CONTENTS

Data Management Concepts

Index

There are broadly three methods of application program design:

- Application-oriented approach
- Centralized database approach
- Normalization process

The Application-Oriented Approach

- Legacy applications gave little to logically integrating computer-based applications with one another.
- Application evolved one at a time, with each data file customized for a specific need.
- Duplication of data across files was inevitable.
- Changes to format of data in files affected applications, which then needed to be changed.
- Changes to duplicated data needed to be reflected across files causing a maintenance nightmare.

Data Redundancy

Because data files evolved one application at a time, the same data element was often located in several places within the same processing system. For example, consider the employee-related data files that normally exist within a company's computer system; one set of files is created and maintained by the Payroll Department; another set is maintained by the Personnel Department; and, yet another set is maintained by the Medical Insurance Division. When many independent, logically related files exist within the same system, many are likely to contain the same data elements. The repetition of information within separate files of a data processing system called *data redundancy*.

The Figure below illustrates the common data elements that might be required by each department.

Payroll	Personnel	Medical
Social Security Number	Social Security Number	Social Security Number
Name	Name	Name
Address	Address	Address
Salary	Martial Status	Department
Pay Period	Sex	Allergies
Deductions	Dependents	Visits
Exemptions	Job code	Treatments
Absences	Performance evaluations	Medications

Redundant Data Elements

Data Integrity

When many independent data files within a system contain identical data Elements, the significance of change is greatly magnified. For example, if an employee's name or address changes, there is no way to simultaneously update this information in every logically related data file. If values for the data element are changed in some files, but not in others. The integrity of the data is jeopardized.

Data Dependency

Programs within an application-oriented system are custom-designed according to the format and physical location of the data they process. Such applications are called data dependent because they usually must be revised whenever a change in data structure or in access method occurs. In a large computer system, this can mean rewriting hundreds of different programs. Data file reorganization or a change from sequential to direct access to the data is an example of a situations when dependent programs would need to be rewritten.

The Centralized Data Base Approach

One approach to eliminating many of the problems associated with application-oriented data systems is to create a common data file for related entities. This was one of the first design approaches to data management technology. Figure below shows how combining data in a common data file would appear.

Social Security number
Name (last, first, initial)
Salary
Pay period
Deductions
Exemptions
Absences
Marital status
Sex
Dependents
Job code
Performance evaluations
Department
Allergies
Visits
Treatments
Medications

Single Data Base

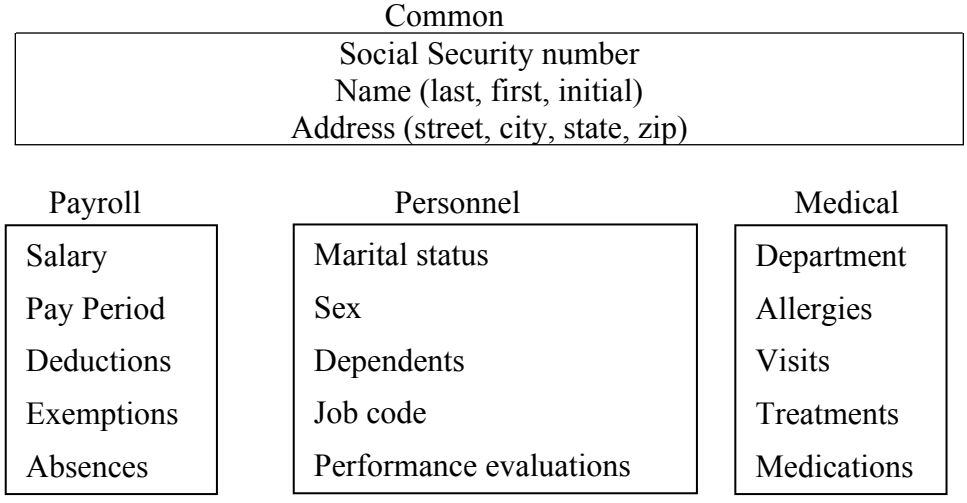
Data redundancy and data integrity problems are solved because storing information in a single data base means that there is only one version of each data element. If an employee's name or address changes, you can correct the data element through a single entry, and everyone will have access to the corrected information.

However, by consolidating all of the data elements into a single database, some new problems are created.

- This approach provides no way to distinguish one application data from another. (The data dependency issue has not been addressed yet.)
- Also, because all of the data is shared, you cannot prevent unauthorized users from accessing confidential information. Thus, a data security problem is created.
- From the efficiency viewpoint, if multiple programs access the same data, parallel access usually cannot be given and access needs to be serialized.
- Figures below show methods you can use to group logically related data into non-redundant groups, thereby creating a simple database. This process is called *normalization.*

The Normalization Process

Normalization helps prevent data dependency and helps enhance data security by analyzing the data's interrelationships and organizing it in the following manner.



Data Base with Mini-Records

In the Figure above, specific data is grouped into mini-records within the database. You now have a record for the data that is common to all three applications, and separate records for each of the three departmental applications. Organizing data this way solves the data separation problem. The data security issue is not addressed, but the foundation for solving the problem has been laid by organizing the data into groups. Grouping the data makes it easier to identify the security needs for each application and to implement measures that will protect confidential information from unauthorized access.

Summary

The application-oriented approach to developing computer systems leads to several problems.

Data integrity is endangered because the system generates several customized data files. For examples, if a value is updated in one file, but not in others, data integrity is lost.

Data dependency is another problem with the application-oriented approach because data requires detailed storage specifications. Its reliance on the physical location of program data leads to inefficient use of programming resources throughout the processing system.

Normalizing data and implementing a centralized database solves data redundancy and data integrity problems, but this does not ensure data-independent programming. A centralized database also creates a new requirement-data security ensuring that only authorized users have access to confidential information within the database.

Information Management Terminology

Index

When you complete this section, you will be able to do the following:

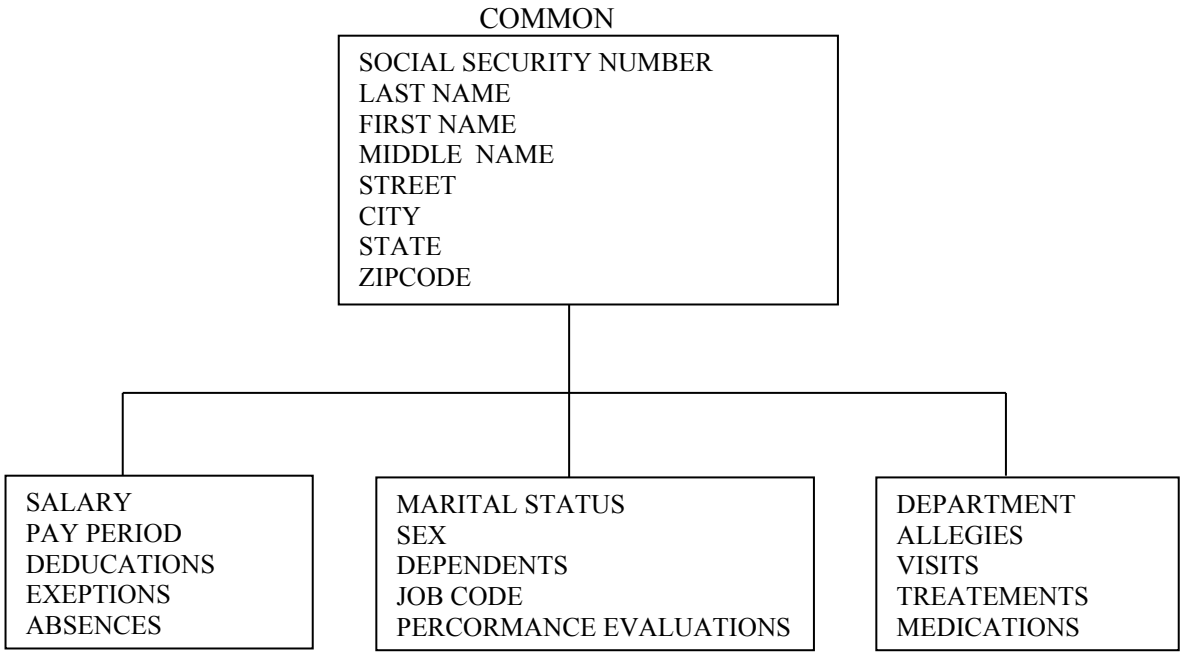
- Organize data into a hierarchically structured physical database.
- Structure logical databases by taking an alternative view of a physical database.
- Structure logical databases by combining two physical data base via a common data link.

Hierarchical Structure

The first phase of developing an information management system is initiated by organizing data for several different applications into non-redundant data groups. This process is known as normalization.

To create a viable data structure, you must extend your analysis beyond the normalization process. You must develop a logical system in which the relationships between all of our data groups are defined in a consistent manner from one data structure to the next. A common system used to organize data elements into logical groups is the Hierarchical Data structure. Hierarchical data structures are characterized by groups of data elements organized by their relative dependence or independence upon each other.

Figure below is an example of a hierarchical structure that represents a company's payroll, Personnel, and Medical department data files.



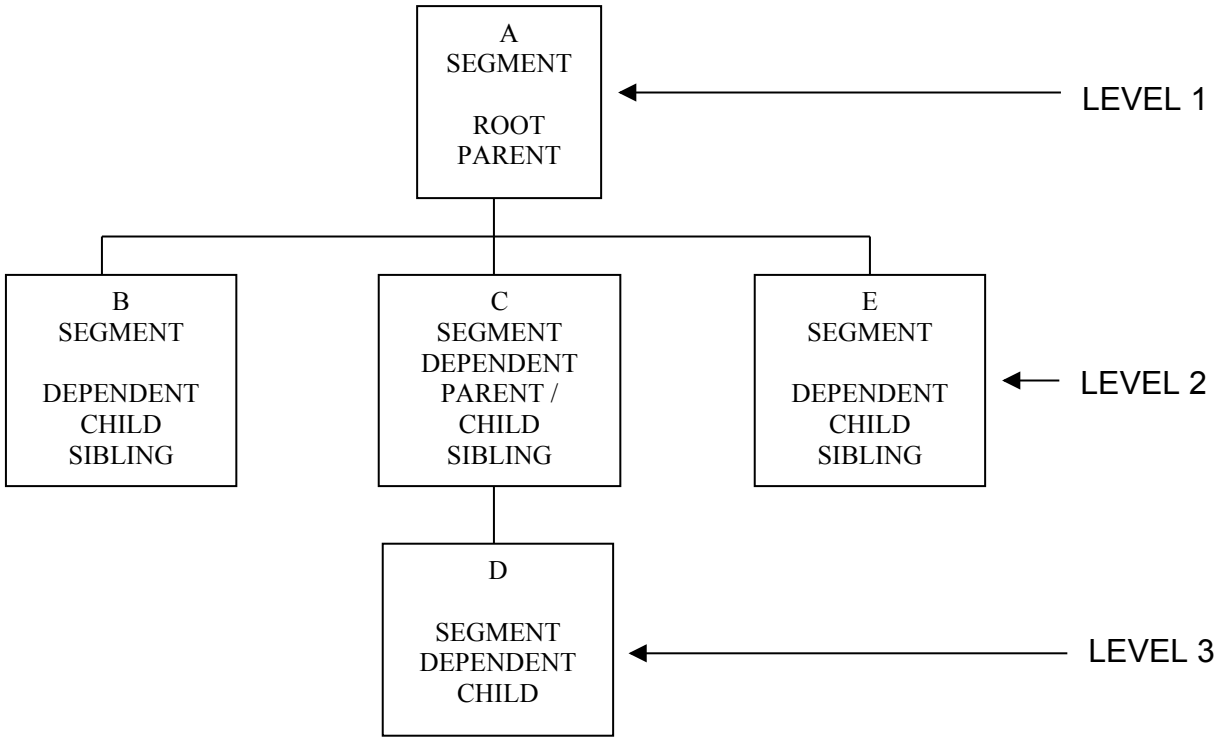
Hierarchical Data Representation

In the Figure above, the COMMON data group is related to the PAYROLL, PERSONNEL, and MEDICAL data groups. The lines connecting the COMMON box to the PAYROLL box, the PERSONNEL box, and the MEDICAL box illustrate their relationship. The logical structure can be Specified into more detailed relationships, but the conceptual framework for relating groups remains essentially the same.

