACTER(26);
4     DECLARE MAST_SEG_IO_AREA ......;
      DECLARE DET_SEG_IN_AREA  ......;
        •
5     DECLARE 1 DB_PCB_MAST,
      DECLARE 1 DB_PCB_DETAIL,
        •
6     DECLARE THREE FIXED BINARY INITIAL(3);
      DECLARE FOUR FIXED BINARY INITIAL(4);
```

Figure 24. (Continued on Page 47.)

```
     /* ------------------------------------------ */
     /*    MAIN PART OF PL/I BATCH PROGRAM         */
     /* ------------------------------------------ */
       •
7    CALL PLITDLI(FOUR,FUNC_DB_IN,DB_PCB_DETAIL,
         DET_SEG_IO_AREA,SSA-NAME);
       •
8    CALL PLITDLI(FOUR,FUNC_DB_IN,DB_PCB_MAST,
         MASTER_SEG_IO_AREA,SSA_NAME);
       •
9    CALL PLITDLI(THREE,FUNC_DB_NEXT,DB_PCB_MAST,
         MAST_SEG_IO_AREA);
       •
10   CALL PLITDLI(THREE,FUNC_DB_OUT,DB_PCB_MAST,
         MAST_SEG_IO_AREA);
       •
11   END DLITPLI;

12   PL/I - LANGUAGE INTERFACE
```

Figure 24. General PL/I batch program structure

Figure 24 is a general illustration of the significant parts in the design of a PL/I batch program which might retrieve data from a detail file to update a master data base. A structure similar to the one shown must be used to create a data base in a batch region.

The following explanation relates to the reference numbers along the left side of Figure 24:

1. This is the main standard entry point to a PL/I batch program. After the region controller has loaded and completed the PSB and one or more DBD's for the program in the batch region, it gives control to this entry point. The PSB contains all the PCB's used by the program. There must be a PCB name for every PCB that is used by the batch program.

NOTE: When link-editing your compiled PL/I program with the language interface, your load module ENTRY must be either IHESAPB or IHESAPD and your load module member should be the name of your PL/I program. This ensures that PL/I does its house-keeping properly and passes the PCB's properly to the problem program.

Example as shown.

2. By declaring, each working area defines each of the CALL functions used by the PL/I batch program. Each character string is defined as four alphameric characters, with a value assigned for each function (for example, 'GUbb'). Other constants and working areas may be defined in the same manner.

3. This working area defines all the segment search arguments used by the problem program. This SSA has been defined as a character string. SSA_NAME can be broken down with other DECLARE statements and then in the problem program, where needed, concatenated together.

   Example: (lower case b's represent blanks)

```
DECLARE SSA_NAME        CHARACTER(26);
   "    SEG_NAME        CHARACTER(8) INITIAL('ROOTbbbb');
   "    SEG_QUAL        CHARACTER(1) INITIAL('(');
   "    SEG-KEYNAME     CHARACTER(8) INITIAL('KEYbbbbb');
   "    SEG_OPERATOR    CHARACTER(2) INITIAL('b=');
   "    SEG_KEY_VALUE   CHARACTER(6) INITIAL('vvvvvv');
   "    SEG_END_CHAR    CHARACTER(1) INITIAL(')');
```

   An example in the problem program where SSA_NAME lower levels are concatenated together:

```
SSA_NAME=SEG_NAME||SEG_QUAL||SEG_KEYNAME||
  SEG_OPERATOR||SEG_KEY_VALUE||SEG_END_CHAR;
```

   NOTE: Structure similar to COBOL levels may be used with discretion (only two levels), but PL/I might be confused that this is fixed or floating point arithmetic when it is intended to be character string.

4. A working storage area entry defines the program segment I/O area.

5. A level 1 declarative (similar to COBOL's linkage section) describes the PCB entry for every input or output data base. It is through this description that a

PL/I program may access the status codes after a Data Language/I CALL.

6. This is a descriptive statement used to identify a binary number that represents the "parameter count" of a CALL to Data Language/I. The parameter count value equals the remaining parameters following the parameter count set off by commas.

7. and 8. These are typical CALL's used to retrieve data from a data base using a qualified search argument.

```
CALL   PLITDLI  (parameter count, function, pcbname,
                 segment, I/O area, segment search argument);
```

9. This is a typical CALL used to retrieve data from a data base using an unqualified search. This call is also a HOLD call for a subsequent delete or replace.

```
CALL PLITDLI (parameter count, function, pcbname,
segment I/O area);
```

10. This CALL is used to replace data from a Data Language/I batch program onto a data base.

11. This END statement causes the batch program to return control to the region controller. Another statement that causes the batch program to return control to the region controller is the RETURN statement. The RETURN statement may or may not immediately precede the END statement.

12. A language interface is provided for all programming languages. This module is link-edited to the batch program and provides a common interface to Data Language/I.

Assembler Language Batch Program Structure

The entry point to an Assembler Language program which utilizes Data Language/I may have any desired name. However, Register 1, upon entry to the application program, contains a variable-length full-word parameter list. Each word in this list contains a control block address which must be saved by the application program. The last word in the parameter list is signed negative. The addresses in this list are subsequently used by the application program when executing Data Language/I calls.

All Data Language/I calls from an Assembler Language program should be executed with the CALL macro-instruction. Register 1 must be constructed prior to execution of the CALL statement to

point to the variable-length full-word parameter list. This may be done through operands of the CALL macro-instruction. The parameters in this list are addresses of:

- Input/output functions
- Control block addresses associated with data bases
- Input/output work area
- Zero or more segment identifiers

The entry point for the CALL macro-instruction is CBLTDLI.

Application programs used in the batch Data Language/I environment may use both Data Language/I for data base processing and standard Operating System/360 data management for nondata base input/output operation.

THE LANGUAGE INTERFACE

The language interface module provides the standard interface mechanism which allows a batch processing program to communicate with Data Language/I data base calls. A copy of this module must be link-edited with each batch processing program. When the module is entered, the structure and addresses of the Data Language/I call are verified. If an invalid call structure is received, the batch processing program is terminated.

The language interface is designed to handle all supported languages which interface with Data Language/I. Upon entry into the language interface, a pointer to a parameter list is provided by the call structure.

Two types of parameter lists may be constructed: implicit lists and explicit lists. The COBOL program may use either type of list and the language interface modifies the list as required to pass an implicit list to Data Language/I. The list is restored to its original format before being returned to the application program. PL/I, on the other hand, allows only explicit parameter lists.

The following calls permit the standard entry points to the correct language interfaces and should be used for all data calls.

PL/I - CALL PLITDLI.....

COBOL - CALL 'CBLTDLI'.....

Assembler - CALL CBLTDLI,.....

Parameter List Contents

The generated format of the parameter lists may be of interest to the Application Programmer (see Figure 25). The actual construction of these lists is accomplished by the CALL statement parameters in the high level languages. The contents of these lists, as seen by Data Language/I, are shown for information purposes.

The format for CALL parameter lists is standard and should be as follows:

IMPLICIT PARAMETER LIST CONTENTS

Bytes	
+0	Function address
+4	PCB address or PSB name address
+8	Segment input/output area address
+12	First Segment Search Argument address
+16	Next Segment Search Argument address
+20	Last Segment Search Argument address

The high order byte of the word containing the last parameter in an implicit parameter list contains an X'80'

EXPLICIT PARAMETER LIST CONTENTS

Bytes	
+0	Parameter count
+4	Function address
+8	PCB address or PSB name address
+12	Segment input/output area address
+14	First Segment Search Argument address
+20	Next Segment Search Argument address
+24	Last Segment Search Argument address

Figure 25. Parameter list contents.

Parameter count is a binary halfword count of the number of other parameters that exist in the parameter list.

In PL/I, the function, PCB, segment I/O area, and segment search argument addresses are addresses of the dope vectors for the parameters.

The function, segment I/O area, and segment search arguments should be defined as character strings when PL/I is used. Each segment search argument may be described as a two level structure. All PCB's can only exist as two level structures. Level one is the PCB address. Level two is the fields within the PCB.

The application language interface disables any interrupt traps set by the application program with an SPIE macro instruction.

SEGMENT SEARCH ARGUMENTS (SSA)

When an application programmer requests Data Language/I to perform data base functions, it is frequently necessary for him to specifically identify a particular segment by its key field and the key fields of all parent segments along the hierarchical path leading to that segment. These key field values do not appear directly in the CALL statement parameters provided to Data Language/I. Instead, a segment search argument name is given which points to an area in the user's program which contains the actual segment search argument (SSA).

Segment search arguments may be used with GET calls and are required for all INSERT calls.

The SSA consists of two pieces, the segment name and (as required) a segment qualification statement. The segment name points Data Language/I to the entry in the Data Base Description which contains and defines the characteristics of the segment and its key field.

The qualification statement contains information which Data Language/I uses to test the value of the segment key or data field with the data base to determine if the segment meets the user's specifications. Using this approach, Data Language/I does the data base segment searching and the program need process only those segments in which it is interested.

A segment qualification statement is composed of several elements. Except where they are used to fill out a field, there must be no blanks in this statement. The complete qualification for each segment is contained between the left and right parentheses.

The segment search argument (SSA) structure is:

segment-name(segment-field-name-RO-comparative-value)

Segment Name

The segment name is 8 bytes long.

segment-name

is the segment name that pertains to a specific segment in the hierarchical structure of a data base record and is established by the Data Base Description.

Segment Qualification Statement

The segment qualification statement is of variable length. If a segment search argument has no qualification statement, the eight-byte segment name must be followed by a character other than (.

Begin-qualification-operator

is the left parenthesis, (. It indicates the beginning of a qualification statement.

Segment-field-name

is the name of a segment search field which appears in the description of that segment-type in the Data Base Description. The name is eight characters long, with right-justified embedded blanks as required. If the I/O function is GET, the named field may be either the key field or a data field within a segment. It must be the key field if the segment search argument applies to a root segment. Only the last SSA may be qualified on a data field. The last SSA in the INSERT call may not have a qualification statement.

RO = Relational-Operator

is a set of two characters which express the manner in which the contents of the field, referred to by the segment-field-name, are to be tested against the comparative-value. The sequence of checking is less than, equal to, then greater than.

Operator	Meaning
b =	must be equal to
b >	must be greater than
b <	must be less than
¬ =	must be not equal to
= >	must be equal to or greater than
= <	must be equal to or less than

Note: As used above, the lower case b represents a blank character.

If the qualification statement applies to a root segment, only the =, =>, or >, relational operators may be used.

Comparative-value

is the value against which the contents of the field, referred to by the segment-field-name, is to be tested. The length of this entry must be equal to the length of the named field in the segment of the data base, that is, it includes leading or trailing blanks (for alphameric) or zeros (usually needed for numeric fields) as required.

End-qualification-operator

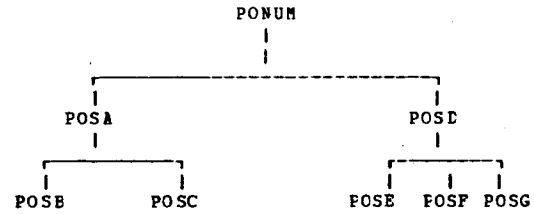
is the right parenthesis,). It indicates the end of a qualification statement.

The qualification statement test is terminated as soon as an occurrence within the data base of a segment-type is found that satisfies that qualification test. This procedure continues for all SSA's in a Data Language/I data base call until the desired segment is found.

The following are examples of segment search arguments with and without a qualification statement.

Examples of SSA Usage

The data base structure and the segment names are as follows:



The segment search argument for the various degrees of qualification might then be as follows:

1. SSA with no qualification

PONUMbbbb

A call using an unqualified SSA could access the next root segment called PONUM. Note that the ninth position must not contain a left parenthesis.

2. SSA with qualification

PONUMbbb(ACTUALPOb=AB60733)

A call using this simple qualification would access the root segment of a data base record whose root segment key field, called ACTUALPO, has a value of AB60733.

3. SSA's that form a complex qualification

a. PONUMbbb(ACTUALPOb=AB60733) POSDbbbb(AFLDbbbbb=4234) POSFbbbb(KFLDbbbbb=24357)

This type of qualification accesses the POSF segment whose key field, KFLD, value is 24357; whose parent segment's key field, AFLD field value equals 4234; and whose root segment's key field, ACTUALPO, equals AB60733.

b. PONUMbbb(ACTUALPOb=AB60733) POSDbbbb(AFLDbbbbb=4234)

This type of qualification obtains the POSD segment whose AFLD field equals 4234, and whose root segment's key field, ACTUALPO, equals AB60733.

SEGMENT INPUT/OUTPUT AREAS

The segment input/output (I/O) area is an area in the application program into which Data Language/I puts a requested segment, or from which Data Language/I takes a designated segment. If a common area is used, it must be as long as the longest segment to be processed. The segment I/O area name points to the leftmost byte of the area. Segment data is always left-justified within a common segment I/O area.

PROGRAM COMMUNICATION BLOCK (PCB) FORMAT

A Program Communication Block (PCB) exists external to an application program for each data base used by the program. The linkage section of the data division of a COBOL program defines an external data field for each PCB. The EXTERNAL data attribute in PL/I performs the same function.

The PCB is a set of contiguous fields which provide the application program with the ability to make Data Language/I calls supplying the following information:

- The name of the data base to be processed
- The specification of the Data Language/I functions which will be used
- Indications of the types of segments to be processed
- Areas for receiving status responses from Data Language/I

No initial values are defined in a PCB in the linkage section. The values for a PCB exist in the Program Specification Block (PSB) and are fixed at PSBGEN time.

PCB for a Data Base

The PCB provides specific areas used by Data Language/I to advise the application program of the results of its calls. At execution time, all PCB entries are Data Language/I controlled, where control means the exclusive authority to change the contents of a PCB entry. The programmer exercises his options as to what goes into the PCB at PSB generation time.

The following fields comprise a PCB for a data base:

1. Name of the PCB - refers to the entire structure of PCB entries and is used in program statements.

2. Name of Data Base Description - the DBD name from the library of Data Base Descriptions. This field contains character data and is eight bytes long.

3. Segment Hierarchy Level Indicator - Data Language/I loads this area with the level number of the lowest segment encountered in its attempt to satisfy a program request. When a retrieve is successfully completed, the level number of the retrieved segment is placed here. If the retrieve is unsuccessful, the level number returned is that of the last segment, along the path to the desired segment, which satisfied the segment search argument. This field contains character data and is two bytes long and is a right-justified numeric.

4. Data Language/I Status Code - A status code that indicates the results of a Data Language/I call is placed in this field and remains here until another Data Language/I call uses this PCB. (Specific status codes are discussed with their associated calls in a later section of this manual.) This field contains two bytes of character data. When a successful CALL is executed, this field is returned blank or with a warning status indication.

5. Data Language/I Processing Options - This area contains a character code which tells Data Language/I the kinds of calls that will be used by the program for data base processing. This field is four bytes long. Only one of the following processing options may be specified in a particular PCB. It is left justified to the first byte of the four byte field. The remaining three bytes are reserved.

Possible values for the processing options are:

G - for get function

A - for get, delete, insert, and replace functions

L - for loading a hierarchical indexed sequential or hierarchical sequential data base

If delete, replace, or insert option is specified for a hierarchical indexed sequential data base, A must be used. The only valid options for a hierarchical sequential data base are G and L, and they are mutually exclusive in the same PCB. The L option is mutually exclusive with other options in the same PCB.

6. Reserved Area for Data Language/I - Data Language/I uses this area for its own internal linkage related to an application program. This field is one binary word.

7. Segment Name Feedback Area - Data Language/I fills this area with the name of the lowest segment encountered in its attempt to satisfy a call. When a retrieve is successful, the name of the retrieved segment is placed here. If a retrieve is unsuccessful, the name returned is that of the last segment, along the path to the desired segment, which satisfied the segment search argument. This field contains eight bytes of character data.

8. Length of Key Feedback Area - This entry specifies the length of the area required to contain the completely qualified key of any sensitive segment. This field is one binary word. The completely qualified key of a third-level segment includes the first- and second-level keys.

9. Number of Sensitive Segment Types - This entry specifies the number of segment-types in the data base to which the application program is sensitive. This field is one binary word.

10. Key Feedback Area - Data Language/I places in this area the completely qualified key of the lowest segment encountered in its attempt to satisfy a call. When a retrieve is successful, the key of the requested segment, and the key field of each segment along the path to the requested segment, are concatenated and placed in this area. The key fields are positioned from left to right, beginning with the root segment key and following the hierarchical path. When a retrieve is unsuccessful, the keys of all segments along the path to the requested segment for which searching was successful are placed in this area.

11. Names of Sensitive Segments - This area contains the name of sensitive segments, the number of which is specified in field 9 above. Each entry is eight bytes of character data and may include embedded right-justified blanks. The names of the sensitive segment-types must be listed in the same order as they appear in the Data Base Description.

COBOL: The following is an example (Figure 26) of a PCB for a data base in a COBOL program. The reference numbers relate to the preceding description of entries.


```
Reference
  1      01  SAMPLE PCB.
  2          02 DBD-NAME          PICTURE X(8).
  3          02 SEG-LEVEL         PICTURE XX.
  4          02 STATUS CODE       PICTURE XX.
  5          02 PROC-OPTIONS      PICTURE XXXX.
  6          02 RESERVE-DLI       PICTURE S9(5) COMPUTATIONAL.
  7          02 SEG-NAME-FB       PICTURE X(8).
  8          02 LENGTH-FB-KEY     PICTURE S9(5) COMPUTATIONAL.
  9          02 NUMB-SENS-SEGS    PICTURE S9(5) COMPUTATIONAL.
 10          02 KEY-FB-AREA       PICTURE X(19).
 11          02 ROOT-SEG-NAME     PICTURE X(8).
 11          02 SECOND-SEG-NAME   PICTURE X(8).
 11          02 THIRD-SEG-NAME    PICTURE X(8).
```

Figure 26. COBOL example

PL/I: The following is an example (Figure 27) of a PCB for a data base in a PL/I program. The reference numbers relate to the preceding description of entries.

```
Reference
Number

  1   DECLARE 1   SAMPLE PCB,
  2           2   DBD_NAME         CHARACTER (8),
  3           2   SEG_LEVEL        CHARACTER (2),
  4           2   STATUS_CODE      CHARACTER (2),
  5           2   PROC_OPTIONS     CHARACTER (4),
  6           2   RESERVE_DLI      FIXED BINARY (15,0),
  7           2   SEG_NAME_FB      CHARACTER (8),
  8           2   LENGTH_FB_KEY    FIXED BINARY (15,0),
  9           2   NUMB_SENS_SEGS   FIXED BINARY (15,0),
 10           2   KEY_FB_AREA    CHARACTER (19),
 11           2   ROOT_SEG_NAME    CHARACTER (8),
 11           2   SECOND_SEG_NAME  CHARACTER (8),
 11           2   THIRD_SEG_NAME   CHARACTER (8);
```

Figure 27. PL/I example

It should be noted that no initial values have been defined in the PCB. The values for the entries in the PCB exist in the Program Specification Block (PSB).

ENTRY TO APPLICATION PROGRAMS

For purposes of standardization and clarity, a standard entry point is used for programs to be run under Data Language/I. The first statement in the PROCEDURE DIVISION of a COBOL or the first statement after the DECLARE statements of a PL/I program should be as follows:

COBOL

```
ENTER LINKAGE.

ENTRY 'DLITCBL' USING pcbname-1,.....pcbname-n.