A hierarchical data chart is arranged similar to topic outline in descending order of specificity. The most general information in the database, such as employee Name, address, and Social security numbers, is located in the COMMON box. The more specific data appears below the common group. From a functional perspective, the higher a data group is situated in data base hierarchy, the greater the number of users who might access it. Its range of potential applications is also wider.

Segments

Each group of logically related data elements is called a segment. A segment contains a group of data elements, similar to a data record. However, a date record usually contains data elements related to a specific application, while a segment contains only data elements that are logically related to one another. The Figure below is a graphic representation of five hierarchically structured date base segments. There is a maximum of 255 segment types in any single IMS database.



Segment Relationships in a hierarchical database

Roots and dependents

Every segment in a database is either a root or a *dependent*. A root is the segment at the top of a hierarchical data base structure. Every hierarchical structure has only one root. A dependent segment is one that has a contingent relationship to another segment situated above it in the hierarchical structure. A root is never a dependent segment because it is the first segment in a hierarchical structure. Conversely, segments below the root are always dependent ones. The hierarchical structure shown in the above figure (Segment Relationships in a Hierarchical database) contains one root and four dependent segments.

Levels

Level designations are given on the right side of the Figure(Segment Relationships in a Hierarchical data base) above. The hierarchical structure in this example is composed of three levels. The root segment of all the databases is located on level one. Segment A is the root segment of the above Figure(Segment Relationships in a Hierarchical data base). Segments B, C and E are all direct dependents of segment A, and are all located on level two of the hierarchical structure. Segment D is directly dependent on segment C, and indirectly dependent on segment A. Since segment D is a dependent on both the root and segment C, it is situated in level three of the hierarchical structure.

Parents and Children

A segment that is directly related to segments in a lower level is a *parent* segment.
Segment A is the parent of segments B, C and E. Segment C is the parent of segment D.

A segment that is directly related to a segment in a higher level is a child segment segments B, C, D and E and all children segments B, C, and E are children of segment A, and segment D is the child of segment C. Two additional points are important regarding our hierarchy example. Segment A, the root, can never be a child. Segment C is both a parent and a child. The root is always a parent segment in a hierarchical data base structure. All dependent segments are children. However, some dependent segments are parent segments.

Another segment relationship is the sibling. A sibling is any segment that has the same parent as another segment, but is of a different segment type. Segments B, C and E are siblings. They all have the same parent (segment A), but each is a different segment type. Siblings must be at the same level in the database. No segment can have more than one parent. This type of hierarchical arrangement is called a tree structure.

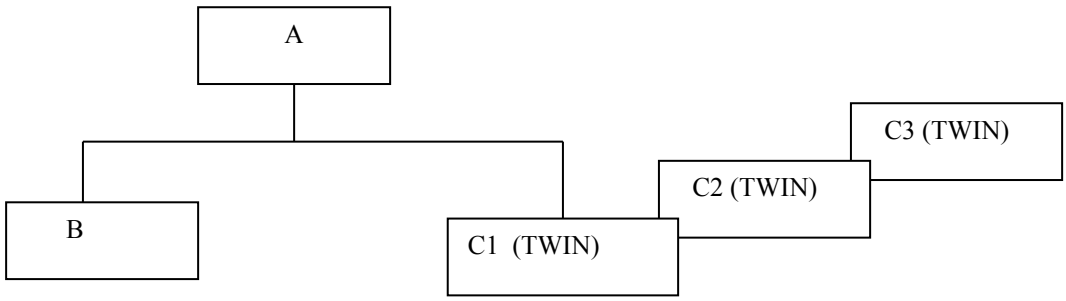
Relational Lines

Each vertical line in the above figure shows a relationship between two particular segments. In this case, there are four such relationships;
A to B;
A to C;
C to D;

A to E .

Segment Types and Segment Occurrences

It is important to distinguish between segment types and segment occurrences. Segment types are categorical descriptions of segments in a database.



Segment Types and Occurrences

In the Figure (segment Types and Occurrences), there are five segment occurrences (A, B, C1, C2 and C3), but only three segment types (A, B, and C). Each segment in the structure represents a segment occurrence. Each occurrence contains data of a given segment type.

Twins

A segment type that has two or more occurrences is a twin. Segments C1, C2, and C3 are twins. All twins share the same parent.

It is important, to distinguish between twins and siblings. Siblings share a parent, but are different segment types. Twins are multiple occurrences of the same segment type.

Looking back at Figure (Segment Relationships in a hierarchical database), you see that the payroll, personnel, and the Medical segments are siblings. The parent of these segments is the root, which contains other common information. If a segment type occurs more than once, for example, a payroll segment, the occurrences are referred to as twins.

Data Bases

- A hierarchical structure is made up of logically related segments.
- Each segment is made up of logically related data elements. This pool of logically organized data is called a database.
- A data file services a single application, but a database can service many applications.
- A database contains all occurrences of all segment types in a logically organized data structure.

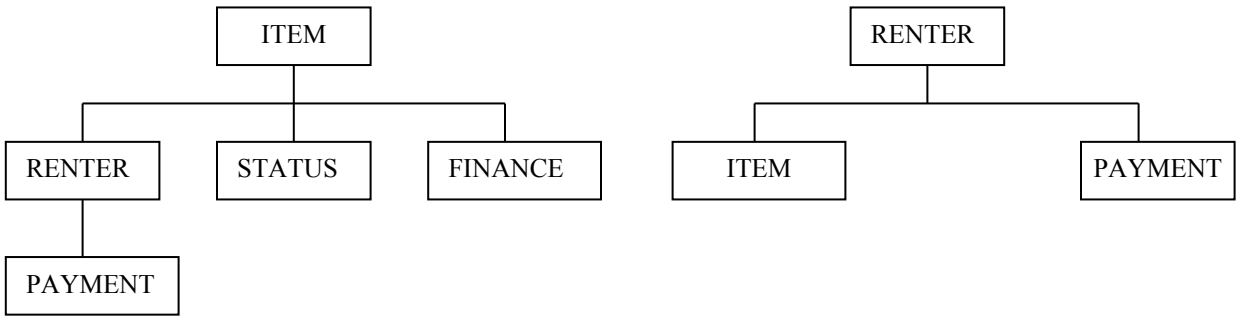
Theoretically, an organization needs only one date base that contains all data elements used for every application. However, today's hardware technology cannot provide the necessary speed to efficiently access such large volumes of data through a single database. To compensate for this, large modern organizations use a number of smaller databases. Each services a limited number of applications. Using smaller, multiple data bases also avoids the risk of losing all data if a system problem arises while a given data base is being accessed.

Physical Data Base

The structures that have been discussed so far contain at least one occurrence of every possible segment type. However, each segment type may not physically exist in a database, depending on how the database has been loaded or maintained. The physical database consists of all existing occurrences of segment types.

Logical Data Base

It is possible that an application may require access to a physical database, but the logical perspective is different from its physical organization. In the Figure below a physical database is shown on the left, and a logical database on the right.



Physical data base **Logical data base**

Logical Data Base from a Single Physical Data Base

The physical data base structure represents a company that rents items to the general public. The company is primary concern is its rental items. Accordingly, the data base was designed with ITEM as the root segment. Suppose, the company is going to reward customers who meet the following two qualifications:

- A satisfactory payment history
- Past rental of two or more items

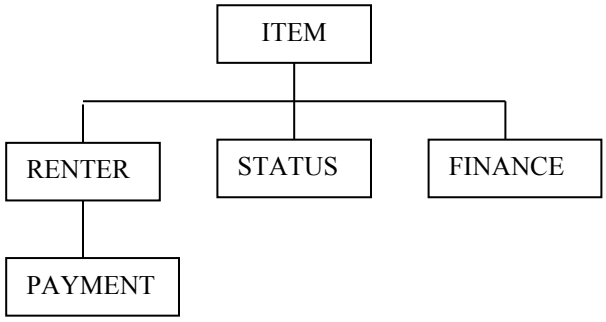
To identify customers who qualify. You want to view the physical database differently, using the RENTER segment as the root. To achieve this you do not need to create a new

database with RENTER as the root segment. Instead, you can define a logical database view of the physical database to your application. The Figure above is an example of that logical database. It contains references to only those segments required for processing.

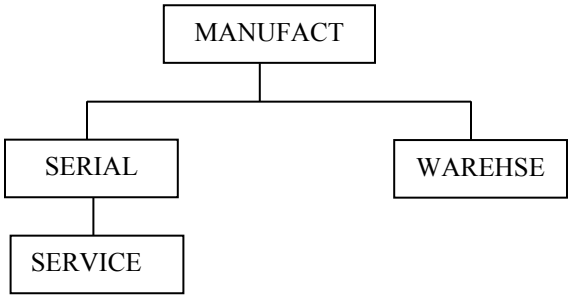
- The RENTER segment
- The PAYMENT history segment
- The rented ITEM segment

Creating an alternative view of a physical database is one way of using a logical database. It is important to know that the segment-to-segment relationships in the physical database are preserved in the logical database, even though the structure appears different. For example, the ITEM segment is still directly related to the RENTER segment in the logical database shown in above Figure. However, in the physical structure, ITEM is the parent of RENTER, but in the logical structure, RENTER is the parent of ITEM.

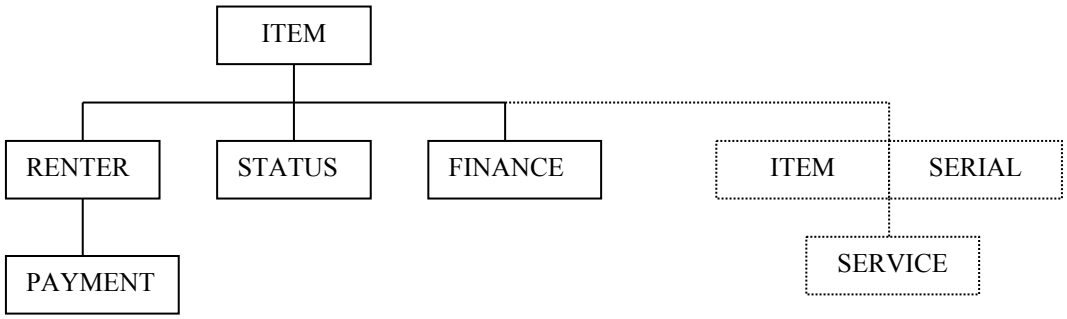
Another way of using a logical database is to create an alternate view of two or more physical databases.



Rental physical data base



Manufacture physical data base



Rental Logical Data Base

Logical Data Base form Multiple Physical Data Base

The above Figure (RENTAL LOGICAL DATA BASE) is an example of a logical database derived from two physical data bases. They are the rental item database and the item manufacturer database.

You want to identify the items that require the most service, but the item data base does not contain service information, and the manufacturer data base does not contain the internal item numbers. Here you have two physical databases, each contains part of the data you need to solve your problem. Your problem can be solved by logically viewing the required data from both physical databases. To do this, find a common data element In this case, the manufacture's serial number data element is in both data bases and can serve as a link The resulting logical data base is shown in Figure (RENTAL LOGICAL DATA BASE).

Let's examine the logical structure closely. Notice the addition of the ITEM / SERIAL and SERVICE segments to the rental data base. This is a logical addition, not a physical one. The dashed lines indicate that this part of the structure is logical.

The ITEM / SERIAL segment is a concatenated segment. The concatenated segment contains the data from both the ITEM and the SERIAL segments.

You have learned about two hierarchical structures; the physical and logical databases. The physical database is the actual database. The logical database is a view of one or more physical databases. All types of segments, such as parents, siblings, and twins, can coexist in both physical and logical databases. In Figure (RENTAL LOGICAL DATA BASE), the following information can be derived from one of the two physical data bases;

- The SERIAL segment is a physical child of the MANUFACT segment.
- The MANUFACT segment is the physical parent of the SERIAL segment.
- The ITEM segment is the physical parent of the FINANCE segment.

The following is true in the logical data base structure.