ENTER COBOL.
```

PL/I

```
DLITPLI: PROCEDURE (pcbname-1,.....pcbname-n) OPTIONS(MAIN);
```

The 'pcbname' parameters in these statements establish a correlation between the problem program and the PCB's with which the program deals. Each 'pcbname' must appear at the first level in either the linkage section (COBOL) or an external DECLARE statement (PL/I); 'pcbname-1' through 'pcbname-n' correspond positionally to the PCB's specified during PSB generation.

DATA LANGUAGE/I DATA BASE CALLS

The data services of Data Language/I are available to the application program through the use of standard language calls. The following calls are used in conjunction with the function codes shown below.

For COBOL - CALL 'CBLTDLI' USING function-code, pcbname, segment I/O area, ssa.....

For PL/I - CALL PLITDLI (parm-count, function-code, pcbname, segment I/O-area,ssa.....);

Valid batch processing program Data Language/I call functions are:

Meaning	Function Code
GET UNIQUE	'GUbb'
GET NEXT	'GNbb'
GET NEXT WITHIN PARENT	'GNPb'
GET HOLD UNIQUE	'GHUb'
GET HOLD NEXT	'GHNb'
GET HOLD NEXT WITHIN PARENT	'GHNP'
INSERT	'ISRT'
DELETE	'DLET'
REPLACE	'REPL'

The GET UNIQUE Call (GUbb) - Data Base

The GET UNIQUE call is used to retrieve a unique statement occurrence from the data base described in the PCB. The GET UNIQUE call can be used for random processing or it can be used to establish the position in the data base where sequential processing is to begin.

SSA's in GET UNIQUE calls must conform to the following rules:

1. The call must have SSA's.
2. The first SSA must be for the root segment, and any following SSA's must proceed down a hierarchical path with no missing intermediate levels.
3. The search field must be the key field in the qualification statement of the root SSA, and the operator must be =, >, or = >.

4. The search field may be any defined field (key field if not last SSA) in the qualification statement of dependent segment SSA's, and the operator may be =, ¬=, <, >, =>, or <=. A field is defined if it is described by an FLDK or FLD card at DBD generation time for the data base. All comparisons on key or data fields are logical bit-for-bit compares.

Status Codes for GET UNIQUE Calls

At the completion of a GET call, a status code indicating the results of the call made is available, in the PCB status code field, to the programmer. The status code should always be interrogated upon completion of a call.

If the GET call was completed as requested, the two-byte status code is blank; otherwise, the status code is one of the following:

STATUS CODE	MEANING
'AB'	No segment I/O area in call
'AC'	Hierarchical error in SSA's
'AD'	Illegal function parameter
'AG'	On GU/GHU calls the first SSA must be for level one segment
'AH'	GU/GHU must have SSA's
'AI'	Data management open error
'AJ'	Invalid SSA qualification format
'AL'	Invalid PCB used in batch program execution (wrong PCB address)
'AM'	Invalid field in GET call SSA qualification statement
'GC'	Qualification statement illegal operation code (less than, equal to, greater than)
'GD'	Qualification statement field symbolic undefined

'GE'	Segment not found
'GH'	I/O error QISAM
'GI'	I/O error BSAM/OSAM

The GET NEXT Call (GNbb) - Data Base

The GET NEXT call is used to retrieve the next desired segment from the data base as described by the DBD name and sensitive segments in the PCB.

SSA's in GET NEXT calls must conform to the following rules:

1. The call may or may not have SSA's.
2. SSA's may or may not have qualification statements.
3. The first SSA may be for any level segment, but any following SSA's must proceed down a hierarchical path with no missing intermediate levels.
4. The search field must be the key field in the qualification statement of a root SSA, and the operator must be, =, >, or =>.
5. The search field may be any defined field (key field if not last SSA) in the qualification statement of dependent SSA's, and the operator may be =, ¬=, <, =<, =>, or > . All comparisons on key or data fields are key or data compares.

The execution of a GET NEXT call without SSA's returns the next segment occurrence within the data base relative to the positioning of the data base during the previous GU, GN, or GNP call. An uninterrupted series of these call statements could be used to retrieve each segment occurrence from the data base, beginning with the first, and proceeding sequentially through the last for all sensitive segments. The parameters for this form of a GET NEXT are the function, PCB name, and segment I/O work area.

The GET NEXT call only progresses forward from the position in the data base established in the preceding call, in an attempt to satisfy the current call requirements.

Status Codes for GET NEXT Calls - Data Base

At the completion of a GET call, a status code, indicating the results of the call made, is available in the PCB status code field to the programmer. The status code should always be interrogated upon completion of a call.

If the GET call was completed as requested, the two-byte status code is blank or GA or GK; otherwise, the status code is one of the following:

STATUS CODE	MEANING
'AB'	No segment I/O area in call
'AC'	Hierarchical error in SSA's
'AD'	Illegal function parameter
'AI'	Data management open error
'AJ'	Invalid SSA qualification format
'AL'	Invalid PCB used in batch program execution (wrong PCB address)
'AM'	Invalid field in GET call SSA qualification statement
'AN'	GET NEXT call not allowed immediately after ISRT call
'GA'	Crossed hierarchical boundary into HIGHER level (That is, closer to Root). Note: On GN and GHN without SSA's, this code will be returned every time this happens, including the retrieve of a new Root Segment. (See paragraph on Cross Hierarchical Boundary definition.)
'GB'	End of data set (you have reached the last segment)
'GC'	Qualification statement illegal operation code (less than, equal to, greater than)
'GD'	Qualification statement field symbolic undefined
'GE'	Segment not found
'GH'	I/O error QISAM

'GI'	I/O error BSAM/CSAM
'GK'	For unqualified (no SSA's) Get-Next calls: the segment which has now been placed in your segment-area is the same level as the segment handled in the immediately prior call, but the segment-name is different.

Definition of Cross Hierarchical Boundary

The GA status code is a warning indication. When a GN or GNP call without SSA's is issued, Data Language/I may return this status code to indicate the crossing of hierarchical boundaries. This status code indicates that Data Language/I has passed from one segment in the data base at level X to another segment in the data base at level Y where Y is less than X. In other words, it has proceeded upward in the hierarchy toward the root segment. This code is not returned to the using application program when a GU, GN with SSA's, or GNP with SSA's is issued, because the user is explicitly asking, through the presence of the SSA's, to traverse a known path in the data base. Thus the GA status code is a warning (to the user of the GN or GNP call to move sequentially through a portion of the data bases) that Data Language/I has taken him implicitly from a segment at one level of the hierarchy to a segment at another, higher, level of the hierarchy.

The GET NEXT WITHIN PARENT Call (GNP) - Data Base

The GNP call obtains lower-level segment occurrences within the family of a parent segment. It may be used to retrieve all segments or specific segments within the family of the given parent segment.

At the issuance of the first GNP call, the relevant parent is established by looking back to the last successfully completed GET UNIQUE or GET NEXT call. The parentage established with a GU or GN call remains constant for successive GNP calls. However, the parentage will be destroyed whenever a GU or GN call is executed. The parent segment may be at any level in the hierarchical structure.

SSA's in GET NEXT WITHIN PARENT calls must conform to the following rules:

1. The call may or may not have SSA's.
2. SSA's may or may not have qualification statements.
3. The first SSA must be for a child segment one level below the parent segment established by the GU or GN call; and

any following SSA's must proceed down a hierarchical path with no missing intermediate levels.

4. An SSA for the root segment is not allowed (it has no parent).

5. The search field may be any defined field (key field only if not last SSA) in the qualification statement of dependent segment SSA's, and the operator may be =, ¬ =, <, >, =<, or = >. All comparisons are logical bit-by-bit compares.

If a GNP call without SSA's is reissued, this call will read all segment occurrences under the relevant parent segment, going up and down hierarchical levels and crossing boundaries in the structure beneath the parent for all sensitive segments. A not-found condition results when Data Language/I encounters the next segment occurrence that is at the same level as the parent or higher.

Status Codes for GET NEXT WITHIN PARENT Calls

At the completion of a GET call, a status code, indicating the results of the call made, is available in the PCB status code field to the programmer. The status code should always be interrogated upon completion of a call.

If the GET call was completed as requested, the two-byte status code is blank or GA or GK; otherwise, the status code is one of the following:

STATUS CODE	MEANING
'AB'	No segment I/O area in call
'AC'	Hierarchical error in SSA's
'AD'	Illegal function parameter
'AE'	On GNP/GHNP calls, first level SSA illegal
'AI'	Data management open error
'AJ'	Invalid SSA qualification format
'AL'	Invalid PCB used in batch program execution (wrong PCB address)

'AM'	Invalid field in GET call SSA qualification statement
'AN'	GET NEXT call not allowed immediately after ISRT call
'GA'	Crossed hierarchical boundary into HIGHER level (that is, closer to root). Note: On GNP and GHNP, this code will be returned every time this happens. This code returned on unqualified (no SSA's) GNP calls only. (See paragraph on Cross Hierarchical Boundary definition.)
'GB'	End of data set (you have reached the last segment)
'GC'	Qualification statement illegal operation code (less than, equal to, greater than)
'GD'	Qualification statement field symbolic undefined
'GE'	Segment not found
'GH'	I/O error QISAM
'GI'	I/O error BSAM/OSAM
'GP'	Gaps in SSA levels for GNP/GHNP calls
'GK'	For unqualified (no SSA's) Get-Next calls: the segment which has now been placed in your segment-area is the same level as the segment handled in the immediately prior call, but the segment-name is different. This code returned on unqualified GNP only.
'GP'	Gaps in SSA levels for GNP/GHNP calls

The GET HOLD Calls - Data Base

To change the contents of a segment in a data base, through a DLET or REPL call, the program must first obtain the segment. It then changes its contents and requests Data Language/I to place the segment back into the data base.

When a segment is to be changed, this must be indicated to Data Language/I at the time the segment is obtained. This

indication is given by using the GET HOLD calls. These function codes are like the standard GET function, except the letter H immediately follows the letter G in the code; that is, the hold form of the standard GET NEXT WITHIN PARENT (GNPt) is GHNP. There are three GET HOLD calls: GHUt, GHNt, and GHNP. They function like the standard GET calls. They also indicate to Data Language/I that the segment may be changed or deleted. (See the section on the use of the GET calls and status codes.) The HOLD forms of GET permit Data Language/I to make certain that the segment to be placed back into the data base is the same segment Data Language/I returned on completion of the last GET HOLD call.

After Data Language/I has returned the requested segment to the user, one or more fields in the segment, but not the key field, may be changed.

The user should also guard against changing data from one type to another type; for example, binary data should not be replaced with decimal data.

After the user has changed the segment contents, he is ready to call Data Language/I to return the segment to the data base. If, after issuing a GET HOLD call, the program determines that it is not necessary to change the retrieved segment, the program may proceed with other processing.

Status Codes for GET HOLD Calls

At the completion of a GET call, a status code, indicating the results of the call made, is available in the PCB status code field to the programmer. The status code should always be interrogated upon completion of a call.

The actual status codes for the GET HOLD calls are the same as for the non-HOLD type of call. That is, for GHU (Get Hold Unique), see the GU status codes; for GHN (Get Hold Next), see the GN status codes; and for GHNP (Get Hold Next Within Parent), see the GNP status codes.

The INSERT Call (ISRT) - Data Base

The Data Language/I INSERT call is used for two distinct purposes: It is used to initially load the segments for creation of a data base. It is also used in the Hierarchical Indexed Sequential organization to insert new occurrences of an existing segment-type into an established data base. The processing options field in the PCB associated with the data

base dictates Data Language/I execution of the call. The format of the INSERT call is identical for either use.

The INSERT call may be used with other Data Language/I segment processing calls in a processing program. In this environment, the INSERT call is used to place new occurrences of existing segment-types into an established Hierarchical Indexed Sequential data base.

When a segment is inserted into the data base, the user must tell Data Language/I precisely where the segment is to be logically placed. This placement is given to Data Language/I by referring to one or more SSA's in the call. Through the SSA's, the user tells Data Language/I the segment name and qualification statement for each segment along the path. However, the SSA for the segment to be inserted must contain only the segment name. The name may not be followed by the character (. The path begins with the root segment and proceeds to each segment, down the hierarchical path, upon which the inserted segment depends for its full meaning.

a. The COBOL call format for inserting segments is:

   CALL 'CBLTDLI' USING function, PCB-name, segment-I/O-area, ssa-1,...,ssa-n.

b. The PL/I call format for inserting segments is:

   CALL PLITDLI (parm-count, function, PCB-name, segment-I/O-area,ssa-1,.....,ssa-n)

When inserting to a Hierarchical Sequential data base, INSERT means to load an output data base. The PCB processing option L is used. The option A is invalid for the Hierarchical Sequential organization. Inserts to an established Hierarchical Sequential data base cannot be made without reprocessing the whole file or by adding to the end, and must be in sequence.

The user must follow each INSERT call in his program with statements which examine the returned status codes in the PCB, to determine if the requested action was completed properly.

When inserting a segment into a data base, it is not necessary to cause the prior positioning and holding of a segment using the GET HOLD calls. The INSERT call itself contains all the qualification necessary to cause automatic positioning.

Status Codes for INSERT Calls

If the segment is inserted properly, the INSERT module places a blank status code in the PCB; otherwise, one of the following status codes will be returned to the problem programmer.

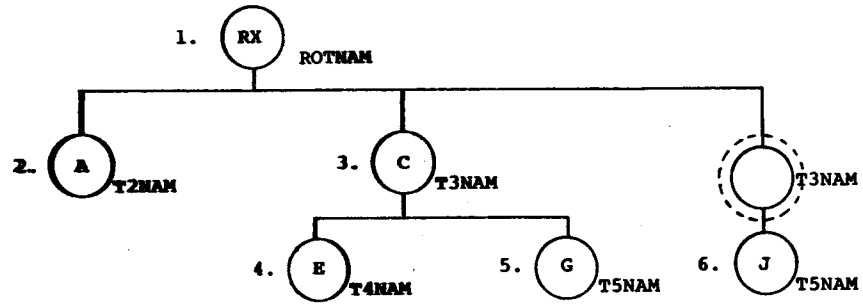
STATUS CODE	MEANING
'AB'	No segment I/O area in call
'AD'	Illegal function parameter
'AG'	On Insert calls, the first SSA must be for level one segment
'AH'	Insert call must have SSAs
'AI'	Data mangement open error
'AK'	Invalid key field in INSERT SSA qualification statement
'AL'	Invalid PCB used in batch program execution (wrong PCB address)
'GE'	When attempting to insert a dependent segment with PROCOPT =A (get, delete, insert, and replace), a parent segment was not found
'IE'	**ISAM error
'IF'	**OSAM I/O error
'II'	**Duplicate segment already in database

Note: Any status code which starts with the letter I will only occur with PCB processing option equal to (I) load.

'LA'	On insert calls with PCB processing option equal to L, the first SSA must be for level one segment
'LB'	Duplicate segment already in database
'LC'	Segment key field contains out of sequence
'LD'	A parent for this segment has not been submitted for loading within this logical record (at least, not within the proper recent stream of submissions). (See example below)
'LE'	Sibling segments (same level, same parent, different segment names) have been presented for loading in the wrong order; they must be loaded in the order in which they are shown on the DBD table feedback. (See example below)
'LF'	ISAM I/O error
'LG'	I/O OSAM error
'LH'	Segment presented for loading is not in hierarchical order. This code means the same as LD and/or LE, but is returned in lieu of either when certain kinds of hierarchical paths are under construction. (See example below)

** These occur only with PCB processing options equal to A.

EXAMPLE OF LD:

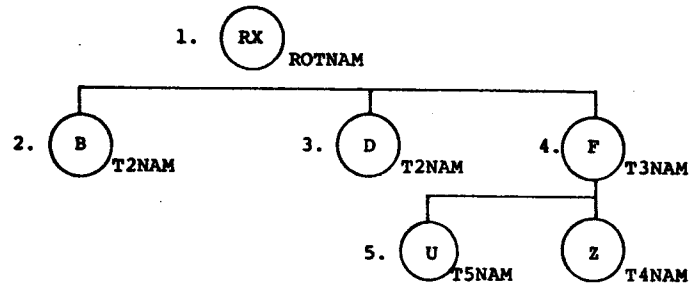


1. User loads segments RX, A, C, E, and G.
2. Next segment presented is a level three (J).
3. Since G was the last segment loaded, Data Language/I's level table contains entries for RX, C, and G.
4. When segment J is presented, Data Language/I checks the parent level table entry (currently containing data from the previously loaded segment, G).

5. The nonzero level table entry corresponding to segment C causes the "SAMEDS" switch to examine parent data.

6. If the parent name of the new segment is not the same as the name currently held in the level table, status code 'LD' is returned to the user.

EXAMPLE OF LE:



1. User loads segments RX, B, D, F, and U.

2. Next segment presented is a level three (Z) with segment name T4NAM.

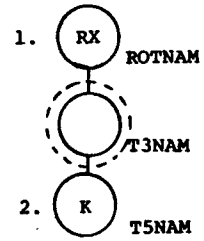
3. Data Language/I first checks the level table entry immediately preceding the level of segment Z.

4. This parent entry is nonzero, and a level three segment is just loaded. Data Language/I therefore proceeds to check the hierarchy relationship between sibling segment U and the new segment, Z.

5. From the DBD hierarchy definition, Data Language/I finds that the previously loaded sibling segment U is defined after Z. The segments must be loaded in the same sequence as they are defined in the DBD.

6. This sequencing error of sibling segments thus generates status code 'LE'.

EXAMPLE OF LH:



1. User loads root segment RX.

2. Next segment presented is a level three (K).

3. Since a new root was the last segment loaded, Data Language/I's level table contains all zero entries, except for the first level.

4. When segment K is presented, Data Language/I looks at the level table entry immediately preceding the level of segment K.

5. Since this entry contains a zero, a status code of 'LH' is returned to the user.

The DELETE Call (DLET) - Data Base

To delete the occurrence of a segment from the data base, the program must first get the segment and then ask Data Language/I to delete it. When a segment has been deleted, it is no longer available to any program. Before the program can ask Data Language/I to delete a segment, however, the segment must first be obtained by issuing a GET HOLD call through Data Language/I. Once the segment has been acquired, the DELETE call may be issued.

There must be no Data Language/I calls which use the same PCB intervening between the GET HOLD call and the DELETE call or the DELETE call is rejected. Quite often a program may want to process a segment prior to deleting it. This is permitted as long as the processing does not involve a Data Language/I call which refers to the same PCB used to get the segment.

Data Language/I is advised that a segment is to be deleted when the user issues a CALL that has the function DLET. When the DELETE call is executed, the specified segment occurrence is not physically deleted, but simply flagged as being deleted. The occurrence is physically deleted when the file is reorganized. The deletion of a parent, in effect, deletes all the segment occurrences beneath that parent. If the segment being deleted is a root segment, all dependent segments under that root in relevant data set groups are also flagged as deleted.

The segment to be deleted must occupy the area referred to by the segment-I/O-area in the DELETE call.

Note that the SSA has no meaning to the DELETE call, since positioning is accomplished by the previous GET HOLD call. SSA's should not be included in a DELETE call.

a. The delete call format for a COBOL program is:

   CALL 'CBLTDLI' USING function, PCB-name, segment-I/O-area.

b. The delete call format for a PL/I program is:

   CALL PLITDLI (parm-count, function, PCB-name, segment-I/O-area);

For a program which processes Hierarchical Sequential data bases where each record is rewritten on a new data base, the DLET call has no meaning and will be rejected as an invalid function. If a segment occurrence is to be deleted, it is simply not written to the output data base.

The REPLACE Call (REPL) - Data Base

The purpose of the REPLACE call is to allow a segment that has been retrieved through a GET HOLD call and modified through program processing to be replaced in the data base. The segment to be modified and replaced must first be obtained by a GET HOLD call. No intervening calls involving the associated PCB may be made between the GET HOLD and the REPLACE call. If this rule is violated, the REPLACE call is rejected.

In the modification of a segment to be replaced in the data base, care must be taken not to modify the segment key field. If modification of the key field is attempted, the REPLACE call is rejected.

The segment to be replaced must occupy the area referred to by segment-I/O-area in the REPLACE call. The segment in the Data Language/I buffer area is overlayed with the segment I/O area in the REPLACE call.

a. The replace call format for a COBOL program is:

   CALL 'CBLTDLI' USING function, PCB-name, segment-I/O-area.

b. The replace call format for a PL/I program is:

   CALL PLITDLI (parm-count, function, PCB-name, segment-I/O-area);

For a program which processes Hierarchical Sequential data bases where each record is rewritten on a new data base, the REPLACE call has no meaning. If a segment occurrence is to be replaced, it is simply placed in the output data base with an INSERT call.

Status Codes for DELETE/REPLACE calls

The following error codes may be generated as a result of either a DELETE or a REPLACE call, and are placed into the PCB status code field.

STATUS CODE	MEANING
'AB'	No segment I/O area address in call
'AD'	Illegal function parameter
'AF'	No SSA's allowed for DLET/REPL calls
'AL'	Invalid PCB used in batch program execution (wrong PCB address)
'DA'	User has tried to change segment key field on a DELETE or REPLACE
'DJ'	Before a DELETE/REPLACE, no preceding successful GET HOLD

TERMINATION OF AN APPLICATION PROGRAM

At the completion of processing of any application program (batch), control must be returned to the region controller. The RETURN statement must be given in every program as follows:

COBOL

```
ENTER LINKAGE.

RETURN.

ENTER COBOL.
```

PL/I

```
RETURN;
```

ASSEMBLER

```
RETURN  (14,12),RC=0
```

The return statement in a batch-type program also returns control to the region controller. The region controller subsequently returns control to the Operating System/360 job terminator after Data Language/I resources are released.

The return statement from an application program written in Assembler language should contain Register 15 equal to zero if the program is terminating normally.

PROCESSING REGION ABENDS

Comp Code	Issuing Component	Explanation
0030	DFSIRA00	Program (PSB) name was omitted from the parm field on the EXEC statement.
0033	DFSIRA00	Parm field of EXEC card is invalid format. Comma does not follow first positional parameter.
0034	DFSIRA00	Parm field of EXEC card specifies an invalid region-type code.
0035	DFSIRA00	Parm field of EXEC card contains an excessive number of positional parameters.
0036	DFSIRA00	First character of a positional parameter in parm field of EXEC card is blank or invalid.
0037	DFSIRA00	A positional parameter in parm field of EXEC card exceeds maximum allowable length.
0038	DFSIRA00	A required positional parameter is omitted from the parm field of the EXEC card.
0048	DFSIRA00	Parm field was omitted from EXEC statement.
0150	DFSIDBA0	PCB address passed in the USING list of a batch program is not the same as any passed to the program by Data Language/I at first entry. The PCB referred to in the CALL statement may not have been defined at PSBGEN time. The USING list of the CALL statement may be improperly constructed.
0151	DFSIDBA0	USING list of CALL statements in a batch program is truncated at the function position. The USING list contains only one named data item.
0200	DFSIDLK0	The available dynamic main storage in the Operating System/360 region or

		partition in which a program is operating is not sufficient to allow the Data Language/I block loader to fetch the required PSB's and DBD's. Increase region size.
0201	DFSIDLK0	PSB loaded in the application program processing region has invalid or inconsistent processing options specified. Check PSB generation.
0204	DFSIDLK0	Error in hierarchical definition of DBD. Register 8 points to segment name at which error was discovered. Check DBD generation.
0206	DFSIDLK0	Unable to open PSB and DBD libraries. Check proper allocation for DD name.
0208	DFSIDLK0	A sensitive segment is named in PSBGEN for which no corresponding segment name was defined in the associated DBDGEN. Register 3 at abend points to the unmatched sensitive segment name. Register 9 points to the DBD name. Register 8 points to the data base PCB name in the PSB. Check PSB and DBDGEN.
0260	DFSIPR00	Number of parameters (data items named in USING list) in the application program CALL exceeds the allowable limit.
0261	DFSIPR00	One of the values passed in the USING list of the application program Data Language/I CALL is invalid. It either exceeds object machine size, does not meet alignment requirements, or violates storage protection boundaries.
0710	DFSIOS60	During OPEN of a Data Language/I overflow data set, the calculated block length exceeded the maximum track length for the device allocated. Check DD cards for OSAM data set allocation. Register 3 points to the Data Control Block (DCB) at abend time.

Commonly Encountered Operating System/360 System Abends

System	Abend Code	Usual Problem
806		Joblib card omitted from library containing Data Language/I modules or user's application program library.
213		DD cards for data sets representing data bases are missing or do not have proper DD name. DD names for data bases must correspond to those used in DMAN cards of DBD generation. or Data sets to be opened as DISP=OLD do not exist or cannot be opened.

DATA BASE DESCRIPTIONS (DBD)

The Data Base Description (DBD) is the Data Language/I control block used to describe in detail the structure and storage organization of every data base. All the information about a data base is available in its DBD, and it is from this pool of information that other internal Data Language/I control blocks are built at execution time.

It is not the responsibility of the Application Programmer to generate the DBD's that affect the data bases he uses. However, it is imperative that he be able to understand the contents of a DBD in order to utilize the data bases that already exist and are available to him.

Before Data Language/I can be used to process data base information, or even before the data base can be created, the data base must be described to Data Language/I. The organization of the data within the data base must be completely described so that it can properly set up the tables and control blocks that determine how the data is to be stored.

The result of the procedures described here is the creation of a Data Language/I data base description (DBD) table that is required when the data for the data base is actually loaded into the system and when it is retrieved and manipulated during execution of any application program. Each DBD is stored as a member of a load module library generically titled the DBD library.

This procedure must precede any processing that would reference in any way the data base it describes. The data base cannot exist until the data base description is completed.

In general, the control statements for DBD generation look as follows:

1	[PRINT	NOGEN]
2	DBD	NAME=,ACCESS=
3	DMAN	DD1=,DEV1=,[DD2=],[DLIOF=]
4	SEGM	NAME=,PARENT=,BYTES=,FREQ=
5	FLDK	NAME=,TYPE=,BYTES=,START=
	[FLD	NAME=,TYPE=,BYTES=,START=]
6	DBDGEN	
7	FINISH	
8	END	

NOTE: Items 3, 4, and 5 are a set; if one is used, all must be present. At least one DMAN, SEGM and FLDK card must exist within the data base.

[ ] denotes optional statement or parameter

The following are the generalized rules for the DBD generation job step:

A- Number of DMAN cards determines whether the data base is composed of a single or multiple data set groups. One DMAN card per data set group.

E- ACCESS - (INDX) - Hierarchical Indexed Sequential organization

(SEQ) - Hierarchical Sequential organization

C- Only one PRINT NOGEN card, - eliminates object listing from DBD generation

One DBD card

One DBDGEN card

One FINISH card

One END card

D- if INDX - only DD1, DLICF, omit DD2=

if SEQ - both DD1, DD2, omit DLICF

E- follow the rules in the paragraph titled DBD Control Card Requirements

An example of a hierarchical description of control cards 3, 4, and 5 is:

```
1-DMAN (DATA SET GROUP 1)

2-SEGM (ROOT SEGMENT)

        3-FLDK
        3-FLD
        3-FLD

2-SEGM (LEVEL 2)

        3-FLDK
        3-FLD

2-SEGM (LEVEL 2)

        3-FLDK

1-DMAN (DATA SET GROUP 2)

2-SEGM (LEVEL 2)

        3-FLDK
        3-FLD

2-SEGM (LEVEL 2 or 3)
```

The job step itself consists of eight types of Data Language/I control cards arranged in a specific order. Each

control card is described individually in detail in the following section.

DBD CONTROL CARD REQUIREMENTS

The description of the data base is presented to Data Language/I on eight types of control cards.

1. Each control card must be identified by a name, called a "card-type code", which comprises three to six characters: PRINT, DBD, DMAN, SEGM, FLD, DBDGEN, FINISH, or END.

2. In the generalized example shown in the following descriptions of the control cards, these conventions apply:

   a. Words written in all capital letters must appear exactly as written.

   b. Words written in lower-case letters are to be replaced by a user specified value.

   c. The control cards are free form but must begin after column 1.

   d. The symbols [ ], { }, and ,... are used as an aid in defining the instructions. THESE SYMBOLS ARE NOT CODED; they act only to indicate how an instruction may be written.

      [ ] indicates optional operands. The operand enclosed in the brackets (for example [VL]) may be coded or not, depending on whether or not the associated option is desired. If more than one item is enclosed in brackets (for example [REREAD] [LEAVE]), one or more may be coded.

      { } indicates that a choice must be made. One of the operands from the vertical stack within braces (for example {input }), must be coded, depending on {output } which of the associated services is desired.

   e. All DBDGEN control card parameters except for the print card are key word parameters and therefore may appear in any sequence on the associated control card.

      ,...indicates that more than one set of operands may be designated in the same instruction.

PRINT Control Card

	[PRINT	NOGEN]

The PRINT NOGEN card is an Operating System/360 macro generator control card used to eliminate a printout of the object listing resulting from a DBD generation. With the PRINT card present, a source statement summary is provided for each DBD define.

DBD Control Card

This must be the first Data Language/I control card in the job step after the PRINT NOGEN. This card names the data base to be described and provides Data Language/I with preliminary information concerning its organization. There can be only one DBD control card in the control card deck. The parameters must be contained on one card.

	DBD	NAME=name ACCESS=orgamization

where:

DBD=

identifies this control card as the DBD control card. This is the card-type code.

NAME=name

name of the DBD for the data base being described. This name may be from one to eight alphameric characters, must be left justified, and must not have trailing blanks since it is not the last parameter. Normally, this name would be the same as that specified in the DD1 parameter of the DMAN control card, although this is not required.

ACCESS=organization

specifies the Data Language/I access method to be used in conjunction with this set and must be one of the following values:

ISAM or INDX

means Hierarchical Indexed Sequential organization

SAM or SEQ

means Hierachical Sequential organization

DMAN Control Card

A DMAN control card must immediately follow the DBD card. Each DMAN control card describes one data set group that is to be set up by Data Language/I as part of the data base being described. There are one primary data set group and zero to nine dependent data set groups in a single data base for the Hierarchical Indexed Sequential organization. There is only one data set group for a Hierarchical Sequential data base.

Since the DMAN card parameters may appear on more than one card, provision has been made to accommodate the overflow of parameters to more cards. When this occurs:

a. Enter a nonblank character in column 72 of each continued card.

b. A particular parameter or operand may not span two cards. If there is not space for the entire parameter on the current card, place the whole

parameter on the next card. When continuation cards are required, a comma must follow each parameter except the last on the last continuation card.

c. Continue statement in column 16 of next card.

d. The continued condition may occur in DMAN, SEGM, and FLD cards.

The number and arrangement of DMAN control cards allowed in a job step are greatly dependent upon the data base organization specified. For instance, for a SINGLE DATA SET GROUP - DATA BASE, only one DMAN control card and its associated overflow cards (if any) are allowed; for a MULTIPLE DATA SET GROUP - DATA BASE, up to ten DMAN cards are allowable. For Hierarchical Sequential data bases, only one DMAN card is allowed. The format of the DMAN control card where the DBD card has ACCESS=INDX

```
|          | DMAN | DD1=name          |
|          |      | [DLIOP=name]      |
|          |      | DEV1=device       |
```

The format of the DMAN control card where the DBD card has ACCESS=SEQ

```
|          | DMAN | DD1=name          |
|          |      | DEV1=device       |
|          |      | [DD2=name]        |
```

where

DMAN

identifies this Data Language/I control card as a DMAN control card. (See the setup examples at the end of this section.)

DD1=name

a one- to eight-character alphameric name that is the ddname of DD card for an ISAM data set, or input data set under SEQ organization. This parameter must be specified regardless of the data base organization.

DEV1=device

designates the physical storage device type on which the prime area for this data set is to be stored. A list of the possible physical devices follows:

Device Name	DEV1=	
DRUM	2301	(only when DBD ACCESS=SEQ)
DISK FILE	2302	
DISK PACK	2311	
DISK FACILITY	2314	
DATA CELL	2321	
TAPE	2400	(only when DBD ACCESS=SEQ)

(The underlined value may be used for DEV1=)

DLIOF=name

a one- to eight-character alphameric name that is the ddname of the DD card is required only if INDX was specified in the DBD card ACCESS parameter. This eight-character name becomes the ddname on the DD card for the OSAM data set. Omit this parameter if the DBD card ACCESS parameter equals SEQ or SAM.

DD2=name

a one- to eight-character alphameric name that is the ddname of the DD card for the output data set under SEQ. This parameter must be omitted if the DBD card ACCESS card equals INDX or ISAM.

The following table summarizes the parameters required on the DMAN control card for each of the Data Language/I access methods.

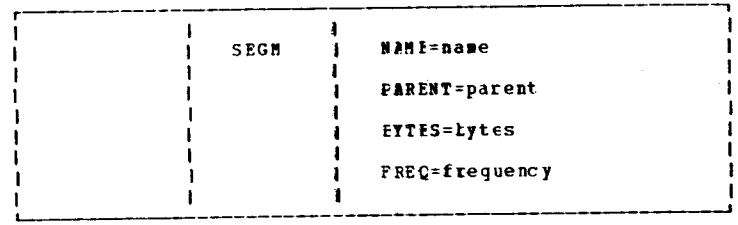
Access Method	Parameters Required
SEQ or SAM	DD1, DEV1, DD2
INDX or ISAM	DD1, DEV1, DLIOF

SEGM Control Card

At least one SEGM control card must immediately follow a DMAN set. The SEGM control card defines a segment to be contained in the data set group defined by a preceding DMAN control card. There may be a maximum of 255 segments defined. SEGM control cards must be entered in hierarchical order. The segments are physically stored in the data base record in the same order in which these cards are entered.

Provision has been made to accommodate the overflow of parameters on a SEGM control card to more cards. When this occurs, follow the rules stated above for overflow on the DMAN card.

The format of the SEGM control card is:



where

SEGM

the card-type code which identifies this as the SEGM control card.

NAME=name

a one to eight character alphameric name of the segment. Within one DBD, duplicate segment names are not allowed.

PARENT=parent

a one to eight character alphameric name of the parent segment of this segment, left justified, and must not have trailing blanks. The first SEGM control card for this job step is assumed to be the root segment and the "parent name" for the root segment must be a zero. (See Examples of DBD Generation section.)

BYTES=bytes

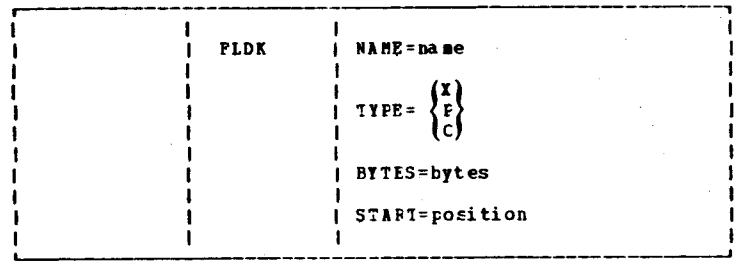
the number of bytes of storage required to accommodate a single occurrence of this segment. If all the fields of this segment are defined by FLD control cards and if none of the fields defined on the following FLD control card(s) overlap, this would be the sum of the lengths specified for these fields.

FREQ=frequency

an estimate of the number of times this segment is likely to occur for each occurrence of its parent segment. If this is the root segment, it is the estimate of the maximum number of data base records that appear in the data base being defined. If this is the root segment, this parameter must be an integer in the range 1-99999999 (Note: For readability, commas are not allowed.) The values given for dependent SEGM's are used in the computation of LRECL, blocking factor, and BLKSIZE. The frequency of occurrence of the root segment is used to determine the allocation of space required for cylinder index and prime ISAM storage at DBD generation execution. The output of DBD generation (the listing) specifies the number of tracks required for cylinder index and prime ISAM area allocation. If the root segment frequency is greater than 99999999, DBDGEN will not provide the definition of the prime space allocation required. The user must calculate this based on the number of root segments which will actually reside in the data base. The number produced by DBDGEN will be erroneous. These three figures are shown on the generation output and help the System Operation function in determining the SPACE parameters of the DD cards for data bases. The parameter is an estimate, not a limit.

FLDK Control Card

The format of the FLDK control card is:



where

FLDK

The card-type code that identifies this as the key field for this segment. The occurrences of this segment are kept in sort order on this field behind each occurrence of its parent segment. There must be one, and only one, key field defined for each segment. Each segment defined must have a key field defined by an FLDK card.

NAME=name

a one to eight character alphameric name of this field. Within one segment, duplicate FLDK names are not allowed.

TYPE=X, P, or C

designates the type of data that is to be contained in this field. The value of this parameter specifies that one of the following types of data is to be contained in this field:

X - hexadecimal data

P - packed decimal data

C - alphameric data or a combination of types of data

BYTES=bytes

specifies the length of this field in terms of bytes

If TYPE = X, BYTES must equal either 2 or 4.

If TYPE = P, maximum BYTES = 16.

If TYPE = C, maximum BYTES = 256.

START=position

specifies the starting position of this field in terms of bytes relative to the beginning of the segment. "Position" for the first byte of a segment is one (1). Overlapping fields are permitted. It must be remembered, however, that the sum of bytes (including bytes for fields that are not defined, and not including any common bytes of overlapping fields) cannot exceed the length of this segment as specified on the SEGM control card.

FLD Control Card

One or more FLD control cards may follow the FLDK control card. This card defines each of the fields, in the segment defined by the preceding SEGM control card, that may appear as part of a Data Langauge/I call qualification statement. All fields do not have to be defined. A maximum of 1000 FLDK and FLD cards may be defined in the entire DBD.

The format and parameters of the FLD control card are as outlined for the FLDK control card, above.

	[FLD]	

FLD

the card-type code that identifies this as the control card for an ordinary data field. There can be many data fields for any given Data Language/I segment. If a field is to be used in a segment search argument, it must be defined with a field control card.

If a nonkey field is not referred to with a Data Language/I Get call, no field control need be included in the DBDGEN.

DBDGEN Control Card

	DBDGEN	

Since it is the key to generating the data base description from the parameters specified above, this control card must be included.

FINISH Control Card

	FINISH	

This control card must be included. It sets the on-zero condition-code for link-edit if there are DBD generation errors.

END Control Card

	END	

Since it signifies the end of the DBDGEN, this control card must be entered.

DBDGEN Examples

1. Set up a Hierarchical Indexed Sequential data base consisting of a single data set group (see Figure 28). Each data base record will contain two segments of two and three

fields, respectively. The data base will be stored on a 2311 disk pack. The access method to be used is Indexed.


```
PRINT          NOGEN
DBD            NAME=DB,ACCESS=INDX
DMAN           DD1=DB,DEV1=2311,DLIOF=OVFL1
SEGM           NAME=S1,PARENT=0,BYTES=15,FREQ=100
FLDK           NAME=KEY,TYPE=X,BYTES=4,START=1
FLD            NAME=DATA,TYPE=C,BYTES=11,START=5
SEGM           NAME=S2,PARENT=S1,BYTES=20,FREQ=1
FLDK           NAME=KEY1,TYPE=X,BYTES=4,START=1
FLD            NAME=DATA1,TYPE=C,BYTES=12,START=5
FLD            NAME=DATA2,TYPE=P,BYTES=4,START=17
DBDGEN
FINISH
END
```

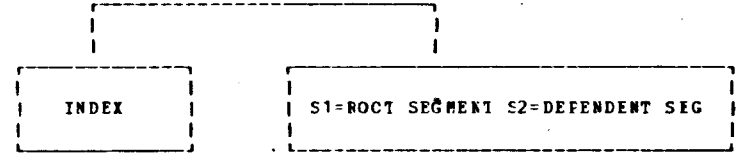


Figure 28. Single data set group

2. Set up a Hierarchical Indexed Sequential data base consisting of multiple (2) data set groups (see Figure 29). Each data base record will contain two segments of two and three fields, respectively. The data base will be stored on a 2311 disk pack. An * indicates the differences between single and multiple data set group organizations but is not physically punched on the DMAN card. The access method to be used is Indexed.

```
PRINT          NOGEN
DBD            NAME=DB,ACCESS=INDX
DMAN           DD1=DB,DEV1=2311,DLIOP=OVLF1
SEGM           NAME=S1,PARENT=0,BYTES=15,FREQ=100
FLDK           NAME=KEY,TYPE=X,BYTES=4,START=1
FLD            NAME=DATA,TYPE=C,BYTES=11,START=5
*DMAN          DD1=DS22,DEV1=2311,DLIOP=OVFL4
 SEGM          NAME=S2,PARENT=S1,BYTES=20,FREQ=1
 FLDK          NAME=KEY1,TYPE=X,BYTES=4,START=1
 FLD           NAME=DATA1,TYPE=C,BYTES=12,START=5
 FLD           NAME=DATA2,TYPE=P,BYTES=4,START=17
 DBDGEN
 FINISH
 END
```

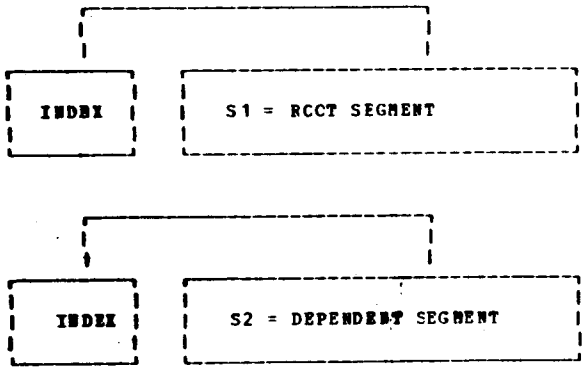


Figure 29. Multiple data set groups

3. Set up a Hierarchical Sequential data base. (A Hierarchical Sequential data base contains only a single data set group.) The data base record contains two segments with two and three fields, respectively (see Figure 30). The access method used is Sequential, and the data base is stored on 2400 series magnetic tape.


```
PRINT          NOGEN
DBD            NAME=DB2,ACCESS=SEQ
DMAN           DD1=DB2,DEV1=TAPE,DD2=DB3
SEGM           NAME=S1,PARENT=0,BYTES=15,FREQ=100
FLDK           NAME=KEY,TYPE=X,START=1,BYTES=4
FLD            NAME=DATA,TYPE=C,START=5,BYTES=11
SEGM           NAME=S2,PARENT=S1,FREQ=1,BYTES=20
FLDK           NAME=KEY1,TYPE=X,START=1,BYTES=4
FLD            NAME=DATA1,TYPE=C,START=5,BYTES=12
FLD            NAME=DATA2,TYPE=P,START=17,BYTES=4
DBDGEN
FINISH
END
```

Figure 30. Hierarchical sequential data base

DESCRIPTION OF DBD GENERATION OUTPUT

Three types of printed output and a load module which becomes a member of the partitioned data set with the generic name of DBD Library are produced by a DBD generation. Each of these outputs is described in the following paragraphs.

Control Card Listing

This is an exact reproduction of the character representation of the contents of each of the 80-column control cards. That is, it is a listing of the input card images to this job step.

Diagnostics

Errors discovered during the processing of each control card will result in diagnostic messages being printed immediately following the image of the last control card read before the error was discovered. The message may either reference the control card immediately preceding it or the preceding group of

control cards. It is also possible that more than one message could be printed per control card. In this case, they follow each other on the output listing. After all the control cards have been read, a further check is made of the reasonableness of the entire deck. This may result in one or more additional diagnostic messages.

Any discovered error will result in the diagnostic message(s) being printed, the control cards being listed, and the other outputs being suppressed. However, all the control cards will be read and checked before the DBD generation execution is terminated. The link-edit step of DBD generation will not be processed if a control card error has been found.

Assembly Listing

An Operating System/360 assembly language listing of the DBD created by DBD generation execution is provided.

Load Module

DBD generation is a two step Operating System/360 job. Step 1 is a macro assembly execution which produces an object module. Step 2 is a link-edit of the object module which produces a load module which becomes a member of the generic DBD library.

DBD Generation Error Conditions

The following are the DBD generation error conditions and The messages displayed for these conditions:

Error Message	Condition
DBD	---DBD010---Incorrect or Missing Access Method
DBD	---DBD020---DBD Name Parameter Not Specified
DBD	---DBD030---Too Many DBD Cards
DMAN	---DMAN010---Incorrect Device Specification
DMAN	---DMAN020---Incorrect Access Specification
DMAN	---DMAN030---DD2 parameter invalid with ACCESS equal to ISAM
DMAN	---DMAN040---Too Many DMAN Cards
DMAN	---DMAN050---BLKFACT specified but no LRECL
DMAN	---DMAN060---LRECL specified but no BLKFACT operand
DMAN	---DMAN070---LRECLxBLKFACT greater than Track length
DMAN	---DMAN080---Missing DLIOF operand with access equal to ISAM
DMAN	---DMAN090---DLIOF is present or DD2 is missing with access equal to SAM
DMAN	---DMAN100---DD1 operand omitted
DMAN	---DMAN110---DD1 and DD2 have same ddnames for HSAM
DMAN	---DMAN120---DD1/DLIOF duplicate ddnames for HISAM
SEGM	---SEGM10---Segment name not specified
SEGM	---SEGM20---Segment Bytes parameter not specified
SEGM	---SEGM30---Segment Frequency Parameter Not Specified
SEGM	---SEGM40---Root Segment Parent must equal zero
SEGM	---SEGM50---Parent Operand Not Specified for Dependent Segment
SEGM	---SEGM60---Too Many SEGM cards; 255 maximum

SEGM	---SEGM070---Segment Length greater than DASD track
SEGM	---SEGM080---Segment Length Specified as zero
SEGM	---SEGM090---Segment Frequency of Zero Invalid
SEGM	---SEGM100---Duplicate segment names
FLD	---FLD010---Field Name Parameter not specified
FLD	---FLD020---Field Bytes Parameter not specified
FLD	---FLD030---Start parameter not specified
FLD	---FLD040---Type parameter not specified or invalid
FLD	---FLD050---FLDK card not first after SEGM card
FLD	---FLD060---Too Many FLD or FLDK cards; 1000 maximum
FLD	---FLD070---Field Length extends beyond Segment end
FLD	---FLD080---First Byte of Segment starts with byte 1
FLD	---FLD090---Field Length Specified as Zero
FLD	---FLD100---Duplicate field name in same segment
FLDK	---FLDK010---Key Field Specified Inappropriately
DBDGEN	---DGEN010---Segment X Parent Y not found
DBDGEN	---DGEN020---Invalid Number of DMAN cards for Access Method Specified
DBDGEN	---DGEN030---DAM not supported

DBDGEN	---DGEN040---No Segments for DMAN X
DBDGEN	---DGEN050---DAM not specified
DBDGEN	---DGEN060---Errors in this DBD
DBDGEN	---DGEN070---Too many levels in data base segment hierarchy
DBDGEN	---DGEN080---First segment in secondary data set group lower than level two
FINISH	---FINI10---No Successful DBD's in this run

Because DBD generation is composed of Operating System/360 assembly language macro-instructions, omission or invalid sequence in DBD control cards, or invalid key word parameters, will result in error statements from the Operating System/360 assembler.

PROGRAM SPECIFICATION BLOCK (PSB) GENERATION

PSB Requirements

Before an application program can be executed under Data Language/I, it is necessary to describe that program and its use of data bases through a PSB generation. The PSB generation control cards supply the identification and characteristics of the PCB's (Program Communication Block) representing data bases to be used in the application program. There must also be a control card supplying characteristics of the application program itself. There must be one PSB for each batch processing program. The name of the PSB and its associated program must be the same.

PSB generation places the PSB into the PSB Library. The PSB and its PCB's (all PCB's to be used by the program are contained within the PSB) are stored in the library.

There are three basic types of control cards required for a PSB generation:

- PCB control cards for Data Language/I data bases
- SENSEG control card for data base sensitive segments
- PSBGEN control card for each PSB

The PCB list passed to the application program upon entry should be referenced within the processing program by the included names for making Data Language/I calls and interrogating PCB information (that is, status codes and feedback information).

Except that coding must not begin in column 1, the format of the three PSB generation control cards is free form. The operation code (PCB, SENSEG, PSBGEN) must be followed by at least one blank. The key word operands must contain no blanks and must be separated by commas.

PCB Control Card - Data Language/I Data Base PCB

The first type of control card in a PSB generation deck is one that specifies a description of a PCB for a Data Language/I data base. The format for this type of control card is:

	PCB	TYPE=DB
		DBDNAME=name
		PROCOPT=X
		KEYLEN

TYPE=DB

is a required key word parameter for all Data Language/I data base PCB's.

DBDNAME=name

is the parameter key word for the name of a data base description that was produced by a DBD generation run. This DBD name associates this PCB with a particular Data Language/I data base. The value for "name" must be eight characters or less in length.

PROCOPT=X

is the parameter key word for the processing options that will be used by the batch processing program. The value for "X" must be one character. Possible values for the processing options are:

G - for get functions

A - for get, delete, insert, and replace

L - for loading a data base

Only valid are:

HISAM - I
G
A

HSAM - G
L

KEYLEN

is the parameter key word for the maximum length in characters of the concatenated keys for all complete hierarchical paths in the data base. This value is determined by identifying a complete hierarchical path, one that goes from the root segment to a lowest level segment, and summing the key lengths for each key in that path. This value is the length of the concatenated keys for that path. This process is repeated for each complete hierarchical path. The maximum length of concatenated keys is then specified in the KEYLEN parameter.

SENSEG Control Card - Sensitive Segments

The SENSEG control card is used in conjunction with the PCB card for a Data Language/I data base and describes the segments in the data base to which the program is "sensitive". There must be one or more SENSEG cards for each data base PCB card, and they must immediately follow the PCB card to which they are related. There must be one card for each segment. The format of the SENSEG card is:

	SENSEG	sensitive-seg-name= name of segment parent-seg-name= parent name of sensitive segment above

sensitive-seg-name

is the name of the segment as defined in the SEGM card at DBD generation time. The field is from one to eight alphameric characters.

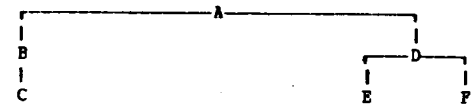
parent-seg-name

is the name of the segment which is the parent to the sensitive segment above. The parent name must also agree with the parent name in the SEGM card at DBD generation time.

Within the definition of each sensitive segment type, the required format is to first specify the name of the segment being identified, follow that by a comma, and then give the segment name of this segment's parent. Since the root segment has no parent, its parent name is omitted.

The order in which sensitive segment cards are arranged must follow the hierarchical structure specified in the DBD generation. The definition should begin with the root segment and proceed down the leftmost path to the lowest level of the structure, then back up to a higher level and down again, continuing toward the right until the entire structure has been specified.

EXAMPLE OF SEGMENT DEFINITION: If the structure of the data is:



and the program is sensitive to the whole structure.

The complete PCB and SENSEG set for the above Data Language/I data base structure might be written as follows:

```
Col. 10     Col. 16                                   Col. 72
PCB         TYPE=DB,DBDNAME=DATABASE,                 X
            PROCOPT=G,KEYLEN=22
SENSEG      A
SENSEG      B,A
SENSEG      C,B
SENSEG      D,A
SENSEG      E,D
SENSEG      F,D
```

PSBGEN Control Card

The third type of control card required for a PSB generation is one that specifies characteristics of the application program. The format for this card is:

	PSBGEN	LANG=XXXXX
		PSBNAME=YYYYYYYY

LANG=XXXXX

is the parameter key word for the compiler language in which this batch processing program is written. The XXXXX value for this parameter must be either COBOL, PL/I, or ASSEM, with no trailing blanks.

PSBNAME=YYYYYYYY

is the parameter key word for the alphameric name of this PSB. The YYYYYYYY value for the PSBNAME must be eight characters or less in length. This name becomes the load module name for the PSB in the PSB library. This name must be the same as the program load module name in the program library. No special characters may be used in the name.

It should be noted that there may be several PCB-TYPE-DB control cards, but only one PSBGEN control card in a PSB generation card deck. The PSBGEN card must be the last control card in the deck preceding the END card.

The three types of PSB generation control card must be followed by an END card. The END card is required by the macro assembler to indicate the end of the assembly data.

DESCRIPTION OF PSB GENERATION OUTPUT

PSB generation produces three types of printed output and one load module which becomes a member of the partitioned data set with the generic name of PSB Library. Each of these outputs is described in the following paragraphs.

Control Card Listing

This is an exact reproduction of the character representation of the contents of each of the 80-column control cards. That is, it is a listing of the input card images to this job step.

Diagnostics

Errors discovered during the processing of each control card will result in diagnostic messages being printed immediately following the image of the last control card read before the error was discovered. The message may either reference the control card immediately preceding it or the preceding group of control cards. It is also possible that more than one message could be printed for each control card. In this case, they follow each other on the output listing. After all the control cards have been read, a further check is made of the reasonableness of the entire deck. This may result in one or more additional diagnostic messages.

Any discovered error will result in the diagnostic message(s) being printed, the control cards being listed, and the other outputs being suppressed. However, all the control cards will be read and checked before the PSB generation execution is terminated. The link-edit step of PSB generation will not be processed if a control card error has been found.

Assembly Listing

An Operating System/360 assembly language listing of the PSB created by DBD generation execution is provided.

Load Module

PSB generation is a two-step Operating System/360 job. Step 1 is a macro assembly execution which produces an object module. Step 2 is a link-edit of the object module, which produces a load module which in turn becomes a member of the generic PSB library.

PSB Generation Error Conditions

Erroneous Control Card	Error Message
PCB	---PCB010---PCB Type parameter missing or invalid
PCB	---PCB020---PCB LTERM parameter not specified for PCB
PCB	---PCB030---DBDNAME parameter not specified for DB PCB
PCB	---PCB040---KEYLEN parameter not specified for DB PCB
PCB	---PCB050---PROCOPT parameter not specified for DB PCB
PCB	---PCB060---DBDNAME specified for PCB
PCB	---PCB070---PROCOPT specified for PCB
PCB	---PCB080---KEYLEN operand for TP PCB
PCB	---PCB090---LTERM operand specified for DB PCB
PCB	---PCB100---Invalid processing option in PCB
SENSEG	---SEG010---SEG Parameter Missing
SENSEG	---SEG020---Too Many Sensitive Segments
SENSEG	---SEG030---Duplicate Segment Names
PSBGEN	---PSB010---PCB in Error, Generation Terminated
PSBGEN	---PSB020---PSBNAME not specified
PSBGEN	---PSB030---Invalid Lang. Operand
PSBGEN	---PSB040---No Sensitive Segments for DB PCB
PSBGEN	---PSB099---System Error, Generation terminated

Because PSB generation is composed of Operating System/360 assembly language macro-instructions, errors in omission or invalid sequence of control cards, or invalid parameters on control cards will result in additional errors specified by Operating System/360 during PSB generation.

SECTION 2. SYSTEMS OPERATION OF DATA LANGUAGE/I

SYSTEM DISTRIBUTION, HANDLING, AND MAINTENANCE

SYSTEM DISTRIBUTION

The distribution of the Data Language/I system is accomplished by means of nonlabeled, nine-track, 800-bpi and nonlabeled, seven-track, 800-bpi magnetic tape. The potential user of Data Language/I should order either the seven- or the nine-track tape. The basic distribution tape includes three data sets. These are:

- Data Language/I Macro-instruction Library (DLI.GENLIB)
- Data Language/I Load Module Library (DLI.LOAD)
- Data Language/I Source Module Library (DLI.SOURCE)

The three libraries are unloaded copies of partitioned data sets. They have been moved to tape using the IBM OS/360 IEHMOVE program. When the user receives the distribution tape, the IEHMOVE program should be employed to copy these data sets to direct access storage. The following job control language statements and utility control cards should assist in the copy execution.

Nine-Track Tape

```
//COPY      JOB   848,NAME,MSGLEVEL=1
//          EXEC  PGM=IEHMOVE,REGION=100K
//SYSPRINT  DD    SYSOUT=A
//SYSUT1    DD    UNIT=2311,DISP=OLD,VOLUME=SER=111111
//TAPE1     DD    UNIT=(2400-4,,DEFER),DISP=OLD,                    X
//                VOLUME=SER=SCRTCH,DCB=(LRECL=80,                  X
//                RECFM=FB,BLKSIZE=800,DEN=2),LABEL=(,NL)
//DISK1     DD    UNIT=2311,DISP=OLD,VOLUME=SER=ILIB01
//DISK2     DD    UNIT=2311,DISP=OLD,VOLUME=SER=ILIB02
//SYSIN     DD    *
          COPY  PDS=DLI.GENLIB,                                   X
                FROM=2400-4=(SCRTCH,1),                           X
                TO=2311=ILIB01,FROMDD=TAPE1
          COPY  PDS=DLI.LOAD,FROM=2400-4=(SCRTCH,2),              X
                TO=2311=ILIB01,FROMDD=TAPE1
          COPY  PDS=DLI.SOURCE,                                   X
                FROM=2400-4=(SCRTCH,3),                           X
                TO=2311=ILIB02,FROMDD=TAPE1
/*
```

Seven-Track Tape