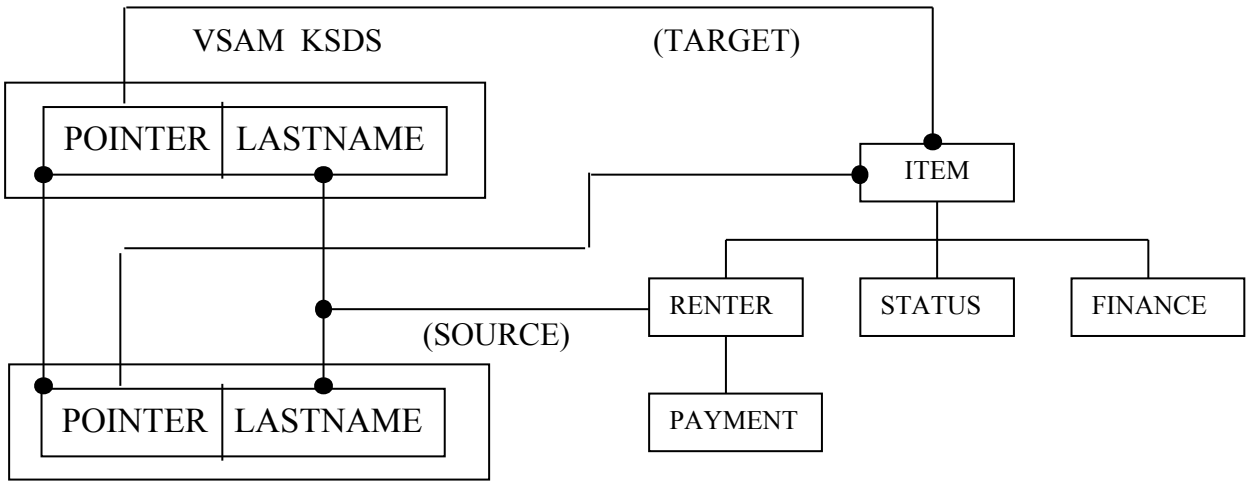
- The SERIAL segment is a logical child of the ITEM segment.
- The ITEM segment is the logical parent of the SERIAL segment.

Secondary index Data Base

Another facility of IMS is the ability to access segment types by data fields other than a designated key field. This is necessary because IMS allows you to define only one key field for each segment type.

Suppose you need to search the rental data base by the renter's last name to see what items this person has rented. You want to use last name because the Social Security number (the key field of the segment) is not available to you. The current definition of the database causes you to search all items and potentially all RENTER segments to satisfy your needs. This is obviously not an efficient method of solving your problem.

IMS provides a facility that lets you create another database that contains an index. This additional database is called a secondary index. A secondary index on last name can rapidly locate a person by last name and the corresponding items that they rented. Figure below shows how IMS creates the secondary index based on your requirements.



Secondary Index Built from a Physical Data Base

To take advantage of secondary indexing, the physical database (or subject data base) must use HISAM, HDAM, or HIDAM storage methods (these will be discussed in the Control Block section). The secondary index must use VSAM as the access method. Typically, the secondary index consists of only a VSAM Key sequenced Data Set (KSDS). In cases where the secondary index key is not unique, as in our example of last name, then a VSAM Entry sequenced Data Set (ESDS) is created containing Keys and pointers to the duplicate keys. In practice, this is transparent to the application programmer. The Data Base Administrator (DBA) is usually responsible for setting up the secondary index after the application programmer has identified what segments need to be accessed for what purpose.

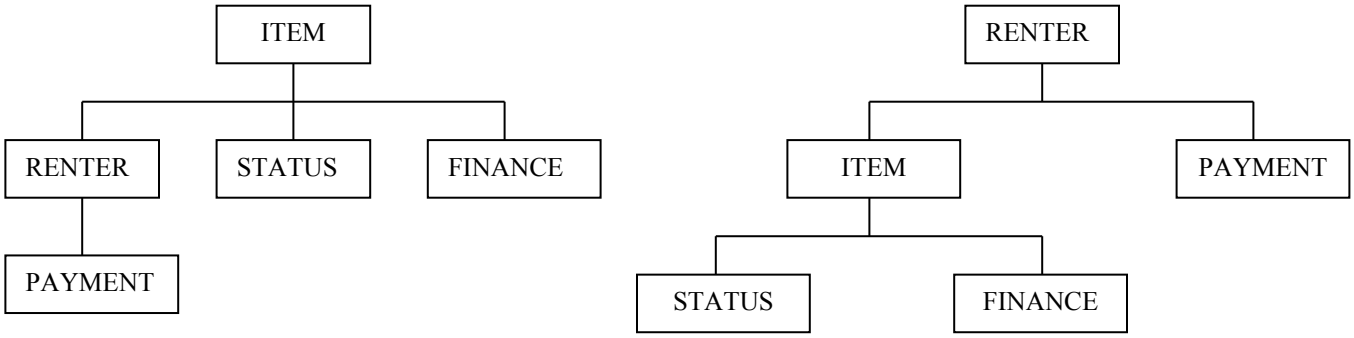
First, define the source segment of the field or fields in the subject database that the secondary index will be built on. The source segment can be the same segment type as the target segment (Described next), or it can be a dependent of the target segment type.

Next, define the target segment in the subject database that you want to access, based on the key defined by the source segment. The target segment can be, but does not have to be, the root segment.

IMS creates the secondary index by using the specified key from the subject database. The secondary index is a root only database. It has only one segment type-the pointer segment. Each segment contains the actual key value and a pointer to the target segment in the physical database. It is also possible for the DBA to include other data fields in the secondary index segment besides the key itself.

Because the secondary index is a key database, it provides quick access to data that otherwise might take much time to access. By providing access directly to the target segment in the subject database, the secondary index changes the application programs

logical view of the database if the target segment is not the root You created a secondary index for the rental item database. The target segment is the RENTER segment in the Figure below shows the logical view with and without access via the secondary index.



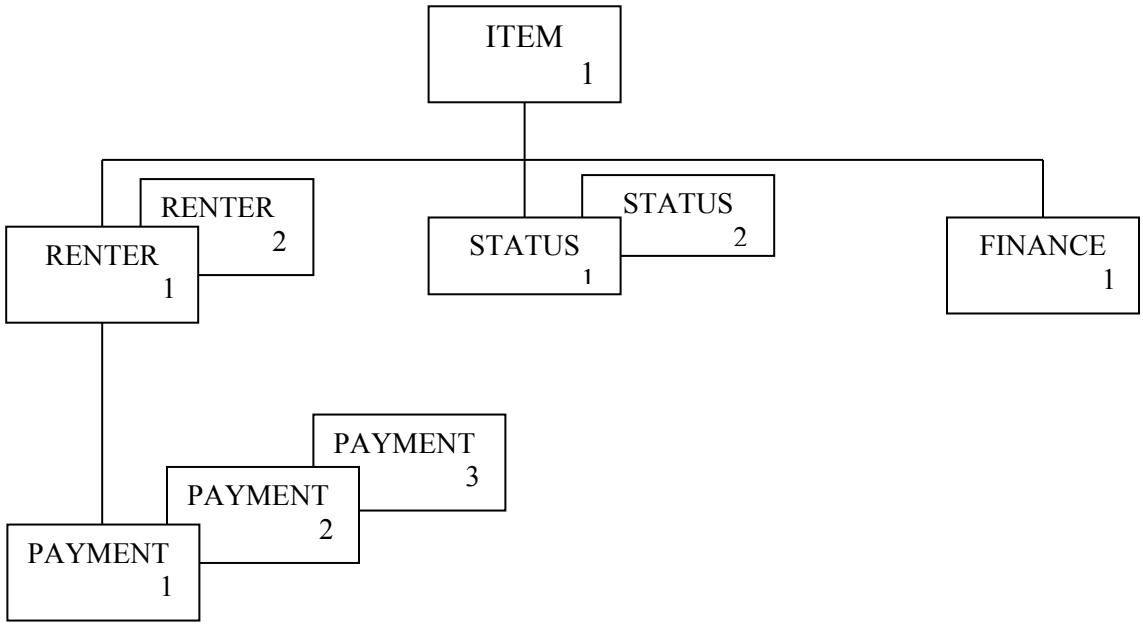
Logical View of Data Base via Secondary Index

When processing a database through the secondary index, remember that if the target segment is not the root segment, you cannot add or delete segment types on which the target segment is directly dependent. The ITEM segment is the target segment in the example. Also, once a secondary index is created, IMS maintains data that is changed or added affecting the contents of the secondary index, That means there is additional overhead when using a secondary index, especially, if the field on which the secondary index is built is updated frequently. This should be a consideration before creating secondary indexes.

IMS considers the secondary index to be a database, so an application can process just the secondary index without actually going to the target database.

Data Base Record

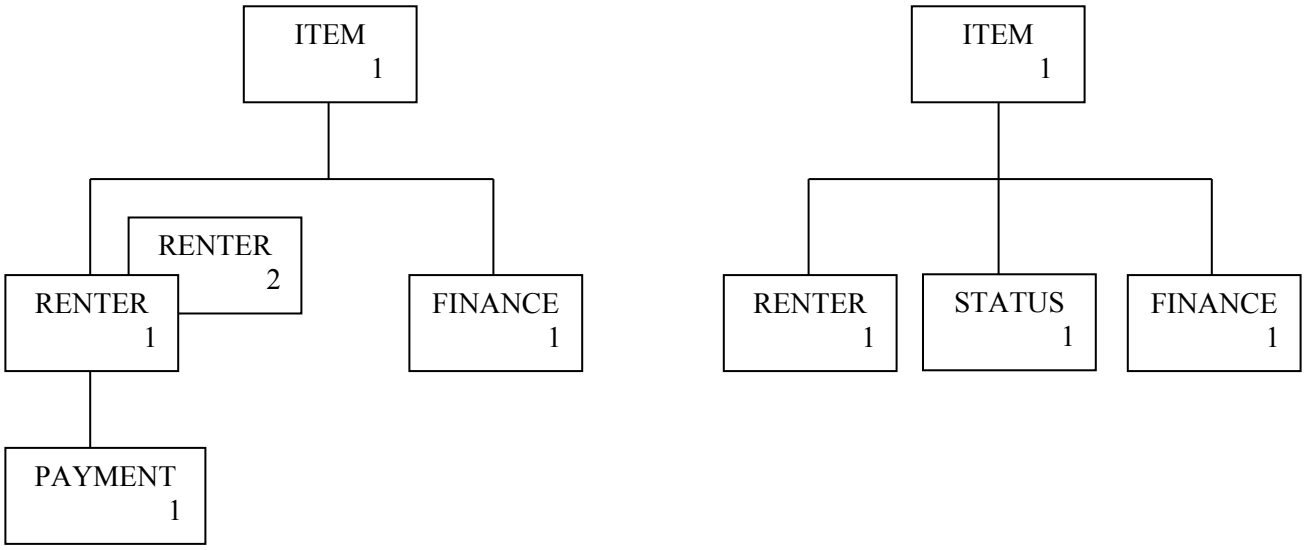
A data record is a collection of related data elements. A data base record is a record of a single occurrence of a root segment and all of its dependents. Figure below shows a data base record. In this example, the occurrence of ITEM 1, and all of its dependents-RENTER 1,RENTER 2, PAYMENT 1, PAYMENT 2, PAYMENT 3, STATUS 1, STATUS 2, and FINANCE 1 comprise one data base record.



Data Base Record

Physical Data Base Record

A physical data base record is the single occurrence of a root and all its dependent segment occurrences. Like a physical data base, a physical record may or may not contain an occurrence for every possible segment type. Figure below shows two examples of physical data base records within the same physical database shown in the figure above.

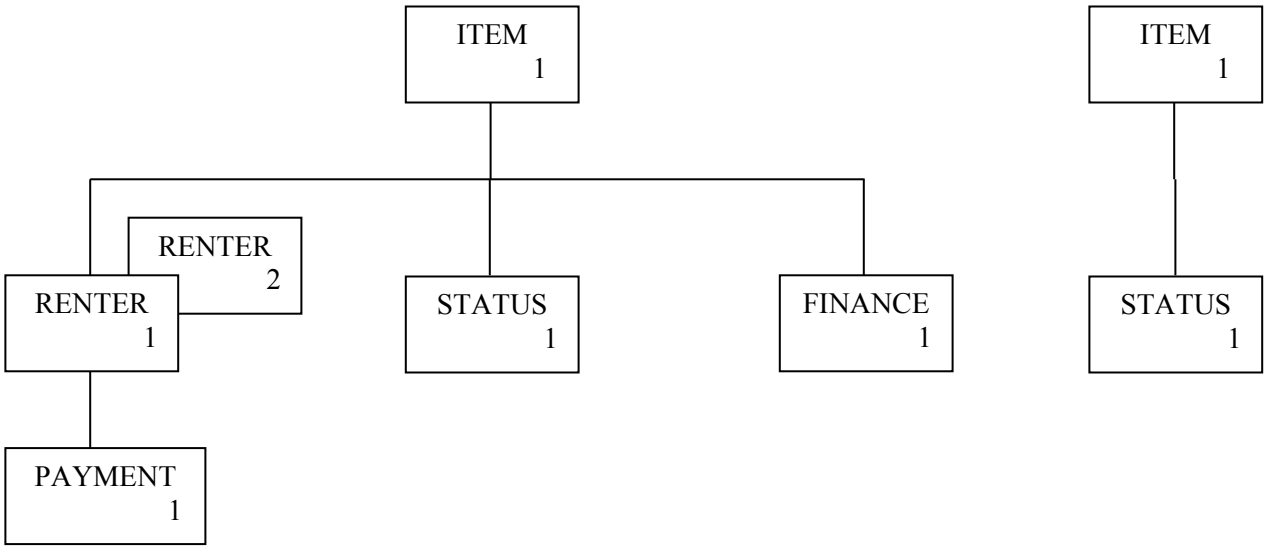


Data Base Record 1 **Data Base Record 2**

You can see from both examples that it is not necessary to have an occurrence of every segment type in the record. It is also unnecessary to have twins for every segment type. The physical database is defined to include every possible segment type. However, an occurrence of each segment type may or may not exist in the actual physical database. From a segment lower in the hierarchy, it necessary to have a segment at every level right up to the root segment. In other words a hierarchical path must exist from a subordinate segment up to the root segment.

Logical Data Base View

An application may not need access to all segment types that can occur in a physical database record. Very often you only need selected segments. Accordingly, you may want to define a logical data base view in a manner that differs from how it is physically defined. The Figure below shows the concept of a logical data base view.



Physical Data Base Record

Logical view of Data Base Record

The Figure (LOGICAL VIEW OF DATA BASE RECORD) might represent the rental company's accounting departments need for financial data on each item. Because this is the only data required, there is no need to process other segments contained in the physical data base record.

The concept of logical views is part of the foundation of data base technology. Logical data base views allow you to access only the data that is needed by your application, improving efficiency and providing a solid foundation to implement security measures.

Summary

A hierarchical structure represents one method of logically defining the relationships between all of the elements in a data structure. Each data element in a hierarchical structure is contained in a group of logically related elements known as a segment.

There is one root segment, which is located at the top of the hierarchical structure. The root is the parent of all segments located immediately beneath it because they are directly dependent on it. The direct dependents of any parent segment are its children.

Each segment shown on a hierarchical data chart is a segment type. It is a categorical description of the type of data that will occur at its designated position in the structural hierarchy. Segment occurrences are the actual data of a given segment type. Multiple

occurrences of a specific segment type under the same parent are called twins. The children of a parent segment are siblings to one another if they are different segment types.

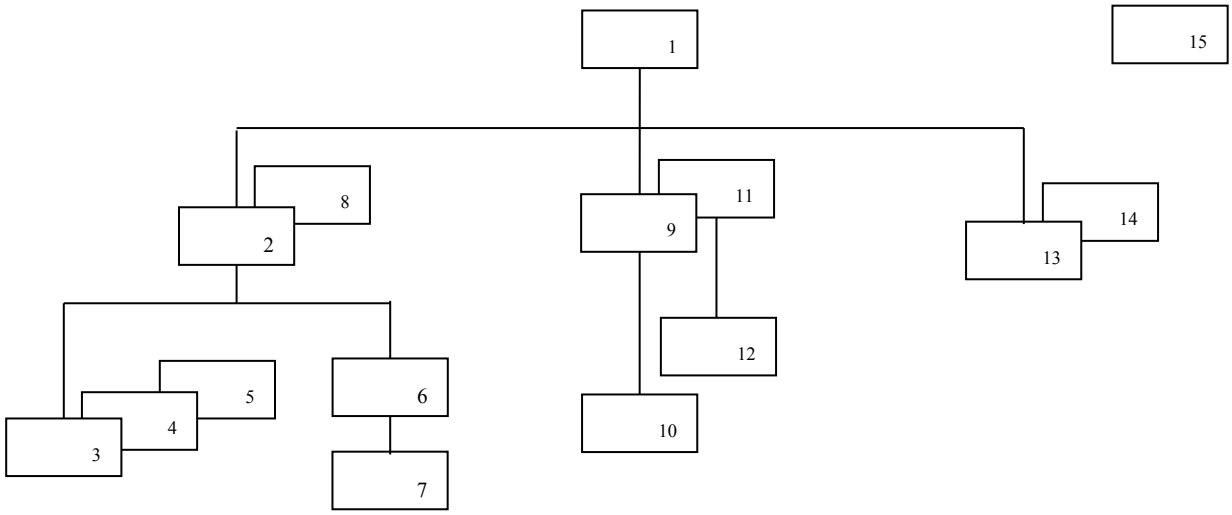
A physical database consists of all occurrences of all existing segment types.
A logical database is an alternative view of one or more physical data bases.

A data base record is a single occurrence of a root segment and all of its occurrences. While the physical data record is designed to include every possible segment type, occurrences may or may not exist for every segment type in the physical data base record. A logical data base view may not include every segment type included in the physical data base record. The logical view includes only those segments required for implementing a specific application.

Data Base Terminology Exercise

Using the hierarchical below, define each segment type using their relational names (for example, root, parent, child, sibling).

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____
9. _____
10. _____
11. _____
12. _____
13. _____
14. _____
15. _____



After you Complete the above exercise on terminology, answer the following questions about the structure:

- How many segment occurrences are there?
- How many segment types are there?
- How many data base records are there?

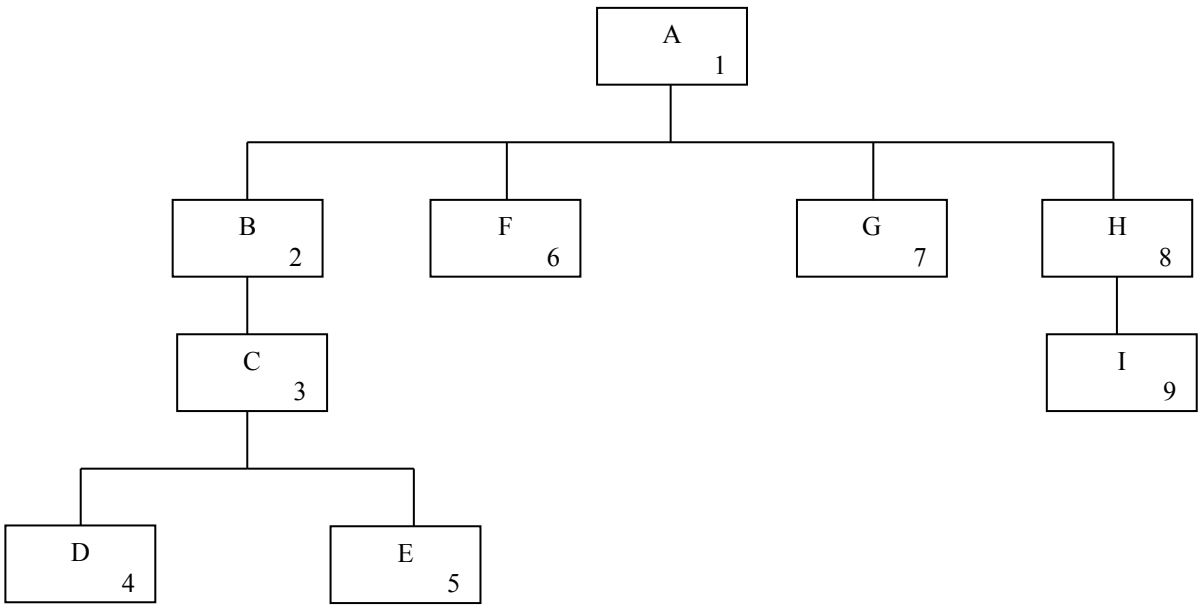
Data Base processing

Index

When you complete this section, you will be able to process a simple or complex data structure in hierarchic sequence by completing all valid paths in the structure.

Hierarchical Sequence

You should have a clear understanding of how to develop and graphically represent a hierarchical structure. The terms used to describe the structures should also be familiar to you. This section will discuss how hierarchical structures are sequentially accessed by IMS. Let's look at the simple structure shown in the Figure below.



Processing in Hierarchical Sequence - Simple

Every segment type, A through I, has a number associated with it, shown in the lower right comer of the segment box. This number, for the purposes of this example, represents the order in which each segment would be accessed or retrieved in an application program. This is called hierarchical sequence. The accessing of hierarchic structures in hierarchical sequence follows three rules;

- Rule One –Move top to bottom. Always check to see if the accessed segment has dependents. If it does, process its dependents next.

- Rule Two –Move left to right.

After performing the top to bottom check, if you encounter sibling segments, access the left-most unprocessed sibling first

- Rule Three-Move front to back.

  If the accessed dependent is at the lowest level of the hierarchy, it is now possible to access all of its twins. Do this in front-to-back sequence.

When you complete rule three, one leg of the hierarchy is finished. Then move up one level, and reapply the rules.

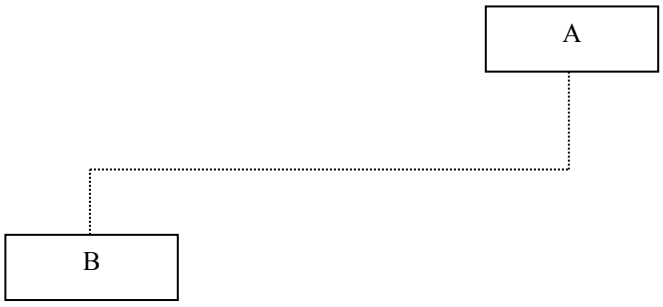
To better understand how a hierarchical structure is accessed, let's trace a hypothetical sequence through the structure shown in Figure below.

1. Start at segment A.

   This is the root segment of the hierarchy. It is the starting point for any hierarchical sequence.

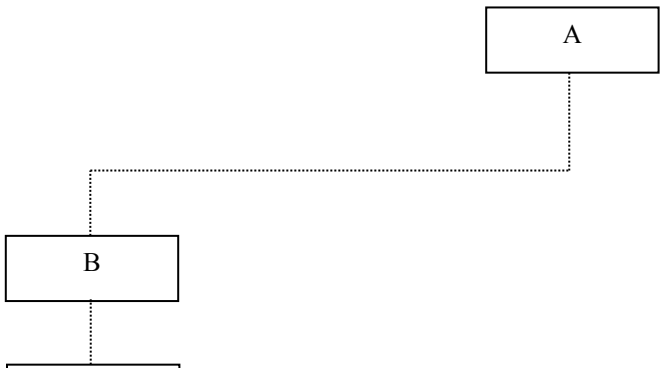
2 Access segment B.

   Move down to level two to determine whether segment A has any dependent segments. Here, segments B, F, G and H all fit this category. Find segment A's left most unprocessed dependent This is segment B.



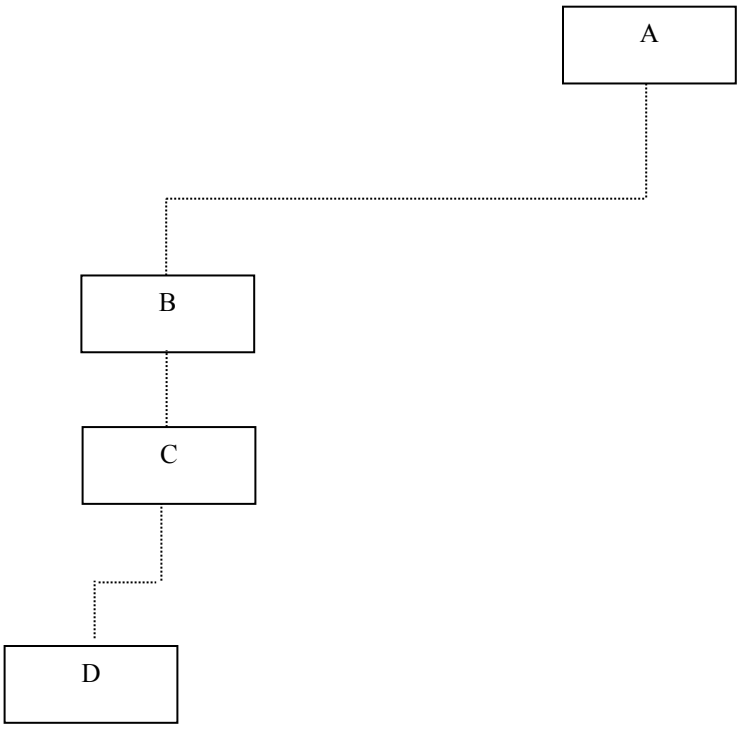
3. Access segment C.