```
//COPY2     JOB   848,NAME,MSGLEVEL=1
//          EXEC  PGM=IEHMOVE,REGION=100K
//SYSPRINT  DD    SYSOUT=A
//SYSUT1    DD    UNIT=2311,DISP=OLD,VOLUME=SER=111111
//TAPE1     DD    UNIT=(2400-2,,DEFER),DISP=OLD,                    X
//                LABEL=(,NL),                                      X
//                VOLUME=SER=SCRTCH,DCB=(LRECL=80,                  X
//                RECFM=FB,BLKSIZE=800,DEN=2,TRTCH=C)
//DISK1     DD    UNIT=2311,DISP=OLD,VOLUME=SER=ILIB01
//DISK2     DD    UNIT=2311,DISP=OLD,VOLUME=SER=ILIB02
//SYSIN     DD    *
          COPY  PDS=DLI.GENLIB,TO=2311=ILIB01,                    X
                FROM=2400-2=(SCRTCH,1),FROMDD=TAPE1
          COPY  PDS=DLI.LOAD,TO=2311=ILIB01,                      X
                FROM=2400-2=(SCRTCH,2),FROMDD=TAPE1
          COPY  PDS=DLI.SOURCE,TO=2311=ILIB02,                    X
                FROM=2400-2=(SCRTCH,3),FROMDD=TAPE1
/*
```

Those parameters which are underlined are user-specifiable (for example, 2314 rather than 2311). The region parameter is required only for Operating System/360 MVT execution. Generic name 2400-4 is nine track at 800 bpi with DCB=(DEN=2); generic

name 2400-2 is seven track with data conversion at 800 bpi with DCB=(DEN=2).

SYSTEM HANDLING

Once the libraries have been copied from the distribution tape(s) to direct access storage, the user is able to begin to tailor Data Language/I to his data processing environment. The tailoring of the Data Language/I system to a particular user's data processing environment is accomplished with the system definition macro-instructions which are contained within DLI.GENLIB. Because the Data Language/I system executes with a collection of control blocks, which describe the user's data processing environment, system definition is required. These control blocks, which describe application programs, data bases, and other resources, are constructed by the system definition process.

The user must prepare a control card input deck to the system definition. The control card types and formats are described later in this manual. Once the control card deck has been prepared, it is appended to a package of job control language for the macro-instruction assembly of system definition (see Figure 31).

The output from Data Language/I system definition includes:

- Generation and placement in the DLI.LOAD or SYS1.LINKLIB library of Data Language/I control program control blocks.
- Generation and placement in the DLI.LOAD or SYS1.LINKLIB library of the Data Language/I batch processing nucleus
- The linkage edit of a user supervisor call (SVC) used for OSAM multivolume execution. This is placed in the DLI.RESLIB library. The user can specify the desired SVC number.
- The naming and creation of the OSAM channel end appendage module into DLI.RESLIB. The user can specify the module name and must move the module to SYS1.SVCLIB.
- The moving of procedures to DLI.PROCLIB or SYS1.PROCLIB. These procedures are used for data base description (DBD) and program specification block (PSB) generation, batch region execution, etc.

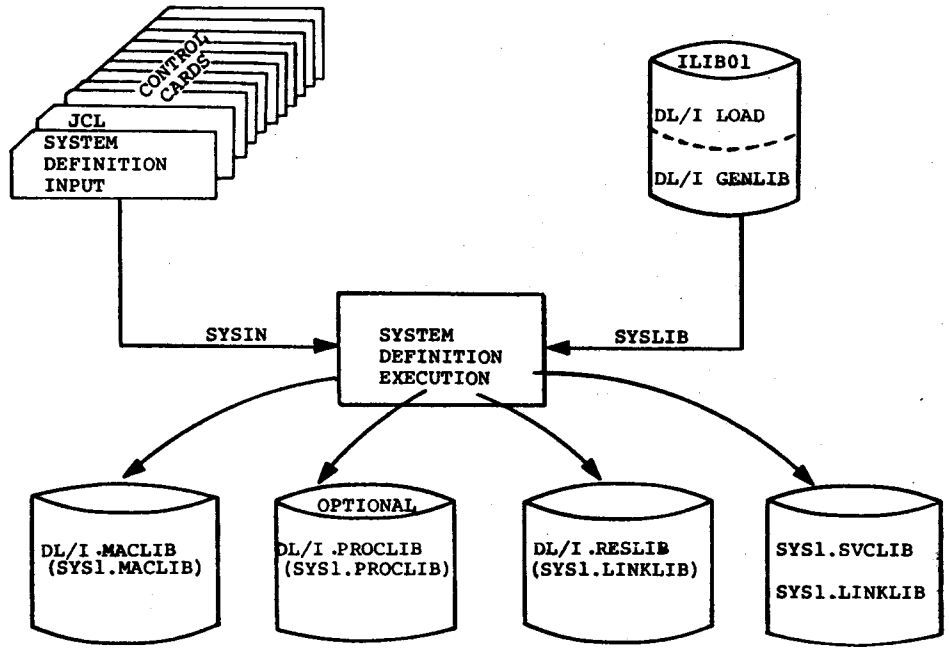


Figure 31. System definition handling

Once Data Language/I system definition has been performed, the SVC routine must be link-edited with the Operating System/360 nucleus.

The Data Language/I user must have provided for one Type 2 user SVC routine in his Operating System/360 system generation. This SVC is used by OSAM. The procedure for relink-editing the Operating System/360 nucleus with the user SVC routine is specified later in this manual.

Once system definition and the SVC-Operating System/360 nucleus link-edit is performed, the user must allocate direct access space for the DBD and PSB libraries. All these data sets should be cataloged. Later sections of this manual describe the execution of these functions.

Finally, the user is ready to create DBD's, PSB's, and application programs. Before any processing may be performed, the required data bases must be created in the batch environment.

PLANNING

Batch Processing Region System Flow

Figure 32 shows the batch system flow using Data Language/I executing under Operating System/360. "Events" refer to figure 32.

1. The application program for batch processing is initiated through the job management routines of Operating System/360 (Event 1).

2. Part of the control facility of Data Language/I is invoked (Event 2) to ensure that the application program is only referencing the specified data bases.

3. Then the Data Language/I facility is invoked. The highest level Data Language/I module analyzes the data base call request. Depending upon the I/O function requested in the call, the insert (Event 11), the retrieve (Event 3), the load (Event 15), the HSAM (Event

16), or the delete/replace (Event 13) module is invoked. These modules subsequently invoke either the ISAM modules (Event 4) or the OSAM modules (Event 5) to reference the data base.

4. The data segment is moved to or from the application program's I/O work area and the data I/O buffers used by the access method (Events 8, 9).

5. Also, the data segment is moved to or from the I/O buffers and the data sets representing the data base (Events 6, 7, 13, 14).

6. After the Data Language/I I/O request has been completed, control is returned from either ISAM or OSAM (Events 4, 5) to one of the Data Language/I modules. Subsequently, control is returned to the Data Language/I analyzer module (Events 3, 11, 12) and finally to the application program (Event 10).

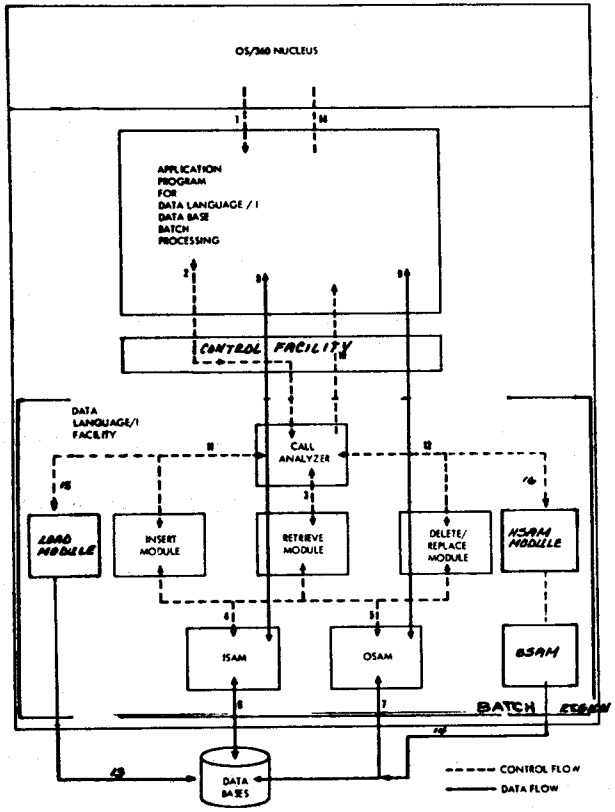


Figure 32. Batch processing region system flow

There is a capability in a batch processing region to specify through job control language parameters a PSB and an application program with different names. Normally there is a one-to-one relationship between PSB and application program. However, the ability to specify different PSB's for one application program opens the door for the user of IMS/360 to create some general purpose "utility programs" which use multiple PSB's (one at a time). Later this manual describes the JCL available to assist in implementing this capability.

Impact of MFT-II and MVT on Data Language/I

Certain features in the MFT and MVT operating system options affect the performance of Data Language/I.

One of the major differences is the support of Operating System/360 in the area of storage management. Storage management for MVT is extended to include subpool management, whereas, under MFT-II, storage management resembles PCP storage management; that is, supervisor control blocks are not resident in system queue space and modules of code are packed contiguously in an MFT-II partition. Under MVT, the storage management algorithm is based upon use of 2K blocks of storage as the minimum quantity that can be manipulated by the storage management facility. In addition, in MVT, FETCH= routines may make a different subpool call to storage management based upon module attributes. MFT-II is essentially transparent to the usability attribute of a load module. As a result, certain Data Language/I load modules will reside in protection key 0 storage within the problem program region in MVT, whereas under the MFT-II environment these modules will essentially be unprotected from modification. Since MVT does provide the added feature of protected code in the problem program region, it might be desirable at the time the system is defined that the Data Language/I resident library (that is, the library in which all Data Language/I executable modules will reside) be in fact either SYS1.LINKLIB or SYS1.SVCLIB. One of these two libraries should be specified, since these are the only two from which Fetch will store in protect code 0 core in the problem program region.

Another consideration in the differences between MFT-II and MVT is in system timing. In MVT regions, the TIME= parameter for job step timing is available, whereas in an MFT-II environment this capability is not implemented. This should not be considered a disadvantage of MFT-II; however, the user should be aware that, since the Data Language/I regions will be expected to persist for many hours of continuous operation, it is necessary to specify the TIME= parameter in the job card which initiates the region. If the TIME= parameter is not used and job step timing has been selected at Operating System/360 system definition time, the default time which appears in the reader procedure used to read the Data Language/I JCL will apply, and Data Language/I will be terminated abnormally by Operating System/360.

ESTIMATING STORAGE REQUIREMENTS

One of the things that must be planned for is the storage requirements of an Operating System/360-Data Language/I installation.

DATA BASE SPACE ALLOCATION, FORMAT, CREATION, AND REORGANIZATION

Data Language/I Data Base Space Allocation

When direct-access storage is required for a data base, the amount of space needed and the device type must be specified. Data Language/I follows the same approach as Operating System/360. Refer to the manual, Supervisor and Data Management Services, IBM Form No. C28-6646, Part 3: Data Set Disposition and Space Allocation.

The amount of space required can be specified in terms of blocks, tracks, or cylinders. If it is desired to maintain device-independence across direct-access device types, space requirements should be specified in terms of blocks. Otherwise, if the request is in terms of tracks or cylinders, such items as their capacity must be considered. ISAM data sets must be allocated by cylinder. Table 12 of the Supervisor and Data Management Services Manual lists the physical characteristics of a number of direct-access storage devices. The amount of space is supplied in the data definition (DD) statement for the data set.

Allocating the space for a Data Language/I data base which uses ISAM and OSAM is similar to allocating space for an Operating System/360 indexed sequential data set; similar because an Operating System/360 data set can be divided into three areas, prime, index, and overflow. The three areas of a Data Language/I data base are index, prime, and OSAM overflow.

Before an example is shown of how one might allocate his logical records in the Data Language/I data base environment, another feature of the Data Base Description (DBD) generation utility is discussed.

Normally, DBD generation computes from the user's definition of segment frequency the logical record length (LRECL) of a data base. It considers the device and rounds to the next higher 1/4 track, 1/3 track, or 1/2 track. The computed LRECL will not exceed 1/2 track for any device.

For the data base designer, Data Language/I has two additional parameters which can be inserted when it executes

the DBD generation. These provide an additional means of specifying the LRECL and blocking factor (BLKFACT) for a data set within a data base.

In the DMAN control card, the additional parameters are LRECL and BLKFACT. Instead of DBD generation specifying the LRECL, it can be overridden by specifying the LRECL and the BLKFACT.

With the knowledge of the data base structure, the application program that will access that structure, and the tools of DBD generation, the following space allocation example will be meaningful. The example deals with an IBM 2314 Direct Access Storage Facility.

Allocation Problem Example

When a Data Language/I Hierarchical Indexed Sequential organization (HISAM) data base is loaded on an IBM 2314 Direct Access Storage Facility, it is necessary to allocate space for that data base with JCL Data Definition Statements. The creation of a Data Language/I single data set group data base may require up to three DD statements, one each for the index, prime, and OSAM overflow areas. This example should provide assistance in initially determining the amount of space to allocate to these areas for any specific application. Each data set group within a Data Language/I multiple data set group can be treated as a Data Language/I single data set group data base when determining space requirements. The output of DBD generation also supplies minimum primary ISAM and OSAM allocation requirements.

ISAM PRIME AREA: This area contains the majority of the data records and all related track indexes. For this example, each root segment and all its dependent segments comprise a single logical data base record. For a specific data base, a logical record could vary from only the root segment to the root segment with a maximum number of segment occurrences for each dependent segment type. The distribution of the lengths of logical records of this example's data bases is plotted in Figure 33.

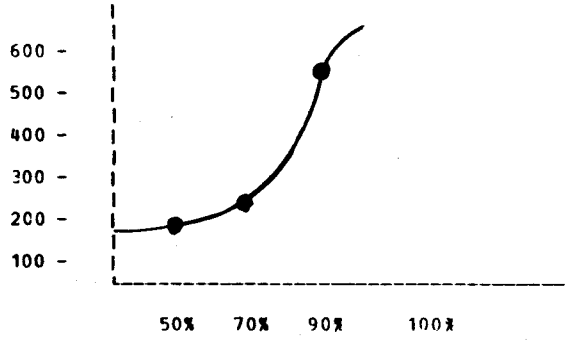


Figure 33. Logical record length distribution

The graph of Figure 33 indicates that 50% of the logical records are 150 bytes or less in length; 70% of the logical records are 200 bytes or less in length; 90% of the logical records are 500 bytes or less in length; and 100% of the logical records are 600 bytes or less in length.

With fixed-length ISAM, it is necessary to establish a fixed value for the logical record length (LRECL) in the prime area. If a value of 600 bytes is selected for the LRECL, then all logical records would fit in the prime area. However, 90% of the records have at least 100 bytes of slack or wasted space; 70% of the logical records have at least 400 bytes of unused space.

In this example, if a value less than 600 bytes is selected for the size of the LRECL, the ISAM prime area is not capable of holding all the logical data base records. Those dependent segment occurrences which do not fit in an ISAM prime LRECL are housed in OSAM overflow records. Therefore, the determination of an ISAM prime LRECL must consider the tradeoff between storage in the ISAM prime area and in the OSAM overflow area.

To determine a best balance between ISAM Prime and OSAM Overflow, the following points must be considered:

1. Access to data that is wholly contained in the Prime area is more rapid than access to data contained in two areas. Access is even slower to those logical data base records that require more than one overflow record.

2. Disk space allocated to OSAM overflow can be used to hold segment occurrences that overflow from any logical data base record. Unused space in the Prime area is tied to specific roots.

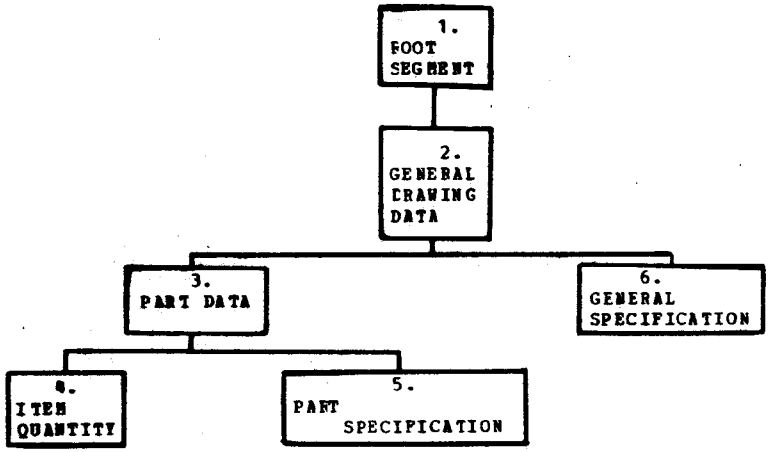
3. Records are blocked in the Prime area but unblocked in the OSAM Overflow area. This difference is negligible for large data base records, but can be significant for small records. For example, with an LRECL of 27 bytes, 4864 records can be stored in one cylinder in the prime area with half-track blocking. Only 840 similar records can be stored in one cylinder of Overflow.

4. The nature of the accesses to the large logical data base records also has an important effect. If the large logical data base records are highly used, the size of the prime LRECL should be increased to completely house more logical records, and the total size of OSAM overflow should be reduced. If the large logical data base records are infrequently accessed, an opposite shift should be made to increase the use of OSAM overflow.

Considering these relevant factors for a specific data base, a percentage balance must be established between the ISAM prime and the OSAM overflow. For example, it may be best in the context of optimizing space and time utilization, that 90% of the logical data base records completely fit in the prime area and 10% require some OSAM overflow storage. After this percentage is selected, the frequency of dependent segment occurrences is developed for the 90th percentile of the parent segments. The 90% is an estimated value for this specific data base. This is similar to the DBD generation requirements, except that the frequency will apply to 90% or more of the parents rather than to all of them. Those dependent segments that occur with certainty, that is, with fixed frequency for 100% of that segment's parents, will be specified at their full value.

Taking the segments contained in the example's data base (Parts List Data Base), Figure 34 shows the hierarchical structure relationship of the segments, a table of parent estimated frequency, and segment length. Note that the segment length in this table contains two overhead bytes needed for Data Language/I.

PARTS LIST DATA BASE



Data Base	Name of Segment	Parent	Estimated No. per Parent	Segment Length
Parts List Data	1. Root Segment	--	--	20
	2. General Drawing Data	1.	1	8
	3. Part Data	2.	10(90% have 10 or less	15
	4. Item Quantity	3.	2	5
	5. Part Specification	3.	1	10
	6. General	2.	4(90% have 4 or less)	25

Figure 34. Hierarchical segment structure and table of parts list data base

The calculations of the prime LRECL are:

Prime LRECL=20 + 1 [8 + 10 [15 + 2(5) + 1 (10)] + 4[25]] + 3*

Prime LRECL = 481 bytes

* The addition of 3 represents overhead bytes for Data Language/I.

A more extensive discussion of Data Language/I record formats follows.

This establishes the needed fixed record size (calculated LRECL) for the ISAM prime area, in order that 90% of the logical data base records can be completely housed in the prime area.

The next step is to determine the Data Language/I LBECL which considers the calculated LRECL, the track length, and the blocking factor. Assuming half-track blocking, there are 3476 bytes in one block on the 2314 storage facility. The value of calculated LRECL is divided into 3476 to determine the number of records in a block, and the remainder from this division is equally distributed among the records.

The results of this process can be tabulated in the convenient form shown in Figure 9:

Calculated LRECL Range	FINAL LRECL	LRECL's per block
1739 - 3476	3476	1
1159 - 1738	1738	2
870 - 1158	1158	3
696 - 869	869	4
580 - 695	695	5
497 - 579	579	6
435 - 496	496	7
387 - 434	434	8
348 - 386	386	9
317 - 347	347	10
291 - 316	216	11
268 - 290	290	12
249 - 267	267	13
233 - 248	248	14
218 - 232	232	15
205 - 217	217	16

Figure 35. LRECL for half-track blocking on the 2314 direct access storage facility

A calculated LRECL of 481 falls in the tablulated range of 435 - 496, which results in a final LRECL of 496 bytes, with seven records in a half-track block.

The next step is to calculate the total amount of space that is required for the ISAM prime area. An estimate must be made for the number of roots that exist in the data base. In the example under consideration, there are 50,000 roots, that is, 50,000 logical data base records.

Fourteen records are blocked on a single track, and there are 19 tracks on a cylinder, excluding track indexes. Therefore, 266 records fit in a cylinder, and 188 cylinders hold the prime area. Since there are 199 usable cylinders in one 2314 pack, the ISAM prime area requires about 95% of one pack.

INDEX AREA: This area contains master and cylinder indexes associated with the data set. It exists for any ISAM data set that has a prime area occupying more than one cylinder. The user can place this area on 2314 or 2301/2303 drum.

OSAM OVERFLOW: The OSAM overflow area holds those dependent segment occurrences of a logical data base record that do not fit on one LRECL in the ISAM prime area. One or more OSAM records may be required in addition to one ISAM record to hold one logical data base record. A physical break in a logical data base record must not divide a segment occurrence. The determination of the amount of space needed for the OSAM overflow area depends on the percentage figure used to develop the space for the prime area. In the Parts List example, 90% of the logical data base records are expected to completely fit into the prime area.

This percentage figure must also consider the fact that the final LRECL is equal to or larger than the calculated LRECL. This may have the effect of increasing the percentage of logical records that completely fit in the prime area. Assuming that 90% of the logical data base records do completely fit in the prime area, the remaining 10% are of interest in calculating the OSAM requirements.

It is necessary to determine which dependent segment-types overflow and how many of these fit into one OSAM record. It is assumed in this example that, in the Parts List Data Base, segments 3 and 6 occur approximately in the proportion of 2-1/2 to 1. Note that, if segment 3 overflows, the children of segment 3, segments 4 and 5, overflow with it. This means that one OSAM record holds approximately 20 segment 3's and 8 segment 6's. The next step is to determine what part of the 10% of the logical data base records will overflow into only one OSAM record; then two OSAM records are considered, and so forth, until the entire 10% has been specified. For the example under consideration, the statistics are in Figure 35.

Number of OSAM Records Required per Logical Record	% of Data Base	Number in Data Base	Number of OSAM Records Needed
0	90%	45,000	0
1	6%	3,000	3,000
2	3%	1,500	3,000
5	1%	500	2,500
	100%	50,000	8,500

Figure 35. Statistics on 10% of the overflow of the logical records in a data base

Since the overflow records are 496 bytes long and unblocked, 220 records can be stored in one cylinder. Therefore, 39 cylinders are needed for OSAM overflow.

EXAMPLE CONCLUSION: Following the above process, which is an estimation process not a precise algorithm with exact values, estimates have been developed for the 2314 space requirements for a sample problem. The space requirements for the Parts List Data Base are 188 cylinders for the ISAM prime area, and 39 cylinders for OSAM Overflow. These values would be used in data definition statements to allocate space for the Parts List Data Base.

It should be noted that the objective of this estimation process is to develop values for use in initially loading and processing a data base. File growth has not been included in this example but it must be considered. The developed space requirements may be later refined by historical data on actual space usage. Even without historical data, however, the use of the estimation process described here should result in reasonably accurate initial space requirements for a hierarchical data base of any degree of complexity.

ISAM DCB Option Codes

The following option codes should be utilized and specified on the DD cards of the job for each data base creation.

DCB = (DSORG=IS,OPTCD=WM, [RECFM=FB])

where

W =

is Write Check

M =

is Master Index creation (optional)

The user must not specify OPTCD=L which indicates the presence of a delete byte in the ISAM logical record. The user should not specify OPTCD=I for ISAM independent overflow, because ISAM is not used to make additions to a data base.

The user may specify the RECFM or it may be omitted. If RECFM is specified, it must state RECFM=FB.

OSAM DCB Option Codes

No DCB parameters need be specified on the DD card for the OSAM data set within a Data Language/I data base. However, the user may specify DCB =(DSORG=PS) if desired.

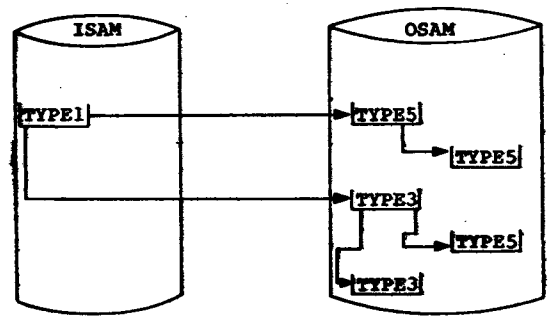
Data Language/I Record Format

In the previous section, the importance of the logical record length was stressed and a method of calculating that length, which included two overhead bytes in each segment contained within the logical record, was included. The logical record length also included an overhead of three bytes, which is added to the total calculation.

For the data base designer, it is felt that the actual Data Language/I record format is required for complete understanding of the organization of the data base. The explanation following assumes that this data base is of the Hierarchical Indexed Sequential organization and contains multiple data set groups.

Figure 37 shows the relationship between the different types of logical record formats. The relationship differs between the data set group, the ISAM or OSAM data set organization, and whether the segment is root or dependent. With reference to the Type numbers in Figure 37, the following explanation is offered:

PRIMARY DSG



SECONDARY DSG

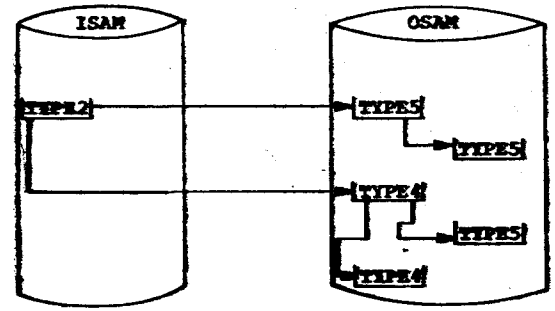
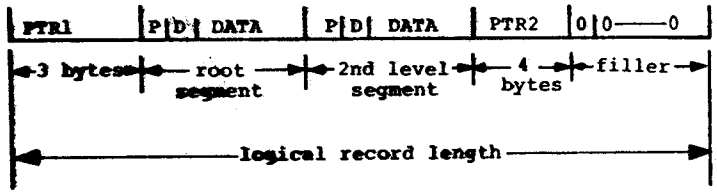


Figure 37. Logical record format relationship

TYPE 1 RECORD FORMAT: The Type 1 logical record Data Language/I format is contained within the primary data set groups, is in the ISAM organization, pertains to root segments, and its primary occurrence comes at initial loading of the data base. The format is:



where

PTR1 = PTR pointer to next root segment (Type 3 format) in OSAM

P = one byte of overhead per segment for physical code of segment type

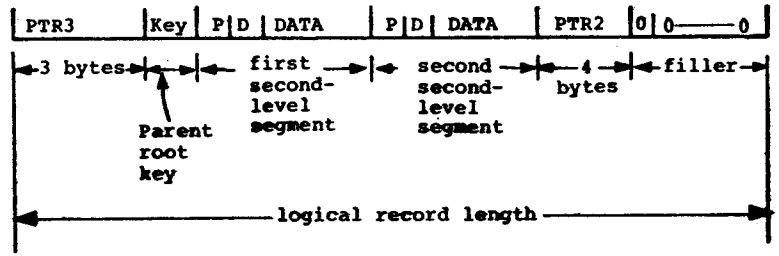
D = one byte of overhead per segment for delete code

DATA = actual data for that segment, including its key

PTR2 = PTR pointer to next dependent segment (Type 5 format) in OSAM, if there is necessity for overflow. Otherwise, all four bytes are binary zeros.

filler = The remaining area not used by data, or the PTR pointer is filled with binary zeros

TYPE 2 RECORD FORMAT: The Type 2 logical record Data Language/I format is contained within the secondary data set group, is in the ISAM organization, pertains to second level segments (can not start with root segments; must be no higher than second level), and its primary occurrence comes at initial loading of the data base. The format is:



where

PTR3 = PTR pointer to next second level segment (Type 4 format) in OSAM

Key = The parent root key

P = one byte of overhead per segment for physical code of segment type

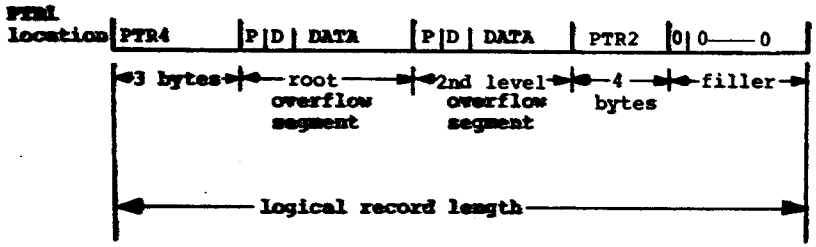
D = one byte of overhead per segment for delete code

DATA = actual data for the segment, including its key

PTR2 = PTR pointer to next dependent segment (Type 5 format) in OSAM, if there is necessity for overflow. Otherwise, all four bytes are binary zeros.

filler = The remaining area not used by data, or the PTR pointer is filled with binary zeros

TYPE 3 RECORD FORMAT: The Type 3 logical record Data Language/I format is contained within the primary data set group, is in the OSAM organization, pertains to root segment overflow, and its primary occurrence comes at insertion or addition to the data base. The format is:



where

PTR1 location = location of PTR1 pointer in Type 1 format

PTR4 = PTR pointer to next root segment in OSAM (Type 3 format)

P = one byte of overhead per segment for physical code of segment type

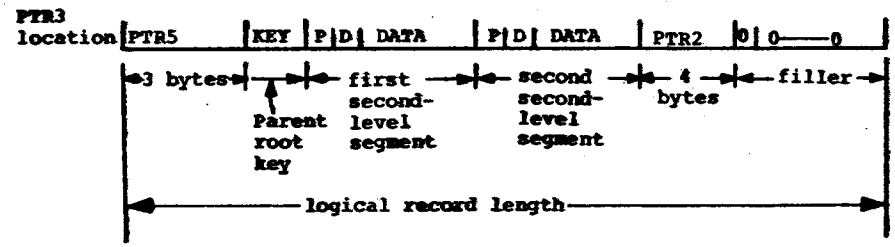
D = one byte of overhead per segment for delete code

DATA = actual data for that segment, including its key

PTR2 = PTR pointer to next dependent segment (Type 5 format) in OSAM, if there is necessity for overflow. Otherwise, all four bytes are binary zeros.

filler = The remaining area not used by data, or the PTR pointer is filled with binary zeros

TYPE 4 RECORD FORMAT: The Type 4 logical record Data Language/I format is contained within the secondary data set group, is in the OSAM organization, pertains to second level overflow segments, and its primary occurrence comes at insertion or additions to data bases. The format is:



where

PTR3 location = location of PTR 3 pointer in Type 2 format

PTR5 = PTR pointer to next second level overflow segment (Type 4 format)

KEY = The parent root key

P = one byte of overhead per segment for physical code of segment type

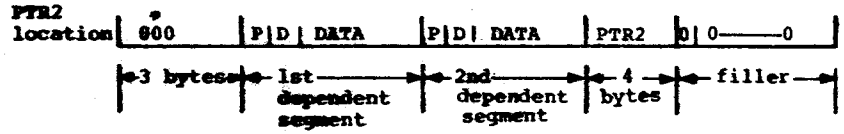
D = one byte of overhead per segment for delete code

DATA = actual data for that segment, including its key

PTR2 = PTR pointer to next dependent segment (Type 5 format) in OSAM, if there is necessity for overflow. Otherwise, all four bytes are binary zeros.

filler = the remaining area not used by data or the PTR pointer is filled with binary zeros

TYPE 5 RECORD FORMAT: The Type 5 logical record Data Language/I format can be contained within the primary and secondary data set groups, is used as a part of the CSAM organization, pertains to all dependent segments, and its occurrence can be at both initial loading and insertions into a data base. The format is:



where

PTR2 location = location of PTR2 pointer from either Type 1, 2, 3, 4, or 5 format

* = three bytes of binary zeros (reserved)

P = one byte of overhead per segment for physical code of segment type

D = one byte of overhead per segment for delete code

DATA = actual data for that segment, including its key

PTR2 = PTR pointer to another dependent segment record of the same type as this one.

filler = The remaining area not used by data, or the PTR pointer is filled with binary zeros

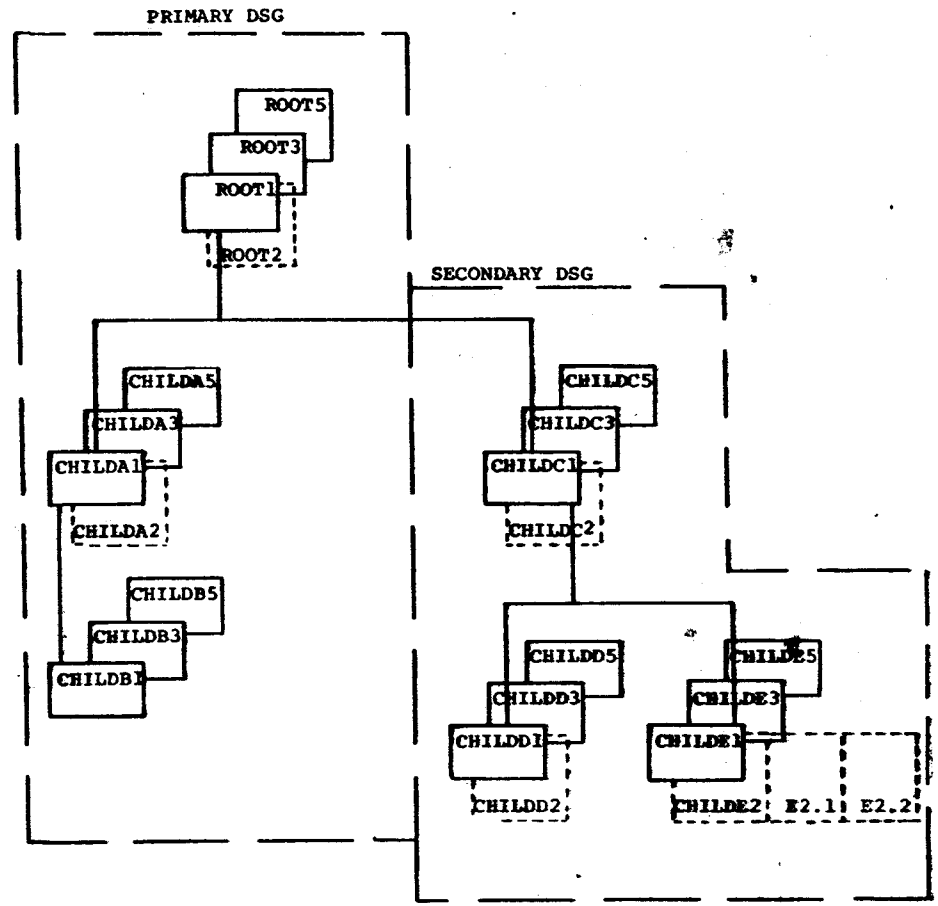


Figure 38. Types of Data Language/I logical records

EXAMPLES OF TYPES OF DATA LANGUAGE/I LOGICAL RECORDS: Figure 36 shows a data base structure that is used to explain, with examples, the different types of Data Language/I logical records. In Figure 38, the children (dependent segments) associated with root 1 are a number with the suffix 1. Such is the case also with roots 3 and 5. Root 2 and its children are noted in the same manner, but with dashes because they were not initially loaded into the data base.

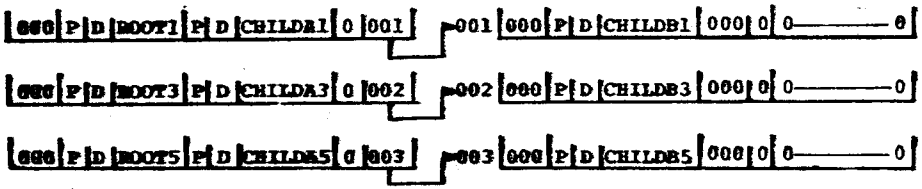
The examples show these combinations:

Type 1 to type 5
Type 2 to type 5
Type 1 to type 3
Type 2 to type 4
Type 4 to type 5
Type 5 to type 5

Note that, for the purposes of example, a short logical record was constructed to force an overflow of OSAM.

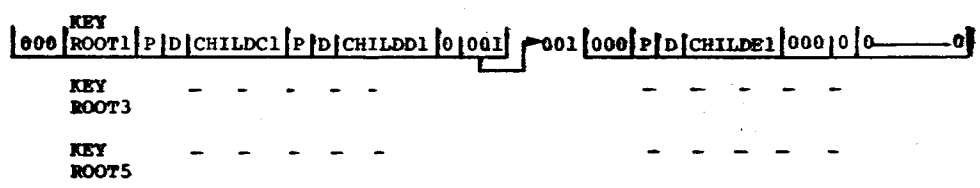
- Example type 1 to type 5

Primary DSG



- Example type 2 to type 5

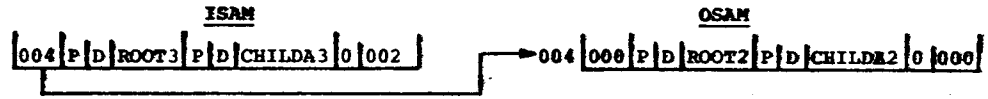
Secondary DSG



- Example type 1 to type 3

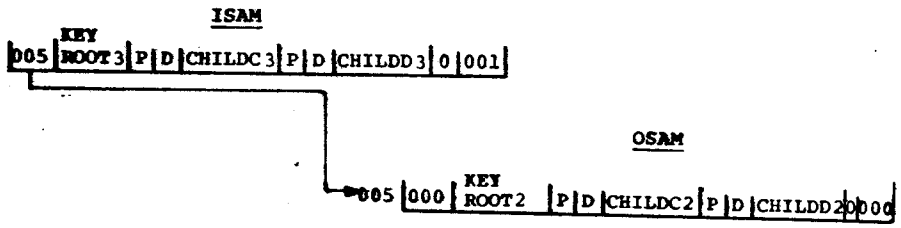
Adding root 2 to data base. See Figure 38.

Primary DSG

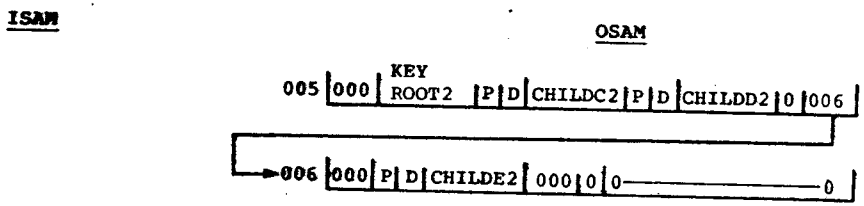


- Example type 2 to type 4

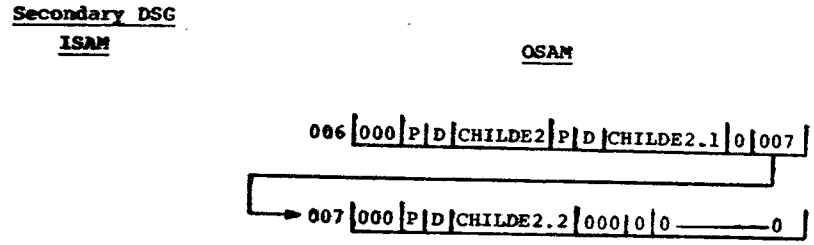
Adding root 2's children to secondary data set group. See Figure 38.



Example type 4 to type 5



- Example type 5 to type 5



Variable Length Data Base Record Processing

The data base creation and processing capabilities of Data Language/I allow an application to define a data base record structure in hierarchical segments. The actual number of segments within a particular data base record may and probably will vary significantly across all data base records within the data base. This creates the need for the ability to handle the physical storage of variable length application logical records or data base records. The degree of variable length capability must not be constrained by the physical attributes (track length) of an input/output device. An application logical record may be a partial track or may exceed a cylinder of direct access device space.

In order to provide this variable length physical storage capability, Data Language/I has adopted the following philosophy. All segments of a data base record may be stored in one physical record or in multiple physical records:

When multiple physical records are required, the first physical record points to the second by relative direct access device address, and the second to the third in a like manner. When the Hierarchical Indexed Sequential organization is used, the first physical record of any data base record is an ISAM logical record (Type 1 record format). Any subsequent physical records for the same data base record are OSAM physical records (Type 5 record format) (Figure 39).

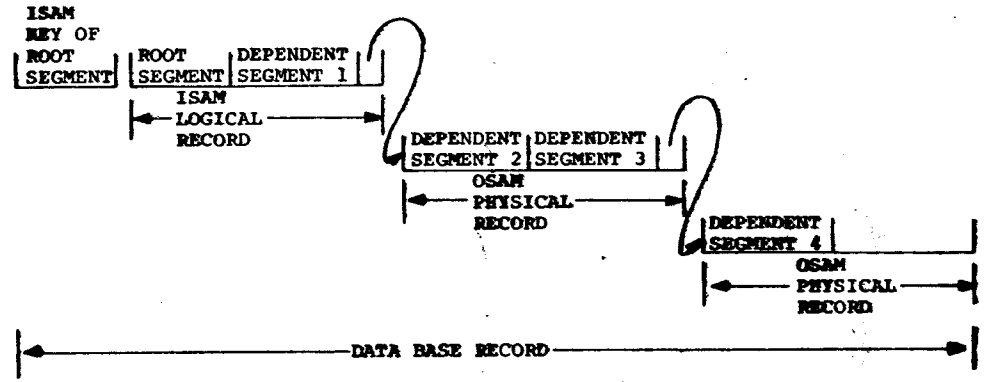


Figure 39. Multiple physical record example

This concept of variable length data base record support is provided through the use of OSAM. When an OSAM data set is opened in the Operating System/360 data management sense, it is used by Data Language/I for both reading and updating in-place existing segments of a data base record or for the addition of new segments of a data base record. An OSAM data set may have as many as sixteen direct access device extents and may exist on up to five direct access device volumes. The physical records of an OSAM data set are the same length as the logical ISAM records of the same data set group within a data base. OSAM does not have a variable length physical record capability with a data set.

The OSAM capability has effectively extended the ability of Operating System/360 ISAM through Data Language/I to create, maintain, and process variable length application logical records.

PSB Generation

The Program Specification Block (PSB) generation utility is an important part of the Data Language/I system. It is the responsibility of the data base designer to perform this generation from the information supplied by the Application Programmer.

There is a close relationship between PSB generation and DBD generation. The data base name must be specified in the PCB control card in the PSB generation. The application program's hierarchical (sensitive) segments must be named in PSB generation, and described in detail in DBD generation.

The details of PSB generation are contained in Section 1. There are no additional parameters that have not already been described.

Note that historical data should be kept for reference along with cross reference information to sensitive data between application programs using the same data base.

DBD Generation

The Data Base Description (DBD) generation is also the responsibility of the data base designer. It is an important factor in building the Data Language/I control block used to describe in detail the structure and storage organization of every data base.

The details of DBD generation are contained in Section 1. However, there are two additional parameters which are a part of DBD generation that are not defined in that section. These are the logical record length (LRECL=) and the blocking factor (BLKFACT=). These parameters are a part of the options of the DMAN control card. The use of the LRECL and BLKFACT have been previously discussed, in the Data Language/I Data Base Space Allocation section of this manual. Both parameters must be specified if either is used. If neither is specified, DBD generation attempts to calculate an optimum in logical record length and block size for the data base.

	DMAN	DD1=,DEVT=,[DD2=],[DLIOP=] [LRECL=,BLKFACT=]

LRECL is a specified number which is less than the maximum length allowed for a particular direct access device track. If the optional parameters are not used, the calculated optimum LRECL will be 1/4, 1/3, or 1/2 track.

The BLKFACT is a number that specifies the number of LRECL's per physical block.

Each data base to be used under Data Language/I definition must be defined by a DBD generation.

The DBD for a data base contains within it the Operating System/360 Data Control Blocks (DCB's) required by Operating System/360 data management. For HISAM, there are four DCB's - QISAM load mode, QISAM scan modes, BISAM read/write update, and OSAM. For HSAM, there are two DCB's - BSAM read and BSAM write. The DCB operands completed by DBD generation for each DCB type are:

```
QISAM LOAD: DSORG=IS,MACRF=PM,RECFM=FB,OPTCD=B,LRECL= ,
   BLKSIZE= ,NTM= ,KEYLEN= ,DDNAME=

QISAM SCAN: DSORG=IS,DDNAME= , MACRF=(SK,GL,PU)

OSAM: DSORG=PS,MACRF=E,LRECL=  , RECFM=F,BLKSIZE= ,
        DDNAME=

BISAM: DSORG=IS,MACRF=(RU,WU) , DDNAME=

BSAM: DSORG=PS,MACRF=(RF,WF) RECFM=U,BLKSIZE= ,
    BUFNO=2 DDNAME=
```

The QISAM load and OSAM DCB's are used to create a HISAM data base in a batch region. The QISAM scan and OSAM DCB's are used to read, update, and add to a HISAM data base in a batch region. The BISAM DCB is not currently being used and is reserved for future expansion of the function.

The output assembly language listing from step 1 of a DBD generation includes an estimate of the cylinder index space and prime data set space for all ISAM data sets with a Data Language/I data base.

Data Base Creation

Initially, the Data Language/I data base must be loaded. Usually the data exists in a machine-readable form acceptable to COBOL, PL/I, or Assembler Language. If so, a user program (COBOL, PL/I, or Assembler Language) must read the data using conventional access methods and then issue Data Language/I insert calls to load the Data Language/I data base. Since the data was not previously organized in a hierarchical fashion, a certain amount of editing may be necessary before doing the Data Language/I load. Also, before the initial load, a DBD GEN and PSB GEN must have been done.

Single and Multiple Data Set Groups

Before creation of the data base, some consideration must be given to using single or multiple data set groups. (See Definition of Multiple Data Set Groups in the first Section of this manual.) The DBD generation controls whether a data base is composed of single or multiple data set groups. The application program is insensitive to the number of data set groups, and, therefore, it is easy to experiment with different combinations until the optimum is found.

The advantages of a single data set group are:

- Only one ISAM index; therefore, less storage space is used.
- On retrievals, using multiple data set groups, the ISAM INDEX for the secondary data set group must be used to access segments in the secondary data set group. This may be more time-consuming than if a single data set group were used. Particularly on sequential retrieval of all dependent segments of a root, multiple data set groups would probably require more time than a single data set group.

- More core will be required for buffers using multiple data set groups.

The advantages of multiple data set groups are:

- The use of multiple data set groups is best indicated when a root has either so many or such long dependent segments that, for most roots, even with a large LRECL, all the dependent segments do not fit into the prime record.
- Under extreme situations, many OSAM blocks may be required to hold all the dependent segments for a single root. When this is the case, it is probably more efficient to use multiple data set groups, thus decreasing the references to OSAM.
- Using a single data set group, Data Language/I must go sequentially from the root through dependent segments to satisfy the call. It may be necessary, therefore, to pass over many dependent segments in order to satisfy a qualified Get Next or Get Unique type call for a second level or lower segment. If all dependent segments of a root are contained in one block, then this is a fast incore scan; but, if the number of dependent segments requires multiple OSAM blocks, considerable time may be necessary to access these blocks sequentially in order to get the one containing the desired segment. If multiple data set groups are used, considerable scanning and possible OSAM access time can be saved because Data Language/I goes directly to the index of the data set group containing the requested segment.

Considerations of HISAM and HSAM

The consideration of whether to use HISAM or HSAM is fairly obvious. First the HSAM restrictions must be considered. Since HSAM is used to reference a sequential data set, data cannot be added, deleted, or replaced in an existing HSAM data set. No hold, delete, replace, or insert calls are valid. Although it is a sequential data set, it can be randomly processed with certain restrictions. The data set will be scanned sequentially either forward or backward to satisfy the call. Therefore, to use an HSAM data set processed in a random fashion may be extremely inefficient. HSAM is not designed for random retrieval. Generally, when random processing is to be done, HISAM should be used. Exceptions to this might occur when the backward searches are very short or when all calls could be satisfied by proceeding forward through a data base.

Data Base Reorganization

Periodically, all data bases should be reorganized. This is necessary to delete those segments from the data base which have been marked 'deleted', and to bring the added root segments which are placed in OSAM physical records during processing back into ISAM prime records. It also decreases the amount of reprocessing necessary to recover from a hard error which requires a data base to be loaded and the update or additions to it to be reprocessed.

The data base may be reorganized either by retrieving all segments from the HISAM data base and loading them to an HSAM data set, either on tape or direct access, or by retrieving them from the HISAM data base and loading them directly to a new HISAM data base. Going directly from HISAM to HISAM would be the most efficient method, but this requires sufficient direct access data space to hold both copies of the data set. After the new one is loaded, the old HISAM packs can be held as the backup copy. Going to HSAM as an intermediate step is not as efficient, but allows reloading of the HISAM data set on the same space previously allocated to it. It also may be convenient for any offline processing required and allows a tape rather than direct access space to hold the backup copy of the data base.

Since the reorganization of a large data base requires a significant amount of time, a pertinent question is "How can I tell when to reorganize my data base?" The answer depends on how volatile the data base is; that is, how many additions are made and how often are these additions being referenced?

Care must be taken to ensure that there is always unused space available in the direct access space allocated to the OSAM data set. The IEHLIST utility program can be used to list the data set control blocks to monitor the amount of unused space available for OSAM additions. OSAM allows for a maximum of 15 extents across 5 volumes.

SYSTEM DEFINITION FOR DATA LANGUAGE/I

System Definition

The System/360 used for the Data Language/I two-stage definition process must be at least a Model 40, with the "F"

assembler and at least 128K storage. The user SVC is placed into (link-edited with) the Operating System nucleus by the system user. Choice of cataloging data sets is the user's.

If Stage 1 was not properly defined, Stage 2 input can be corrected without the necessity for complete regeneration. A system programmer knowledgeable in Data Language/I control block structure can accomplish this function.

System Definition Macro-Instructions

The input to Stage 1 of the Data Language/I system definition is a set of control cards which invoke macro-instructions. These macro-instructions tailor Data Language/I to a particular user environment by creating the control blocks upon which the Data Language/I modules execute. See Figure 40.

	MACRO INSTRUCTION	BATCH DEFINITION
1	DLICTRL	REQUIRED
2	MACLIB	OPTIONAL
3	RESLIB	OPTIONAL
4	PGMLIB	OPTIONAL
5	PSBLIB	OPTIONAL
6	DBDLIB	OPTIONAL
7	PROCLIB	OPTIONAL
8	DLIGEN	REQUIRED

Figure 40. System definition macro-instruction

The following explanation applies to the name field where it appears in the system definition macro-instructions:

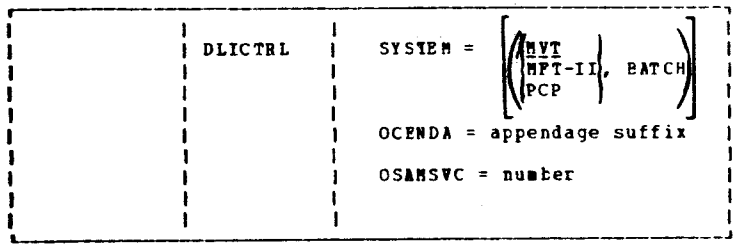
- Name Field:

name

is used in system generation error messages to identify the macro-instruction which produced the error. If no name is entered, the macro-definition supplies sequential identification numbers to the macro-instructions in the same order in which they are presented in the input stream.

DLICTRL Macro

The DLICTRL macro-instruction is used to describe the basic Data Language/I control program options and the Operating System/360 environment under which Data Language/I will operate. The DLICTRL macroinstruction is always required.



- **Operand Field:**

SYSTEM=

specifies whether Data Language/I operates in an Operating System/360 environment with a variable number of tasks (MVT) or a fixed number of tasks (MFT-II). The default value for this key word is 'MVT'. BATCH means only a stand-alone Data Language/I batch system is generated.

OCENDA=

specifies the load module member name to be given the OSAM channel end appendage. This module is moved from the Data Language/I load module library to SYS1.SVCLIB during system definition. The name of this module must start with IGG019. Two additional characters must be appended. These characters may range from WA to Z9. Only the last two characters of

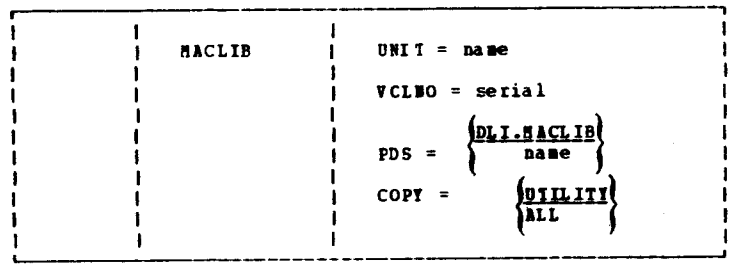
the name should be specified in the macro operand. The default name is OCENDA=Z9 (that is, IGG019Z9).

OSAMSVC=

specifies the user SVC number to be given the OSAM Type 2 SVC. This SVC is used to construct and extend to OSAM data extent blocks (DEB) in system queue space when using MVT. The default SVC number is 255.

MACLIB Macro

The MACLIB macro-instruction designates the library upon which macro-definitions (such as PSB and DBD macro-definitions) output from Stage 2 are placed. If the MACLIB macro-instruction is used, the partitioned data set named in the PDS operand must be allocated and cataloged by the user and exist in the generating system. If the MACLIB statement is omitted, no Data Language/I macro-definitions will be transferred from DLI.GENLIB to any user library, including those required to perform PSB and DBD generation.



- **Operand Field:**

UNIT=

specifies the unit name of the direct access device upon which the macro library PDS is to reside in the generated system. Mandatory entry value must be 2311 or 2314.

VOLNO=

specifies the serial number of the volume that is to contain Data Language/I macro-definitions in the generated system. Mandatory entry.

PDS=

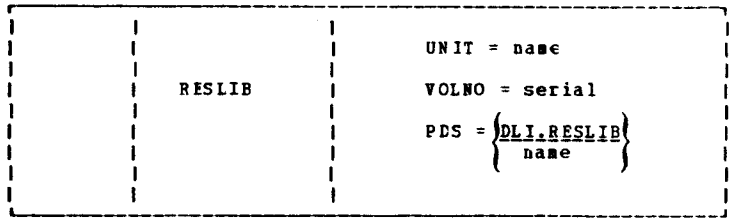
is the name of the macro-definition library upon which the Data Language/I macro-definition will reside in the generated system. If no PDS name is provided, Stage 2 assumes the PDS name to be DLI.MACLIB.

COPY=

specifies which macro-definitions are to be transferred to the PDS specified in the PDS operand. If omitted, only those required to perform PSB and DBD generation are copied. If ALL is specified, all macro-definitions in DLI.GENLIB are copied. Preallocation space requirements will be explained later.

RESLIB Macro

The RESLIB statement defines the PDS in the generating and generated system upon which all Data Language/I load modules are placed during Stage 2. It must be a preallocated, cataloged data set in the generating system. It may be SYS1.LINKLIB.



- Operand Field:

UNIT=

specifies the unit name of the direct access device upon which the macro library PDS is to reside in the generated system. Mandatory entry value must be 2311 or 2314.

VOLNO=

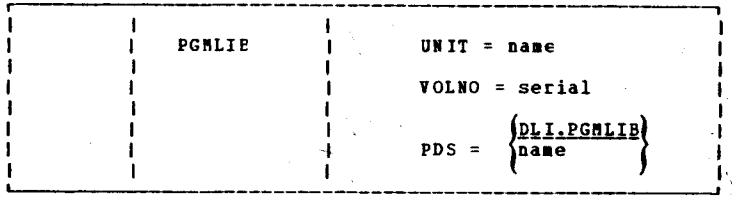
specifies the serial number of the volume that is to contain Data Language/I macro-definitions in the generated system. Mandatory entry.

PDS=

specifies the dsname of the library in the generated system upon which the Data Language/I load module library is placed. If this statement is omitted, it is assumed that the PDS named DLI.RESLIB is cataloged and preallocated in the generated system. Allocation space requirements are supplied later in this manual.

PGMLIB Macro

The PGMLIB macro-instruction designates the library upon which all executable application programs reside.



- Operand Field:

UNIT=

specifies the unit name of the direct access device upon which the application program library is to reside in the generated system. If this operand is omitted, the VOLNO operand must also be omitted. System definition then assumes the data set is catalcged on the generated system.

VOLNO=

specifies the serial number of the volume that is to contain Data Language/I application programs in the generated system. If this operand is omitted, the UNIT operand must also be omitted. See UNIT operand above.

PDS =

is the name of the program library. If the PGMLIB statement is made and the PDS operand is omitted, the default name of the partitioned data set is DLI.PGMLIB. Allocation guidelines are supplied later in this manual.

PSBLIB Macro

The PSBLIB macro-instruction designates the library upon which the output from the Data Language/I PSB generation utility resides.