   Return to the primary rule, and determine whether segment B has any dependents. Segment C is segment B s only dependent.



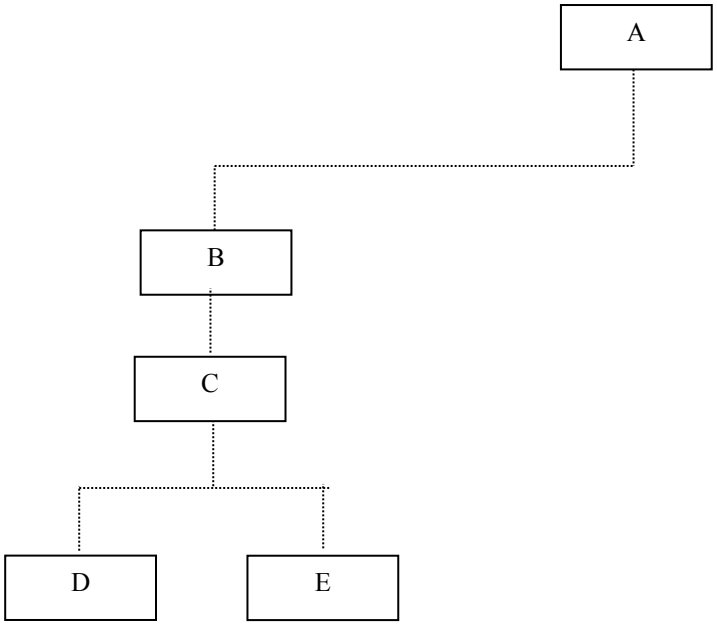
4. Access segment D.

Segment C has two dependents, segments D and E. Start with segment D because it is the left-most unprocessed dependent of segment C. Because Segment D has no dependents or twins, this leg of the hierarchy is completed.



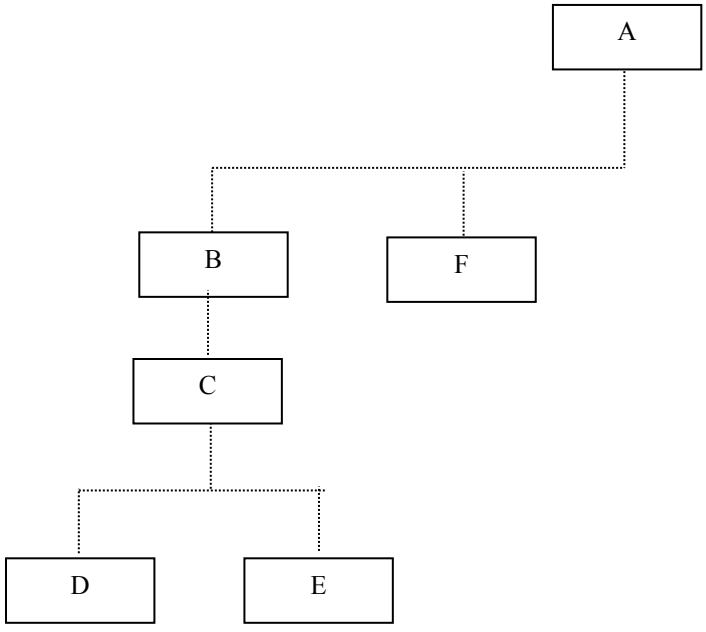
5. Access segment E.

Now that the first leg of the hierarchy is completed, apply rule three for hierarchical sequencing and, move up one level in the structure. Return to segment C because it is the parent of segment D. find segment C s left-most unprocessed dependent, and access it. This is segment E.



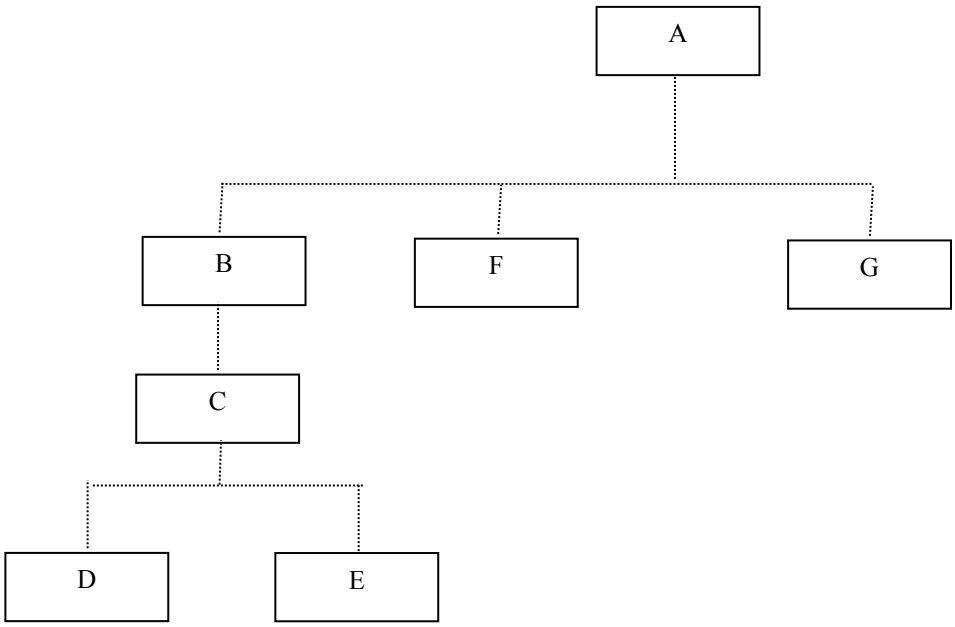
6. Access segment F

Segment E has no dependents or twins, so, you can move up to level three. Finding that segment C has no unprocessed dependents or twins, move up to level two. Segment B has no unprocessed dependents or twins. Now you can return to the root segment at the top of the hierarchy. This is segment A. Find the left-most unprocessed dependent of segment A. Move down to level two, and access segment F.



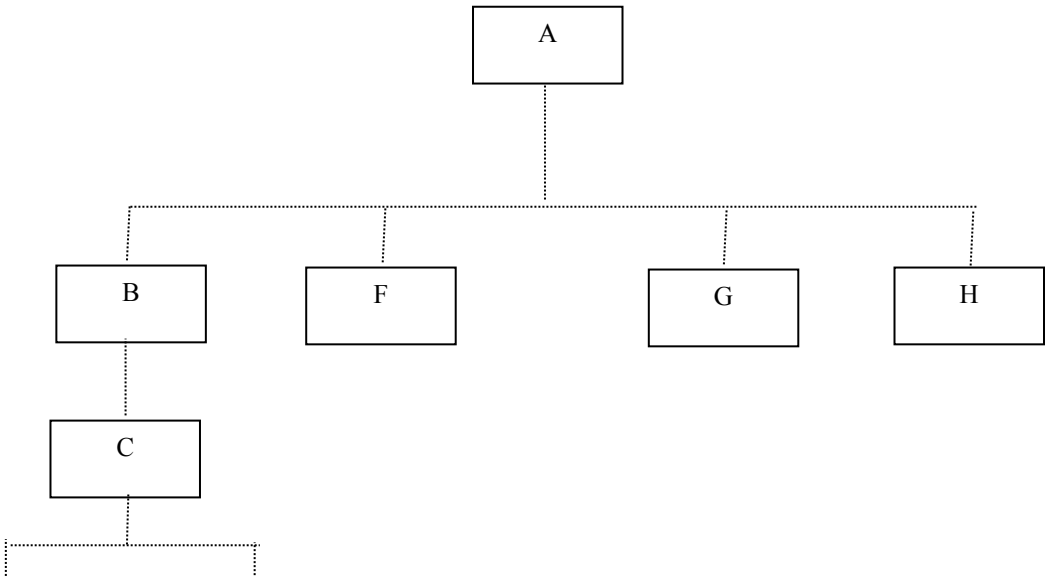
7. Access segment G.

Segment F has no dependents or twins, so, you return to segment A. Once again you must locate segment A's left-most unprocessed segment. This is segment G.



8. Access segment H.

Segment G has no dependents or twins, so, you return to segment Gs parent, segment A. Return to level two to find segment A's left-most unprocessed dependent. This is segment H.

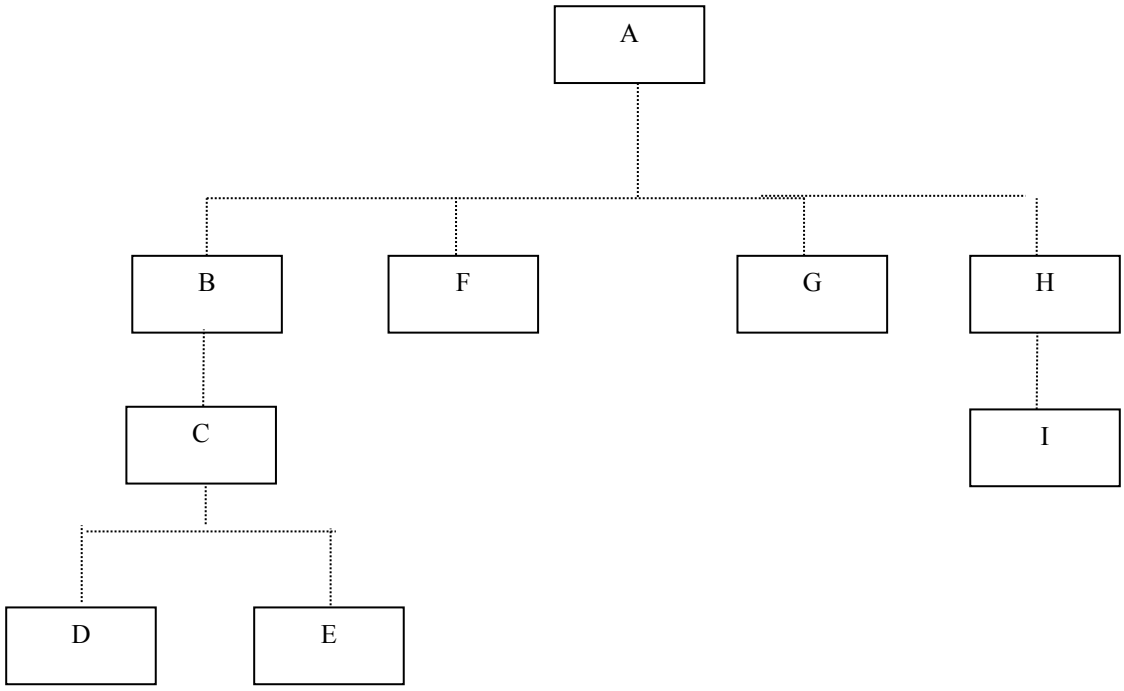


9. Access segment I.

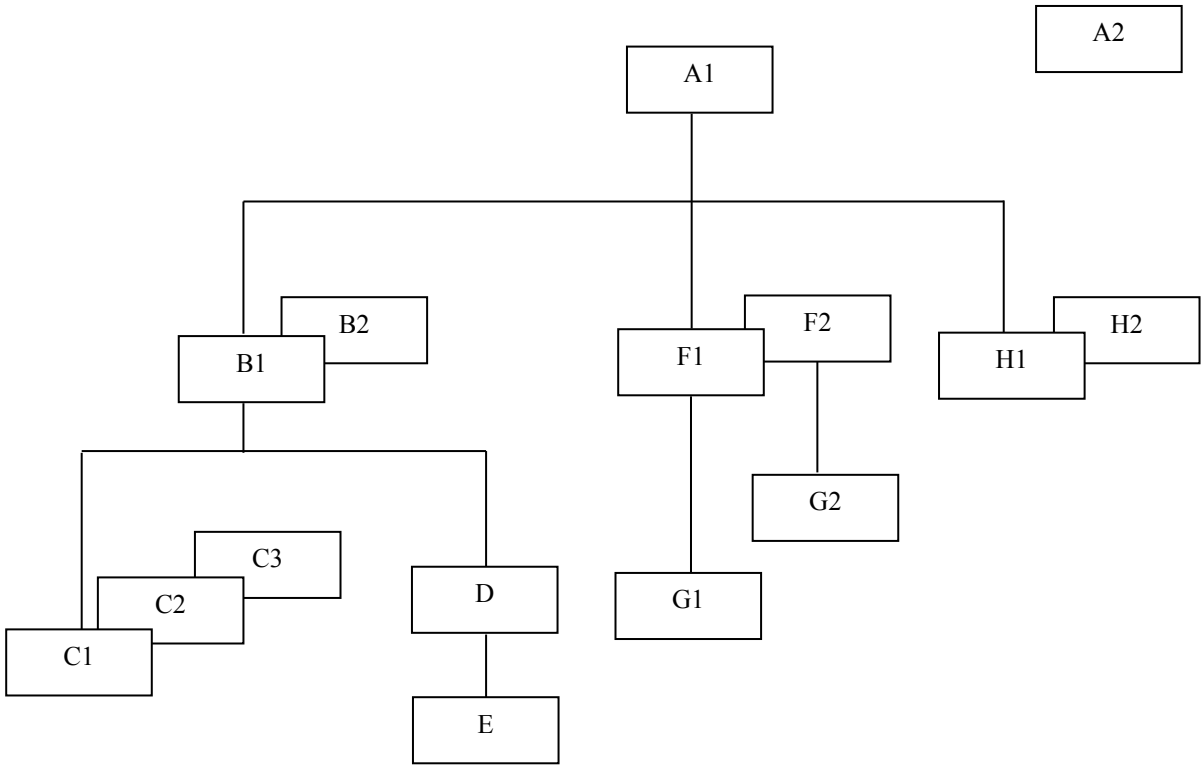
Proceed to level three and search for the left-most unprocessed dependent of segment H. This is segment I.

You have completed your processing based on the following observations;

- Segment I has no dependents.
- Segment I also has no twins.
- Segment H has no unprocessed dependents.
- Segment H also has no twins.
- Segment A has no unprocessed dependents.
- Segment A has no twins. (It is the root segment)
- All segments in this hierarchical structure have been processed.



The previous Figure was simple to process because there was only one occurrence of each segment type. This example contained no twins. The next Figure is a more complex hierarchical structure. Let's examine this structure to see what the correct hierarchic sequences is.



Apply what you learned about processing a hierarchical structure to the Figure. Follow the correct hierarchical sequence.

1. Access segment A1.

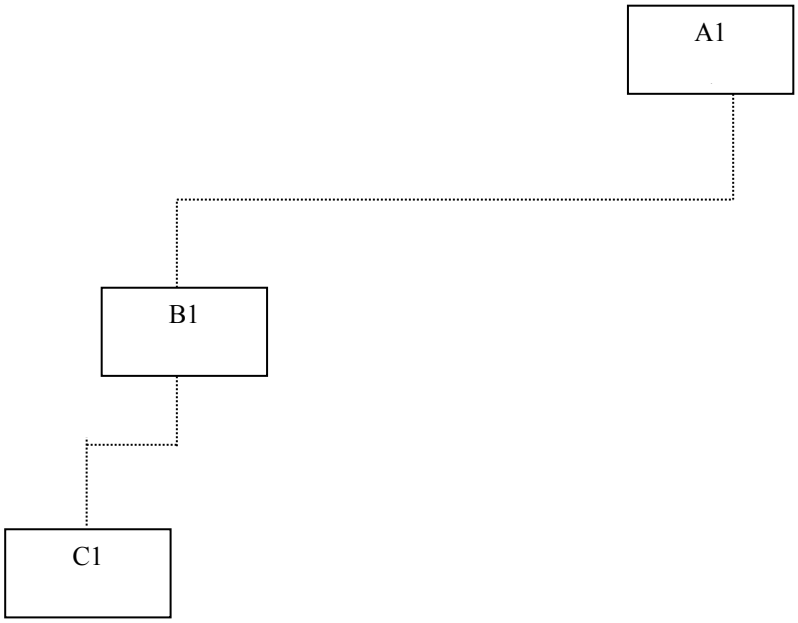
   Begin with the root segment type. This is segment A1.

2. Access segment B1.

   Find the left-most unprocessed dependent of A1. Moving down to level two, you find that this is segment B1.

3. Access segment C1.

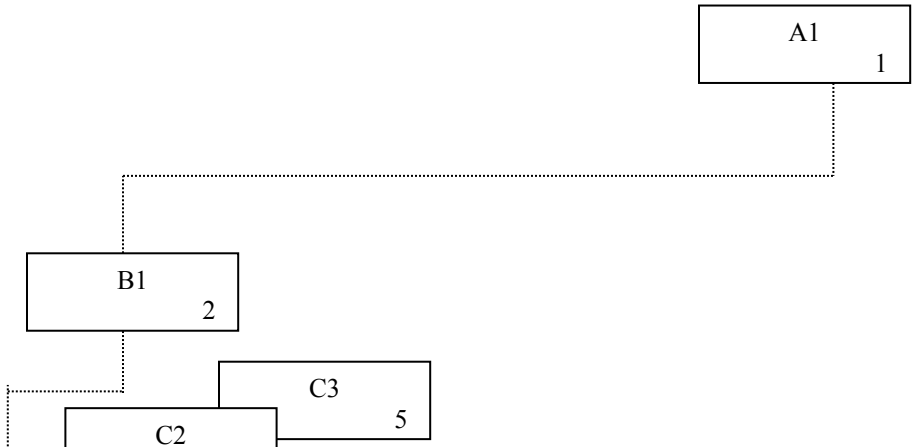
   Segment type B has twins, B1 and B2. This is different from the simple structure used in earlier Figure. Apply rules for each new segment accessed Apply rule one to segment B1. B1 has dependents, so, you must go down to the next level. There are multiple dependent segment types. Find the left most, unprocessed dependent This is segment C1.



4. Access segment C2.

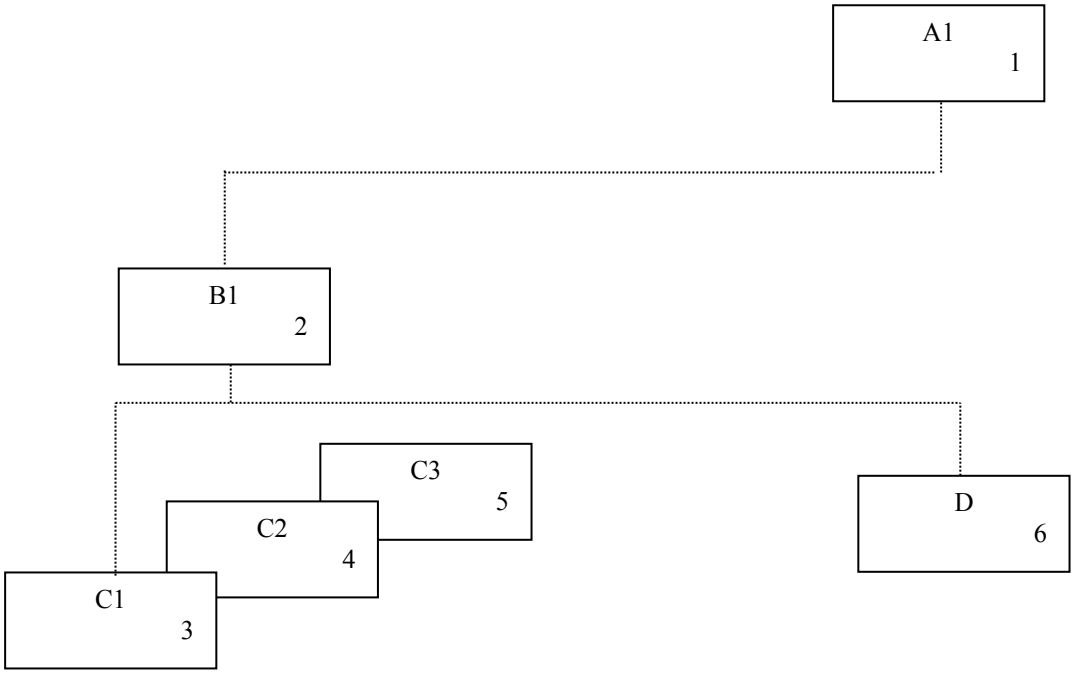
5. Access segment C3.

   Again, apply rule one. C1 has no dependents. Rule two is satisfied C1 is the left-most dependent of B1. Now apply rule three. Because segment type C has twins, you must access segments C2 and C3. This completes the processing of this leg.



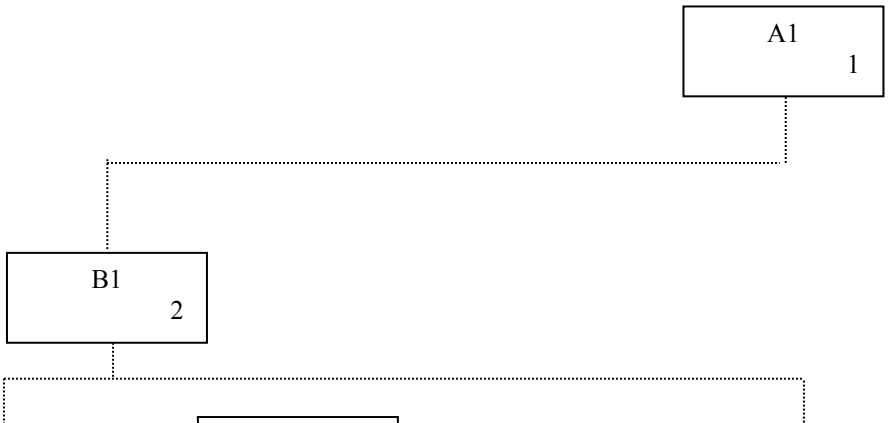
6. Access segment D.

   Move up one level to B1, and apply rule one. Segment B1 has multiple dependents. Access segment D because it is the left-most unprocessed dependent segment.



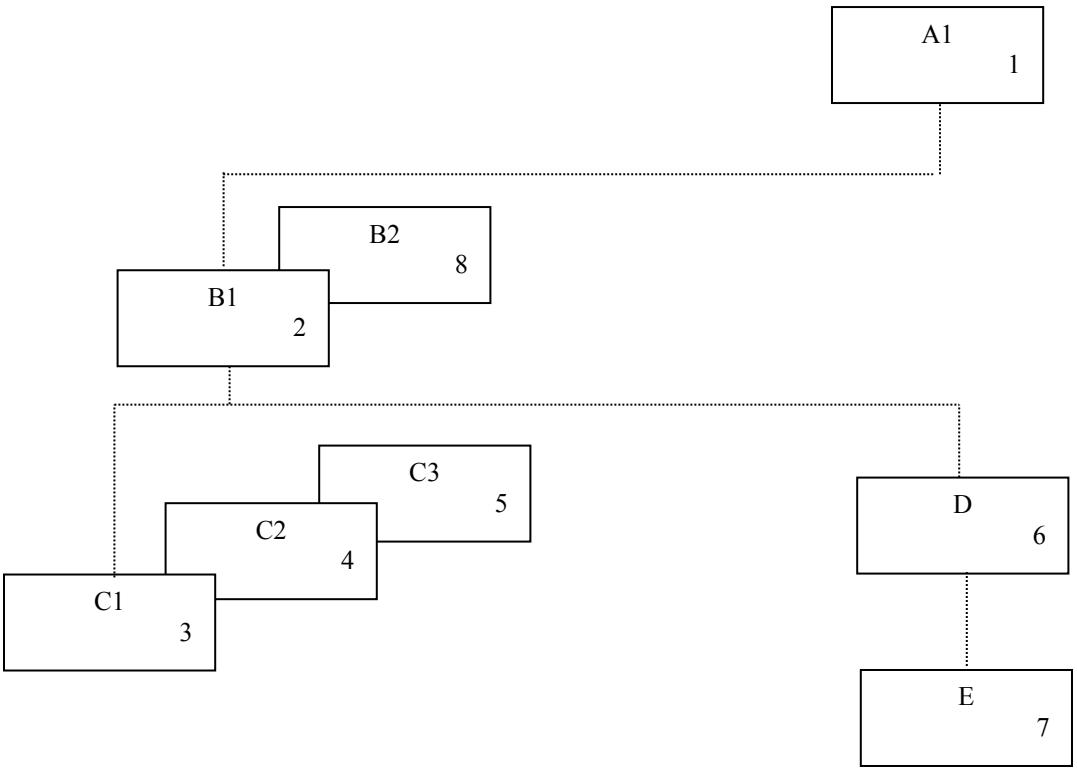
7. Access segment E.