```
PSBLIB    UNIT = name,
          VOLNO = serial,
          PDS = {DLI.PSBLIB}
                {name      }
```


- Operand Field:

UNIT=

specifies the unit name of the direct access device upon which the PSB library is to reside in the generated system.

VOLNO=

specifies the serial number of the volume that is to contain the PSB library.

PDS=

is the name of the PSB library. If the PSBLIB statement is made and the PDS operand is omitted, the default name of the partitioned data set is DLI.PSBLIB. The library need not be allocated in the generated system. Allocation guidelines are supplied later in this manual.

DBDLIB Macro

The DBDLIB macro-instruction designates the library upon which output from the Data Language/I DBD generation utility resides.

```
DBDLIB    UNIT = name,
          VOLNO = serial,
          PDS = {DLI.DBDLIB}
                {name      }
```

- Operand Field:

UNIT=

specifies the unit name of the direct access device upon which the DBD library is to reside in the generated system.

VOLNO=

specifies the serial number of the volume that is to contain the DBD library.

PDS=

is the name of the DBD library. If the DBDLIB statement is made and the PDS operand is omitted, the default name of the partitioned data set is DLI.DBDLIB. The library need not be allocated in the generated system. Allocation guidelines are supplied later in this manual.

PROCLIB Macro

The PROCLIB macro-instruction designates the library upon which procedure output from Stage 2 is placed. If the PROCLIB macro-instruction is used, the PDS name specified must exist in the generated system. If the PROCLIB statement is omitted, no user procedures are generated. If the statement is included, and if all conditions stated in other generation macros which affect procedure generation are satisfied, the following procedures are generated.

- PSBGEN generate (assemble) PSB and link to appropriate library
- DBDGEN generate (assemble) DBD and link to appropriate library
- DII execute stand-alone processing region
- DLICOBGO execute COBCL compile and go batch processing
- DLIPLIGO execute PL/I compile and go batch processing
- DLICOBOL compile and link COBOL to appropriate program library
- DLIPLI compile and link PL/I to appropriate program library

PROCLIB	UNIT = name	
	VOLNO = serial	
	PDS =	{DLI.PROCLIB / name}

- Operand Field:

UNIT=

specifies the unit name of the direct access device upon which the macro library PDS is to reside in the generated system. If this operand is omitted, the VOLNO operand must also be omitted. System definition then assumes the data set is cataloged on the generated system.

VOLNO=

specifies the serial number of the volume that is to contain Data Language/I procedures in the generated system. If this operand is omitted, the UNIT operand must also be omitted. See UNIT operand above.

PDS=

is the name of the procedure library upon which the Data Language/I procedures reside in the generated system. If no PDS name is provided, Stage 2 assumes the preallocated PDS name to be DLI.PROCLIB. Allocation requirements are supplied later in this manual.

DLIGEN Macro

The DLIGEN macro-instruction is used to specify the data sets, volumes, and I/O devices required for the definition process, the system definition output options.

The DLIGEN macro-instruction must be the last macro-instruction in the Stage 1 input stream. It must be followed immediately by an assembler END statement.

```
| DLIGEN  |  UT1SDS = dsname,
|         |
|         |  ASMPRT = {OFF}
|         |           {ON }
|         |
|         |  LEPRT  = (option,    option   )
```

• **Operand Field:**

UT1SDS=

specifies name of a utility data set to be used during Stage 2 of system definition.

ASMPRT=

specifies whether assembly listings are to be procured for the modules assembled during system definition. ON specifies that assembly listings are to be generated; OFF, that assembly listings are not to be generated.

LEPRT=

specifies linkage editor print options as one or two of the following values. The values included in braces are mutually exclusive.

Value	Print Option
LIST	List of control statements in card-image format
MAP	Module map
XREF	Cross-reference table (XREF includes the MAP option)

If this parameter is omitted, only linkage editor error messages, if any, are printed. For a more

detailed description of these options, see the publication IBM System/360 Operating System: Linkage Editor, Form C28-6538.

Supervisor Call Routine

The Data Language/I system utilizes a supervisor call (SVC) routine. This routine is used by OSAM to create its multivolume data extent block (DEB). System definition creates this routine with a user-defined SVC number. The user must link-edit this routine with the Operating System/360 nucleus. How to perform the link-edit is explained later in this section.

OSAM Channel End Appendage

OSAM requires a channel end appendage module. This module is created as a load module by Data Language/I system definition. The name of the module can be specified by the user. The created module is placed in DLI.RESLIB. The user must move this module to his SYS1.SVCLIB data set. How to perform the copy operation is explained later in this section.

System Libraries and Procedures

If a PROCLIB macro-instruction is presented in the Stage 1 input of Data Language/I system definition, certain procedures are created to be placed in the library specified. These procedures are complete only to the extent of the information made available through the optional library macro-instructions. For example, if the user does not specify a name for the program specification block library, the default DSNAME value of DLI.PSBLIB is used in the generated procedure. This could mean that the created procedures are not executable in the Data Language/I operating environment. Other procedures have defaults as specified in the system definition macro-instruction. Created procedures should be examined carefully to determine if the desired JCL has in fact been correctly specified. If all optional library macro-statements are included as input to Stage 1 of Data Language/I system definition, the following procedures are created:

Procedure Library

Member Name	Description
PSBGEN	A two-step assemble and link-edit procedure to produce program specification blocks
DBDGEN	A two-step assemble and link-edit procedure to produce data base definition blocks
DLICOBCL	A two-step compile and link-edit procedure for Data Language/I applications written in COBOL
DLIPLI	A two-step compile and link-edit procedure for Data Language/I applications written in PL/I
DLI	A one-step execution procedure for stand-alone Data Language/I Type 3 processing region
DLICOBGO	A three-step compile, link-edit, and go procedure combining the procedures DLICOBOL and DLI
DLIPLIGO	A three-step compile, link-edit, and go procedure combining the procedures DLIPLI and DLI

Note that these procedures are written to accommodate the type 1 programming system, either MVT or MFT-II. Also, volume serial and unit will appear if specified library is not cataloged.

Specific examples of the default procedures follow:

```
MEMBER NAME PSBGEN
./       ADD   NAME=PSBGEN
./       NUMBER NEW1=10,INCR=10
//       PROC   MBR=TEMPNAME
//C      EXEC   PGM=IEUASM,PARM='LOAD,NODECK',REGION=92K
//SYSLIB   DD   DSNAME=DLI.MACLIB,DISP=SHR
//         DD   DSNAME=SYS1.MACLIB,DISP=SHR
//SYSGO    DD   UNIT=SYSDA,DISP=(,PASS),DCB=(BLKSIZE=400,          X
//              RECFM=FB,LRECL=80),SPACE=(80,(100,100),RLSE)
//SYSPRINT DD   SYSOUT=A,DCB=(LRECL=121,RECFM=FBM,BLKSIZE=605,    X
//              SPACE=(121,(500,500),RLSE,,ROUND)
//SYSUT1   DD   UNIT=SYSDA,DISP=(,DELETE),SPACE=(1700,(100,50))
//SYSUT2   DD   UNIT=SYSDA,DISP=(,DELETE),SPACE=(1700,(100,50))
//SYSUT3   DD   UNIT=(SYSDA,SEP=(SYSLIB,SYSUT1,SYSUT2)),         X
//              SPACE=(1700,(100,50))
//L      EXEC   PGM=DFSILNK0,PARM='XREF,LIST',COND=(0,LT,C),      X
//              REGION=100K
//SYSLIN   DD   DSNAME=*.C.SYSGO,DISP=(OLD,DELETE)
//         DD   DDNAME=SYSIN
//SYSPRINT DD   SYSOUT=A,DCB=(LRECL=121,RECFM=FBA,BLKSIZE=605),   X
//              SPACE=(121,(100,100),RLSE)
//SYSLMOD  DD   DSNAME=DLI.PSBLIB(&MBR),DISP=OLD
//SYSUT1   DD   UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),                X
//              DISP=(,DELETE),SPACE=(1024),(100,10),RLSE)
```

MEMBER NAME DBDGEN

```
./       ADD     NAME=DBDGEN
./       NUMBER NEW1=10,INCR=10
//       PROC    MBR=TEMPNAME
//C      EXEC    PGM=IEUASM,PARM='LOAD,NODECK',REGION=92K
//SYSLIB     DD  DSNAME=DLI.MACLIB,DISP=SHR
//           DD  DSNAME=SYS1.MACLIB,DISP=SHR
//SYSGO      DD  UNIT=SYSDA,DISP=(,PASS),DCB=(BLKSIZE=400),       X
//             RECFM=FB,LRECL=80),SPACE=(80,(100,100),RLSE)
//SYSPRINT   DD  SYSOUT=A,DCB=(LRECL=121,RECFM=FBM,BLKSIZE=605,  X
//             SPACE=(121,(500,500),RLSE,,ROUND)
//SYSUT1     DD  UNIT=SYSDA,DISP=(,DELETE),SPACE=(1700,(100,50))
//SYSUT2     DD  UNIT=SYSDA,DISP=(,DELETE),SPACE=(1700,(100,50))
//SYSUT3     DD  UNIT=(SYSDA,SEP=(SYSLIB,SYSUT1,SYSUT2)),        X
//             SPACE=(1700,(100,50))
//L      EXEC    PGM=DFISLNK0,PARM='XREF,LIST',COND=(0,LT,C),    X
//             REGION=100K
//SYSLIN     DD  DSNAME=*.C.SYSGO,DISP=(OLD,DELETE)
//           DD  DDNAME=SYSIN

//SYSPRINT   DD  SYSOUT=A,DCB=(LRECL=121,RECFM=FBA,BLKSIZE=605), X
//             SPACE=(121,(100,100),RLSE)
//SYSLMOD    DD  DSNAME=DLI.DBDLIB(&MBR),DISP=SHR
//SYSUT1     DD  UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),               X
//             DISP=(,DELETE),SPACE=(1024,(100,10),RLSE)
```


MEMBER NAME DLICOBOL

```
//       PROC    MBR=,PAGES=60
//C EXEC PGM=IEQCFL00,PARM='SIZE=110000,LINECNT=50',REGION=126K
//SYSLIN DD DSNAME=&&LIN,DISP=(MOD,PASS),UNIT=SYSDA,             X
//             DCB=(LRECL=80),RECFM=FB,BLKSIZE=400),             X
//             SPACE=(CYL,(4,1),RLSE)
//SYSPRINT DD   SYSOUT=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=605), X
//             SPACE=(605,(&PAGES.0,&PAGES),RLSE,,ROUND)
//SYSUT1 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//SYSUT2 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//SYSUT3 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//SYSUT4 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//L EXEC PGM=DFSILNK0,REGION=100K,PARM='XREF,LIST,LET',          X
//             COND=(4,LT,C)
//SYSLIB DD DSNAME=SYS1.CCBLIB,DISP=SHR
// DD DSNAME=SYS1.PL1LIB,DISP=SHR
//SYSOBJ DD DSNAME=DLI.RESLIB,DISP=SHR  NOTE
//SYSLIN DD DSNAME=&&LIN,DISP=(OLD,DELETE)
//          DD  DSNAME=DLI.PROCLIB(DLITCBL),DISP=SHR
// DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=DLI.PGMLIB(&MBR),DISP=SHR
//SYSPRINT DD  SYSOUT=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=605),   X
//             SPACE=(605),&PAGES.0,RLSE),,ROUND)
//SYSUT1 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
```

Assumes

1. User supplies source data from SYSIN
2. Output Class A
3. MBR=NAME, when name is load module name for program
4. SYSDA is generic device name

NOTE ASSUMES RESLIB CATALOGED.

MEMBER NAME DLIPLI

```
//       PROC    MBR=,PAGES=50
//C      EXEC    PGM=IEMAA,PARM='XREF,ATR,LOAD,NODECK,NOMACRO,    X
//               CET=1',REGION=114K
//SYSUT1     DD  UNIT=SYSDA,SPACE=(1024,(60,60),RLSE,,ROUND),     X
//               DCB=BLKSIZE=1024,DISP=(NEW,PASS)
//SYSPRINT DD    SYSOUT=A,DCB=(LRECL=121,BLKSIZE=605,RECFM=FBA),  X
//               SPACE=(605,(&PAGES.0,&PAGES),RLSE)
//SYSLIN     DD  UNIT=SYSDA,SPACE=(80,(250,80),RLSE),             X
//               DCB=BLKSIZE=80,DISP=(NEW,PASS)
//L      EXEC    PGM=DFSILNK0,PARM='XREF,LIST,LET',COND=(4,LT,C),X
//               REGION=100K
//SYSLIB     DD  DSNAME=SYS1.PL1LIB,DISP=SHR
//           DD  DSNAME=SYS1.COBLIB,DISP=SHR
//SYSLIN     DD  DSNAME=*.C.SYSLIN,DISP=(OLD,DELETE)
//           DD  DSNAME=DLI.PROCLIB(DLITPLI),DISP=SHR
//           DD  DDNAME=SYSIN
//SYSLMOD    DD  DSNAME=DLI.PGMLIB(&MBR),DISP=SHR
//SYSPRINT   DD  SYSOUT=A,DCB=(LRECL=121,BLKSIZE=605,RECFM=FBA),  X
//               SPACE=(605,(&PAGES.0,&PAGES),RLSE)
//SYSOBJ     DD  DSNAME=DLI.RESLIB,DISP=SHR
//SYSUT1     DD  UNIT=SYSDA,DISP=(NEW,DELETE),                    X
//               SPACE=(CYL,(5,1),RLSE)
```

Same assumptions as DLICOBOL

MEMBER NAME DLI

```
//       PROC    PSB=TEMPNAME
//G          EXEC PGM=DFSIRC00,PARM='3,&PSB',REGION=120K
//DLI        DD  DSNAME=DLI.PSBLIB,DISP=SHR
//           DD  DSNAME=DLI.DBDLIB,DISP=SHR
//SYSUDUMP   DD  SYSOUT=A,SPACE=(605,(500,500),RLSE,,RCUND),      X
//               DCB=(RECFM=FBA,LRECL=121,BLKSIZE=605)
```


MEMBER NAME DLICOBGO

```
//       PROC    MBR=,PAGES=60
//C      EXEC    PGM=IEQCB100,                                    X
//               PARM='LINECNT=50,SIZE=110000',REGION=126K
//SYSIN      DD  DSNAME=&&LIN,DISP=(MOD,PASS),UNIT=SYSDA,         X
//               DCB=(L,RECL=80,RECFM=FB,BLKSIZE=400),            X
//               SPACE=(CYL,(4,1),RLSE)
//SYSPRINT DD    SYSOUT=A,DCB=(LRECL=121,RECFM=FBA,BLKSIZE=605),  X
//               SPACE=(605,(&PAGES.0,&PAGES),RLSE,,ROUND)
//SYSUT1 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//SYSUT2 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//SYSUT3 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//SYSUT4 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//L EXEC PGM=DFSILNK0,REGION=100K,PARM='XREF,LIST,LET',           X
//               COND=(4,LT,C)
//SYSLIB DD DSNAME=SYS1,COBLIB,DISP=SHR
//           DD  DSNAME=SYS1.PL1LIB,DISP=SHR
//SYSOBJ DD DSNAME=DLI.RESLIB,DISP=SHR  NOTE 1
//SYSLIN DD DSNAME=&&LIN,DISP=(OLD,DELETE)
//           DD  DSNAME=DLI.PROCLIB(DLITCBL),DISP=SHR
//           DD  DDNAME=SYSIN
//SYSLMOD    DD  DSNAME=DLI.PGMLIB(&MBR),DISP=SHR
//SYSPRINT   DD  SYSOUT=A,DCB=(LRECL=121,RECFM=FBA,BLKSIZE=605),  X
//               SPACE=(605,&PAGES.0,RLSE,,RCUND)
//SYSUT1 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(CYL,(10,1),RLSE)
//G  EXEC  PGM=DFSIRC00,PARM='3,&MBR',REGION=150K,TIME=2,          X
//               COND=(0,LT)
//DLI        DD  DSNAME=DLI.PSBLIB,DISP=SHR
//           DD  DSNAME=DLI.DBDLIB,DISP=SHR
//SYSOUT DD SYSOUT=A,SPACE=(CYL,(1,1)),DCB=(LRECL=133,RECFM=GA)
//SYSUDUMP DD SYSOUT=A,DCB=(LRECL=121,RECFM=FBA,BLKSIZE=3025),    X
//               SPACE=(3025,(200,100),RLSE,,RCUND)
```

1. ASSUMES USER SUPPLIES SOURCE FROM SYSIN
2. OUTPUT CLASS EQUALS A
3. MBR=NAME, WHERE NAME IS LOAD MODULE NAME FOR PROGRAM.
4. USER MUST SUPPLY JOBLIB CARDS FOR DLI.RESLIB AND DLI.PROGLIB WITH DISP=(SHR,PASS)
5. SYSDA IS GENERIC DEVICE NAME

NOTE 1 ASSUMES RESLIB CATALOGED

MEMBER NAME DLIPLIGO

```
//        PROC    MBR=NAME,PAGES=50
//C       EXEC    PGM=IEMAA,PARM='XREF,ATR,LOAD,NODECK,NOMACRO,    X
//                OPT=1',REGION=114K
//SYSUT1     DD   UNIT=SYSDA,SPACE=(1024,(60,60),RLSE,,ROUND),     X
//                DCB=BLKSIZE=1024,DISP=(NEW,PASS)
//SYSPRINT   DD   SYSOUT=A,DCB=(LRECL=121,BLKSIZE=605,RECFM,FBA),  X
//                SPACE=(605,(&PAGES.0,&PAGES),RLSE)
//SYSLIN DD       UNIT=SYSDA,SPACE=(80,(250,80),RLSE),             X
//                DCB=BLKSIZE=80,DISP=(NEW,PASS)
//L       EXEC    PGM=DFSILNK0,PARM='XREF,LIST,LET',COND=(4,LT,C),X
//                REGION=100K
//SYSLIB     DD   DSNAME=SYS1.PL1LIB,DISP=SHR
//           DD   DSNAME=SYS1.CCBLIB,DISP=SHR
//SYSLIN     DD   DSNAME=*.C.SYSLIN,DISP=(OLD,DELETE)
//           DD   DSNAME=DLI.PROCLIB(DLITPLI),DISP=SHR
//       DD       DDNAME=SYSIN
//SYSLMOD DD      DSNAME=DLI.PGMLIB(&MBR),DISP=SHR
//SYSPRINT DD     SYSOUT=A,DCB=(LRECL=121,BLKSIZE=605,RECFM=FBA),  X
//                SPACE=(605,(&PAGES.0,&PAGES),RLSE)
//SYSOBJ     DD   DSNAME=DLI.RESLIB,DISP=SHR
//SYSUT1     DD   UNIT=SYSDA,DISP=(NEW,DELETE),                    X
//                SPACE=(CYL,(5,1),RLSE)
//G       EXEC    PGM=DFSIRC00,PARM='3,&MBR',COND=(4,LT),          X
//                TIME=5,REGION=150K
//DLI        DD   DSNAME=DLI.PSBLIB,DISP=SHR
//           DD   DSNAME=DLI.DBDLIB,DISP=SHR
//SYSPRINT DD     SYSOUT=A,DCB=(LRECL=121,BLKSIZE=605,RECFM=FBA),  X
//                SPACE=(605,(500,500),RLSE,,ROUND)
//SYSUDUMP   DD   SYSOUT=A,DCB=(LRECL=121,BLKSIZE=605,RECFM=FBA),  X
//                SPACE=(605,(500,500),RLSE,,ROUND)
```

Same assumptions as DLICOBGO

LIBRARIES: Space allocation for MACRO, PSB, DBD, PROGRAM, PROCEDURE, and RESLIB libraries is dependent upon user requirements. Space requirements for user libraries of programs, program specification blocks, and data base definition blocks will depend entirely upon the user's operating environment. Some examples may be useful:

- DBD Library - Each DBD (one per data base) requires approximately 1500 to 2500 bytes of direct access storage. Exact requirements depend upon number of data set groups, segments, fields, and hierarchical levels.

- PSB Library - Each PSB (one per program) requires approximately 250 to 500 bytes of direct access storage. Exact requirements depend upon number of data bases (PSB) used in PSB and number of sensitive segments.
- PROCLIB Library - About 10 tracks (2314) of space are required.
- RESLIB Library - About 20 cylinders of 2314 space are required.
- MACLIB Library - About 10 cylinders of 2314 space are required for ALL macro-instructions. About one cylinder of PSBGEN and DBDGEN macro-instructions only.
- PGMLIB Library - This contains application programs

Operating System/360 Link Pack Modules

Many of the Data Language/I modules, the OSAM modules, and the BISAM modules used can be placed in the Operating System/360 MFT-II or MVT link pack area. The following module list indicates those modules whose placement in link pack is recommended. The modules to be included in link pack must previously exist in either the SYS1.SVCLIB or the SYS1.LINKLIB data set.

From SYS1.LINKLIB

Module name	Module definition
DFSIRC00	IMS/360 Region Controller
DFSIDLR0	Data Language/I HISAM Retrieve
DFSIDLH0	Data Language/I HSAM
DFSIDLI0	Data Language/I HISAM Insert
DFSIDLD0	Data Language/I HISAM Delete/Replace
DFSIOS20	OSAM Read/Write
DFSIOS30	OSAM Check
DFSIOS60	OSAM Open/Close
DFSIISM0	Data Language/I ISAM Simulator
DFSIWKN0	Data Language/I Write Key New Simulator

From SYS1.SVCLIB

Module name	Module definition
IGG019Z8*	OSAM Channel End Appendage
IGG019GX	BISAM Asynchronous Read/Write
IGG019G9	BISAM Appendage with Write Check
IGG019JV	BISAM Non-Privileged Macro-Time Read/Write
IGG019J7	BISAM Privileged Macro-Time Read/Write

* The last two characters of this module names is are determined by the user during system definition.

IMPLEMENTATION

DATA BASE

DBD Generation

It is assumed that during Data Language/I system definition, the user did not specify a name for the DBD library; the default name is therefore used in the generated DBD procedure. The generated procedure is DBDGEN, which is a two-step assemble and link-edit procedure to produce data base definition blocks. See the example of the DBDGEN procedure.

Use this procedure, DBDGEN, when running each DBD generation. The JCL cards are:

```
//DBDGEN JOB MSGLEVEL=1
//      EXEC    DBDGEN,MBR=
//C.SYSIN  DD  *
      { DBD    }
      { DMAN   }
      { SEGM   }
      { FLDX   }   DBD generation control cards
      { FLD    }
      { DBDGEN }
      { FINISH }
      { END    }
/*
```

where

key word operand MBR=

is the name for the DBD to be generated

Description of Data Language/I Segment Insertion

Data Language/I segment insert logic for hierarchical indexed data bases is designed to handle (1) root segment insertion, and (2) dependent segment insertion. This section describes how Data Language/I implements segment insertion.

ROOT SEGMENT INSERTION: The logic for root segment insertion also includes the handling of second level segment insertion on secondary data set groups. If the segment to be inserted is a root, Data Language/I proceeds to place the segment into OSAM and chain from ISAM of the primary data set group. For second level segment insertion into secondary data set groups, the insert module calls upon the retrieve module for physical positioning within the proper data set and buffer as well as verification of the presence of a root segment. If a root exists for the second level segment to be inserted, the retrieve module attempts to find an associated record on the secondary data set to prohibit duplicate dependent segments. When no record is found on a secondary data set, Data Language/I builds a new second level segment and places the segment into OSAM with a chain from the ISAM data set of the secondary data set group.

The insertion operation in Data Language/I is performed by first searching the appropriate ISAM data set for a root segment key equal to or greater than the root segment or second level segment key in a secondary data set group being inserted. If a segment is found with key equal, and the segment with a delete bit turned on, Data Language/I inserts the new segment in place of the old segment. If a segment is found with key equal, and the segment with a delete bit is turned off, the insertion is rejected with the appropriate status code.

For a key-high condition, Data Language/I examines the first three bytes in the key-high record. This three-byte area signifies whether or not there is a chain of additional roots with keys less than the current key-high record. If the three bytes equal binary zero, Data Language/I inserts the next available OSAM block number into that three-byte area, writes the new root record out into OSAM, and, finally, writes back the ISAM key-high record, to include a pointer to the new OSAM block.

When the three-byte value is nonzero, Data Language/I reads the OSAM block which the three-byte value addresses and compares the newly read root key against the root key of the segment being inserted. If the newly read key is greater than the insert key, Data Language/I backs up to the previous record, moves its three-byte pointer into the new segment record, and writes out the new OSAM record. After completion of the successful OSAM write, Data Language/I updates the previous record by placing the newly written OSAM block number into the three-byte area and performing a write-back operation. If the key field of the segment in the OSAM block read is less than the key field of the segment to be inserted, the first three bytes of the OSAM block are tested equal or not equal to

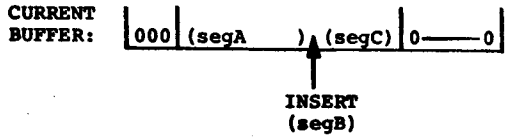
zero. If zero, the new segment to be inserted is written into the next available OSAM block and the last-read OSAM block is updated to point to the new OSAM block. If nonzero, the next OSAM block chained to is read and the key field of the segment in this block is tested against the key field of the segment inserted. At this point, the insert process will iterate through one of the above situations.

DEPENDENT SEGMENT INSERTION: When a dependent segment is to be inserted into an existing data base record, the insert module of Data Language/I calls the retrieve module for the positioning action. After positioning action is complete, the insert module branches to one of four possible conditions, depending on how much slack space is available in the logical record.

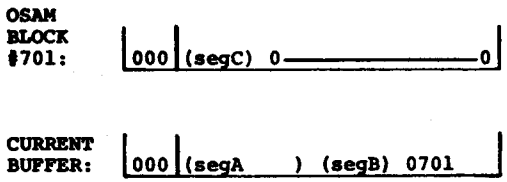
CONDITION 1: If there is enough space available to insert the new dependent segment into the existing physical record, this Condition shifts any data that may exist to the right of the new dependent segment insert position. The new segment is then inserted and a "buffer pending" flag is turned on.

CONDITION 2: When the amount of slack is less than insert segment length, a test is made to determine if the new segment length plus an OSAM block pointer will fit into the area between the insert position and the end of the logical record. If so, any existing shift data is moved to a work area and is immediately written out to the OSAM data set. The new segment and just-written OSAM block number are moved into the current buffer, and the pending-flag is turned on.

If the new segment plus the OSAM pointer do not fit, another test is required to determine if enough space (four bytes) is available to hold an OSAM block pointer only. A "YES" is handled by Condition 3, and a "NO" by Condition 4.

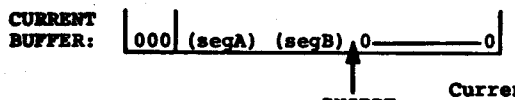


Current buffer before insert

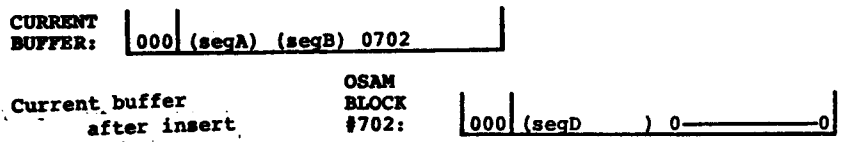


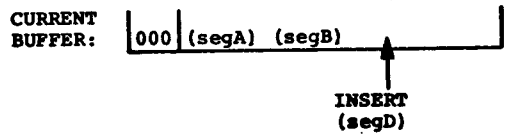
Current buffer after insert

CONDITION 3: This Condition places any new segment and any existing shifted data into a work area, and immediately writes out a new OSAM record. The just-written block number is moved into the current buffer, and the pending-flag is turned on.

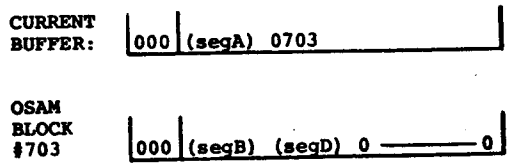


Current buffer before insert will not hold new segment, and slack is greater than 4 bytes





Current buffer before insert has less than 4 bytes



Current buffer after insert

CONDITION 4: When less than four bytes of slack remain, the segment prior to insert position must be extracted to allow room for an OSAM block pointer. The previous segment, new segment, and any shift data are moved to a work area and are immediately written out to the OSAM data set. This newly written block number is then moved into the previous segment position, and the pending-flag is turned on.

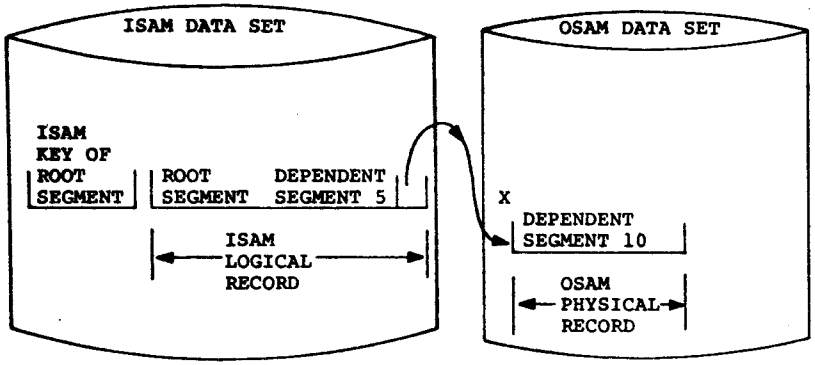


Figure 41. Data base prior to segment insertion

Data Base Integrity Through the Use of OSAM

The modifications made to a Data Language/I data base by the Delete, Replace, and Insert functions create a need for internal capabilities in Data Language/I to attempt to ensure the integrity of a data base. The most complex Data Language/I input/output function of the three is Insert. Whenever a segment (root or dependent) is added to a Data Language/I data base of the Hierarchical Indexed Sequential organization, a new physical record may have to be generated. The OSAM data set(s) of that data base is used for all segment insertion. The Write-Key-New capability of ISAM is never utilized. The following diagrams illustrate a dependent segment insertion and are provided in the sequence of channel program operations. (See Figures 41, 42, and 43.)

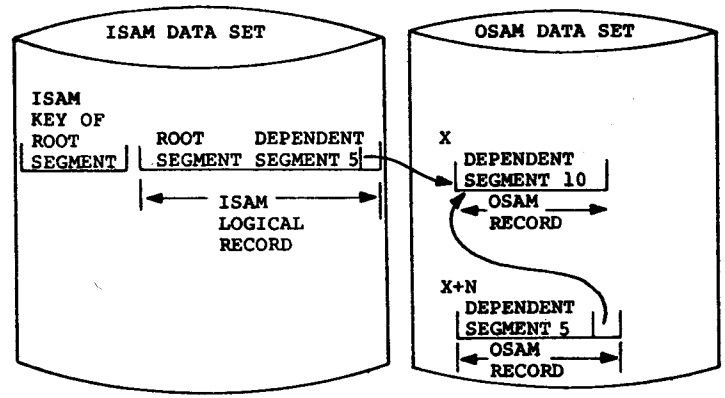


Figure 42. After channel program write and check of new OSAM physical record

The first OSAM physical record (prior to segment insertion) for the data base record is at direct access device address X. OSAM space within a data set is allocated sequentially. Assume that the next allocatable OSAM space is X+N. Assume that a dependent segment number 1 is to be inserted between the root segment and dependent segment number 5. There is no space available in the ISAM record to insert the additional segment and the next available OSAM record address is X+N. A channel program reads the ISAM record and Data Language/I finds that no space exists in the ISAM. Data Language/I also recognizes the existing direct access pointer to record X. Physical location X+N in the OSAM data set is allocated for the Insert. An OSAM record with dependent segment 5 is written at X+N and checked with a channel program (Figure 42).

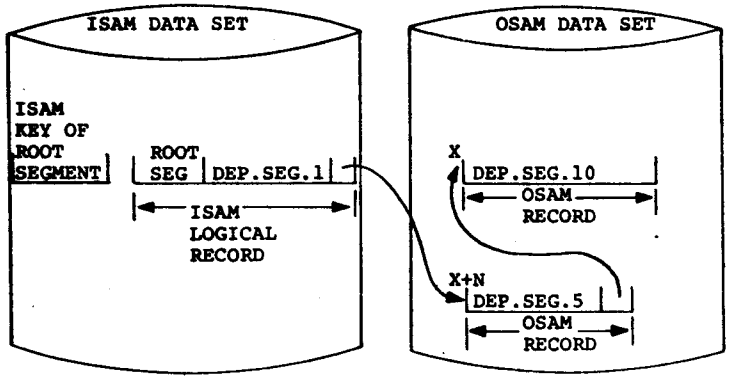


Figure 43. Data base after insertion

The OSAM record has a physical direct access address pointer to record X. Once the record at location X+N is successfully written, the ISAM record is updated, with the root segment and dependent segment 1 unchanged, using a channel program. The OSAM record pointer in the ISAM record is changed to X+N.

In order to maintain the integrity of the data base, the sequence of channel programs is important. The user should try an alternate sequence and consider the possibility of a system failure after start but prior to termination of the Insert operation.

Another critical consideration in the use of a sequentially allocated direct access space such as ISAM overflow areas, the Sequential Access Methods, and OSAM is the proper maintenance of the next allocatable direct access device address. All of the above access methods use a field in the data set control block (DSCB) to maintain this address while a data set is closed. When a data set is opened, this information is placed

in a field in the data control block (DCB) in core storage. As records are added to the data set, the field in the DCB is appropriately updated. When the data set is closed, the DCB field is used to update the DSCB with the new allocatable address. If the system is lost after record adds and the data set is not closed, the DSCB field is not updated. Positioning in the data set area is lost.

A special capability has been added to OSAM to alleviate this problem. Every time an OSAM record is added to a data set, a file mark (Count Field with Key Field and Data Field lengths zero) is written. When another new record is written, the file mark is overlayed and a new file mark is written after the added record. If the system is lost and the OSAM data set is not closed, the DSCB allocation pointer is not updated. If the data set is closed correctly, the DSCB is updated from the DCB. When the OSAM data set is subsequently opened, the next allocatable address from the DSCB is used for a record read. If a Unit Exception indication is received, the positioning is at a file mark and the data set is assumed to have been previously closed correctly. If a Unit Exception indication is not received, the OSAM open routine sequential reads records until a Unit Exception is received. This address then represents the proper positioning in the data set.

PSB Generation

As previously stated (under the section System Libraries and Procedures), the default name of the program specification block (PSB) library is DLI.PSBLIB. This library name is used in the generated procedure, PSBGEN. The procedure PSBGEN is a two-step assemble and link-edit procedure to produce each PSB.

Use the procedure, PSBGEN, when running the different PSB generations. The JCL cards are:

```
//PSBGEN   JOB MSGLEVEL=1
//        EXEC    PSBGEN,MBR=
//C.SYSIN  DD  *
```

```
PCB
SEGSEG
PSBGEN
END
```

The CONTROL CARDS for PSB Generation

```
/*
```

where

key word operand MBR= name of PSB to be generated

JCL - Batch Processing Region

The DLIBATCH procedure is to be used for batch region execution. The user must append to this procedure DD cards for the data sets which represent the physical storage of his data base. The JCL required for invoking the DLIBATCH procedure is:

```
//DLIBATCH     JOB MSGLEVEL=1
//JOBLIB       DD      DSNAME=DLI.RESLIB,DISP=SHR
//              DD      DSNAME=DLI.PGMLIB,DISP=SHR
```

where application program and PSB have same name:

```
//              EXEC    DLIBATCH,PARM='3,PSBNAME'
```

where application program has a different name than the PSB:

```
//              EXEC    DLIBATCH,PARM='3,PGMNAME,PSBNAME'
```

where

PGMNAME equals the application program name

PSBNAME equals the PSB name

System Definition

To summarize, the different libraries made available or modified by the user or by the system definition program are as follows:

Data Base

```
DLI.RESLIB
DLI.MACLIB
DLI.PSBLIB
DLI.PGMLIB
DLI.PROCLIB
DLI.DBDLIB
SYS1.SVCLIB
SYS1.LINKLIB
```

The Job Control Language (JCL) for Stage 1 of system definition is for an assembly execution. Use the standard Operating System/360 assembler procedure (ASMFC) with the following SYSLIB DD card override. The user generates a card deck of the following format and places these cards in the job stream.

```
//        JOB
//        EXEC  ASMFC
//ASM.SYSLIB DD DSNAME=DLI.GENLIB,DISP=OLD
//ASM.SYSIN  DD *
```

```
Data Language/I Stage 1 -
INPUT CONTROL CARDS -
SYSTEM DEFINITION PROGRAM
```

```
/*
```

The resulting output deck completes Stage 1. The JCL for Stage 2 is only a JOB card supplied by the user generating the system and placed in front of the punched card deck received from Stage 1. Place this deck of cards in the job stream.

Data Set Allocation

The various partitioned data sets used by Data Language/I for libraries must be defined and allocationed by the user. The DCB characteristics for these data sets should be specified at time of allocation. In all cases these DCB characteristics should be equated to existing Operating System/360 partitioned data sets. This can be done with a DCB= operand of the DD card used for allocation. The following table illustrates the Data Language/I - OS/360 data sets which should have equivalent DCB characteristics:

Data Language/I	OS/360
DLI.RESLIB	SYS1.LINKLIB
DLI.PGMLIB	SYS1.LINKLIB
DLI.PROCLIB	SYS1.PROCLIB
DLI.MACLIB	SYS1.MACLIB
DLI.PSBLIB	SYS1.LINKLIB
DLI.DBDLIB	SYS1.LINKLIB

Inclusion of SVC Routines in Operating System/360 Nucleus

A user SVC routine must be added to the Operating System/360 nucleus for execution of the Data Language/I system. The SVC routine is created by Data Language/I system definition from macro-instructions. The SVC number utilized may be specified by the system user. The load module which represents the SVC routine is placed in DLI.RESLIB by system definition. The SVC routine used for OSAM is a Type 2 SVC.

When the Data Language/I user performs his Operating System/360 system generation, the appropriate accommodations must be made for the later incorporation of the SVC routine. The SVC routines need not and normally would not be incorporated at Operating System/360 system generation. It may be incorporated at Operating System/360 system definition if desired, however. The following SVCTABLE control card should be included in the Stage 1 input to Operating System/360 system generation no matter when the SVC routine is incorporated.

```
SVCTABLE    nnn-T2-SO
```

If the actual SVC routine is not incorporated during Operating System/360 system generation, a "dummy" load module should be placed in the RESMODS partitioned data set. This should be done prior to Stage II of Operating System/360 system generation. This module is of the format:


```
IGCXXX  CSECT
        BR  14
        END
```

where XXX is the unique SVC number. This effectively "no-ops" the SVC number.

The alternate approach, which would cause inclusion of the actual SVC routines during Operating System/360 system generation, would require placement of the actual SVC modules into the partitioned data set referred to by the RESMODS control card. This would require Data Language/I system definition execution prior to Stage II of Operating System/360 system generation. The RESMODS control card could then refer to the DLI.RESLIB data set for the incorporation of the SVC routine.

If the SVC routine is added after Operating System/360 system generation, the technique is to relink-edit the Operating System/360 nucleus. Basically, this involves replacing the "dummy" SVC routine through the link-edit with the actual. The best method of performing this link-edit is to:

1. Start with JCL and control cards of link-edit step from Stage II of Operating System/360 system generation.
2. Provide an additional card for the DLI.RESLIB data set to access the SVC modules.
3. Provide an additional DD card to reference the SYS1.NUCLEUS data set other than //SYSLMOD.
4. Provide additional INCLUDE control cards for the SVC routine from DLI.RESLIB immediately after the INSERT control cards of the original link-edit.
5. Replace the INCLUDE cards from the original Operating System/360 nucleus link-edit with one INCLUDE card for the old Operating System/360 nucleus (that is, the one without the SVC routines).
6. Use the existing ENTRY card.
7. Provide a NAME card for the new Operating System/360 nucleus (for example, IEANUCOX).

It may be good practice to consider the output from the link-edit of the nucleus as another member in SYS1.NUCLEUS (for

example, IEANUC02). The OS/360 Operators Manual, Form Number C28-6540-5, explains how to IPL an alternate Operating System/360 nucleus. If everything executes properly, then IEANUC02 can be renamed IEANUC01.

Procedure Section

The following procedures should be utilized to place Data Language/I in MFT-II or MVT link-pack. This procedure should be placed in SYS1.PROCLIB using the Operating System/360 utility - IEBUPDTE program.

```
./       ADD          NAME=IEAIGG01,LIST=ALL                    X
SYS1.LINKLIB       DFSIDLD0,                                    X
                   DFSIDLH0,                                    X
                   DFSIDLR0,                                    X
                   DFSIDLI0,                                    X
                   DFSIOS20,                                    X
                   DFSIOS30,                                    X
                   DFSIOS60,                                    X
                   DFSIRC00,                                    X
                   DFSIWKN0,                                    X
                   DFSIISM0

./       ADD          NAME=IEAIGG02,LIST=ALL

SYS1.SVCLIB        IGG0196X,                                    X
                   IGG019J6,                                    X
                   IGG019G9,                                    X
                   IGG019JV,                                    X
                   IGG019Z8
```

The module IGG019Z8 is the OSAM channel end appendage, the last two characters of which are user determined.

When Operating System/360 is "IPLed" and the system responds with:

```
SPECIFY SYSTEM PARAMETERS
```

the modules described in the two preceding procedures are placed in linkpack if the response includes

```
REPLY   00,'RAM=01,02'
```

and are completely user dependent.

OSAM Appendage to SYS1.SVCLIB

The execution of Data Language/I system definition creates a channel end appendage for OSAM. This module is placed in DLI.RESLIB. It is the user's responsibility to move this module from DLI.RESLIB to SYS1.SVCLIB. The Operating System/360 IEHMOVE program should be employed.

END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT
END OF DOCUMENT