   Rule one moves you to the next level. Segment E has no dependents, so rule two is applied. There is only one dependent segment type at this level, so, rule two is satisfied. Apply rule three. There are no twins, so this leg is completed .



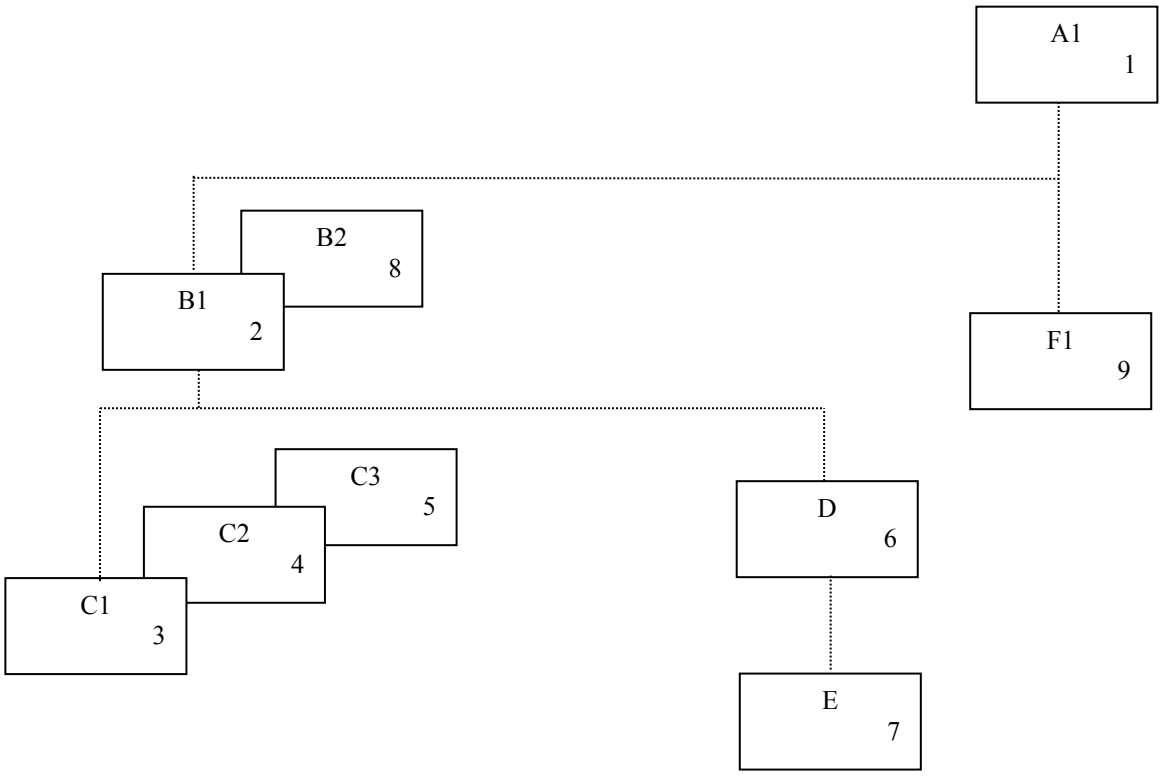
8. Access segment B2.

Move up one level to segment D. There are no more dependent segment types. Move up another level to segment B1. All dependent segments of B1 have been processed Applying rule three, You find that segment B1 has one twin, segment B2. Access segment B2.



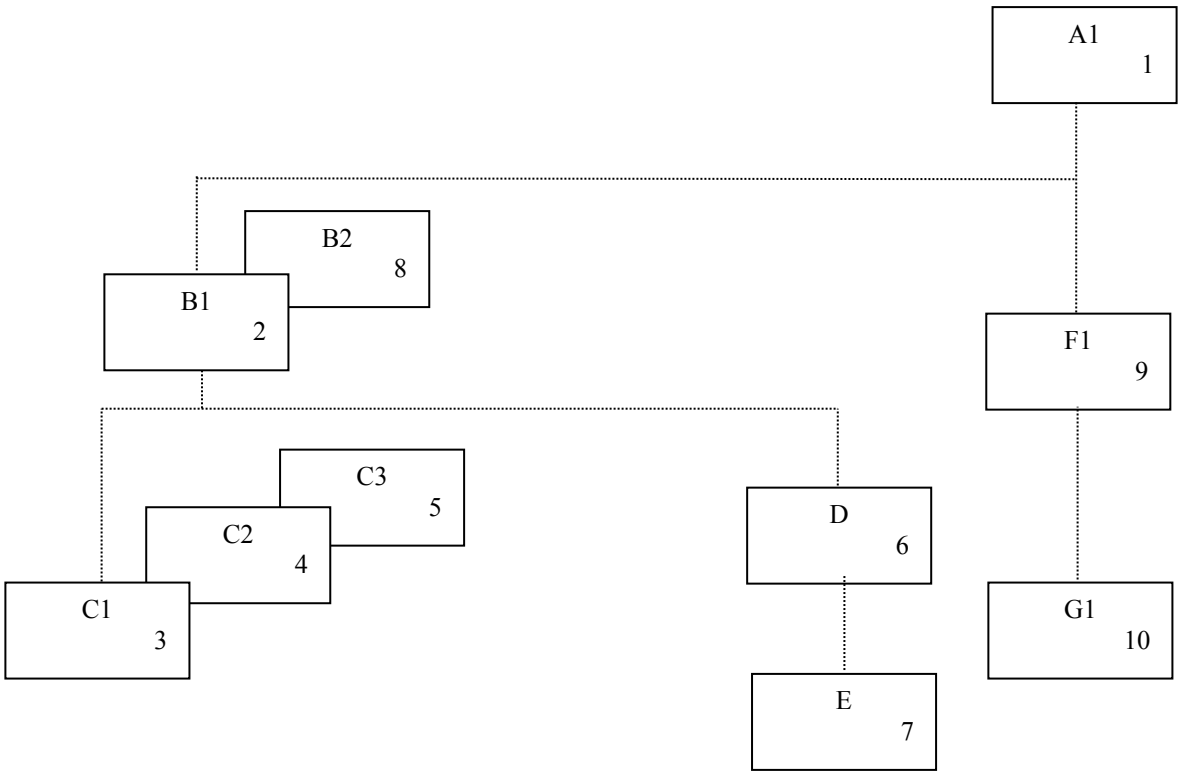
9. Access segment F1.

   Move up one level to segment A1. Because A1 has additional unprocessed segment types, you must move down one level to F1. This is the left-most unprocessed dependent segment of A1.



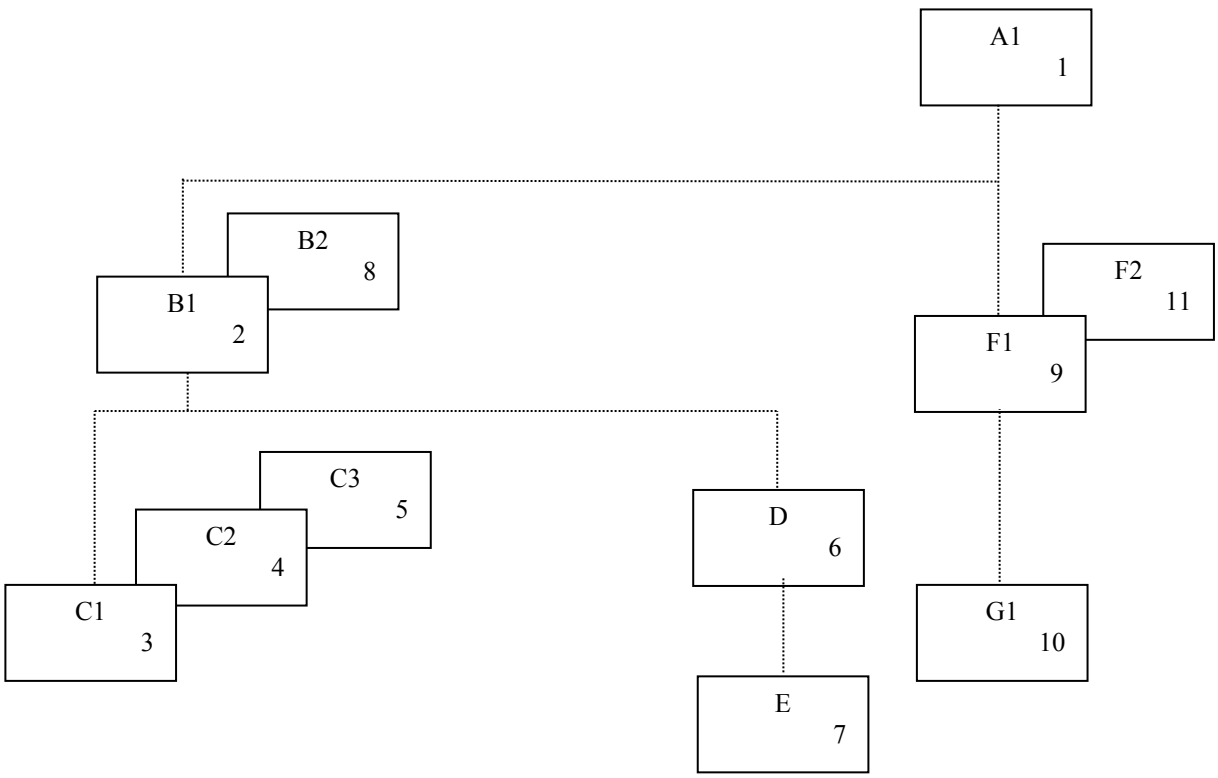
10. Access segment G1.

Search for the left-most unprocessed dependent segment of F1. This is segment G1 (the only dependent of F1).



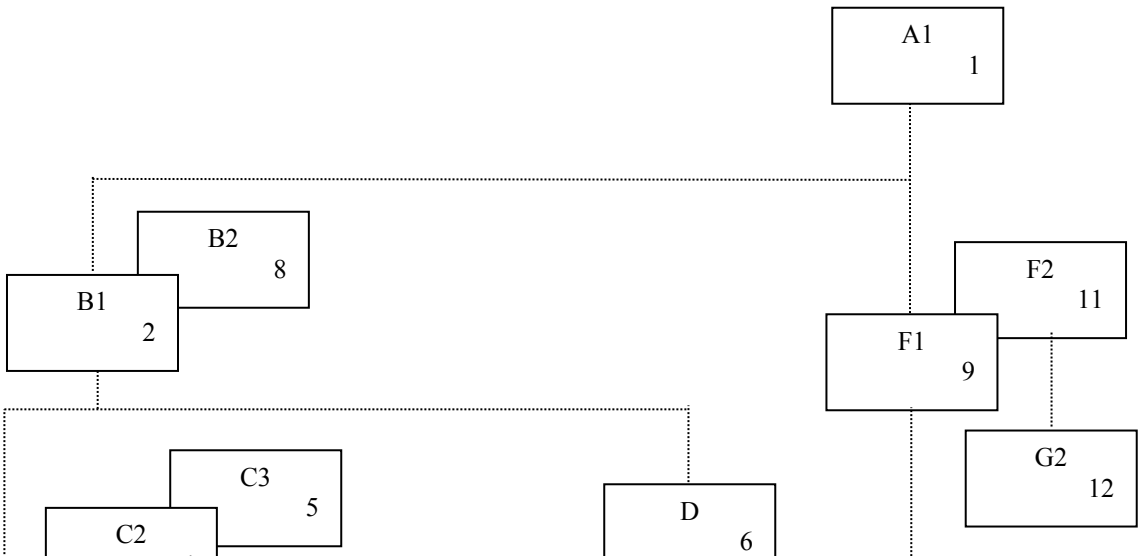
11. Access segment F2.

Segment G1 has no dependents, so, you can return to segment F1. F1 has one twin, so, you must access segment F2.



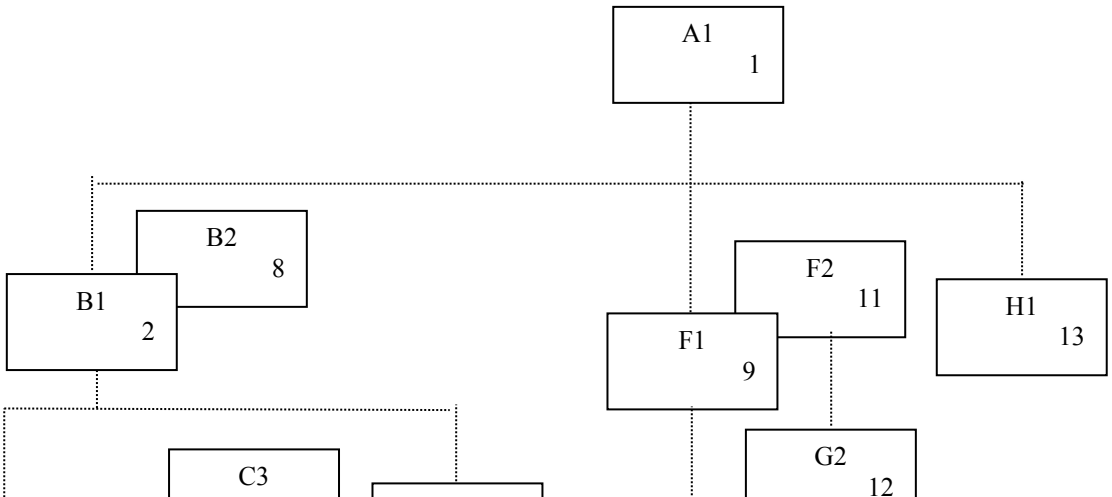
12. Access segment G2.

After processing segment F2, apply rule one and move down to its dependent, segment G2. Segment G2 has no dependents or twins, so you can proceed to segment H1.



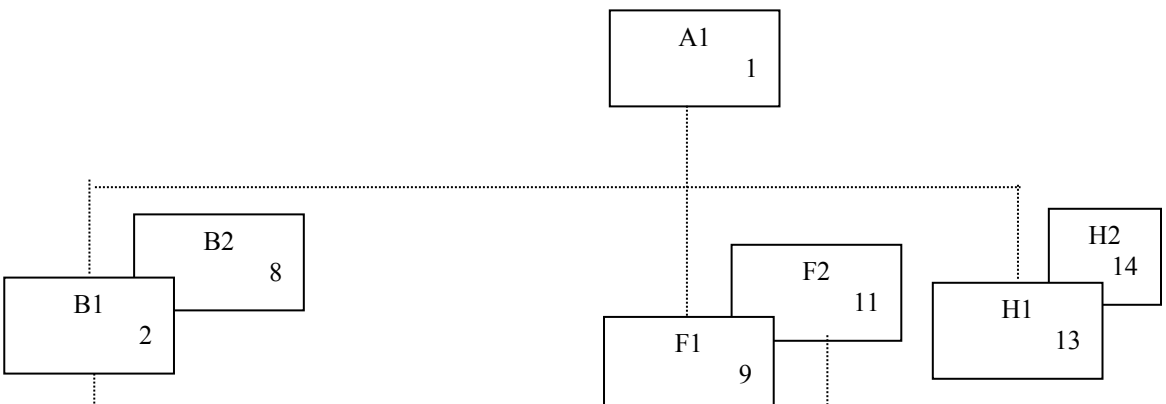
13. Access segment H1.

Neither segment G2 nor F2 has anymore dependents or twins. Return to the top of the hierarchy (segment A1). Apply rule one and access segment H1.



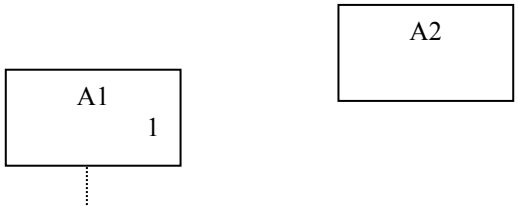
14. Access segment H2.

Because segment H1 has no dependents, check to see if it has any twins. Segment H1 has one twin. Access segment H2.



15. Access segment A2.

All dependent segments for A1 have been processed Check for another occurrence of segment type A. There is one more occurrence of A. Access segment A2. After processing segment A2, segment A has no more dependents or occurrences. Your sequence is complete for this hierarchical structure.



Paths

You should have a clear understanding of how to process both simple and complex hierarchical structures.

The term leg was used in connection with the processing of hierarchical structures, this is also called a path. In this context, leg is synonymous with the more correct term, path. IMS processes hierarchical structures using paths. Paths always begin at the root segment and end at the lowest level of a hierarchical structure.

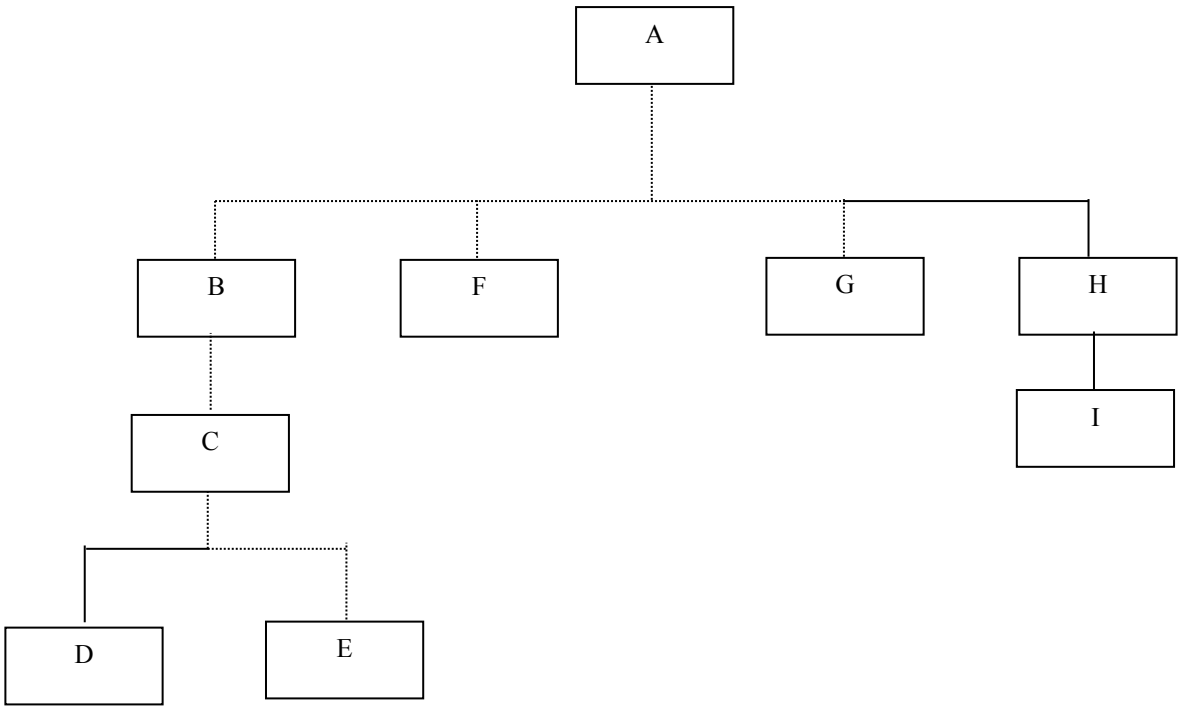


Figure above shows two (of many) paths, both marked with dashes. The first path consists of segments A, B, C and E The second path is made up of segments A and G. Segments A, B and G do not make up a valid path because segment G is not the next segment in hierarchic sequence. Segment B, F, G and H also make up an invalid path because the root is not included, and the segments are not in hierarchic sequence. The Figure above contains five valid paths. They are;

1. A, B, C, D

2. A, B, C, E

3. A, F

4. A, G

5. A, H, I

Summary

Accessing hierarchical data structures in correct sequence is guided by three rules.

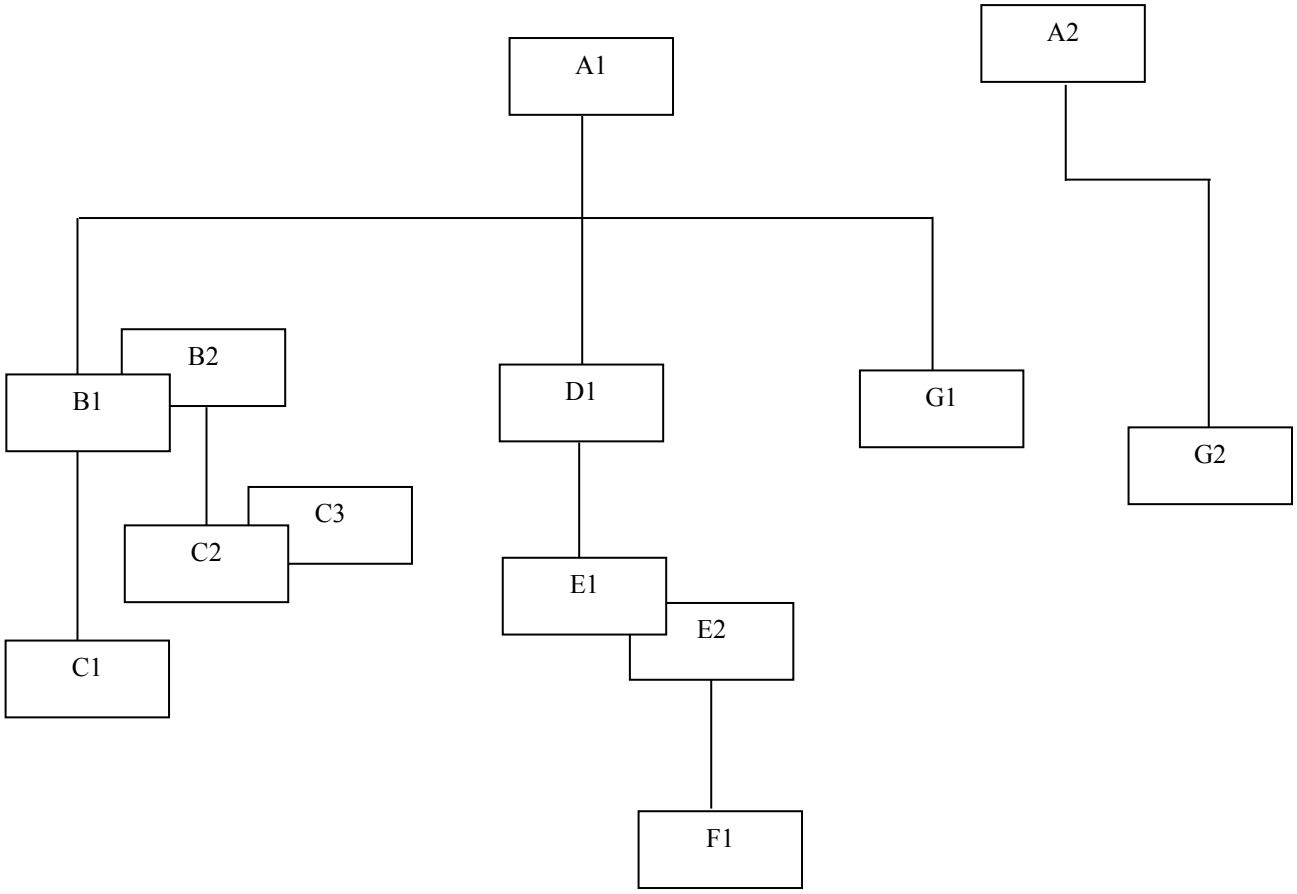
- Top to bottom. Determine whether the segment you are presently accessing has any unprocessed dependents.
- Left to right. If the segment you are accessing has dependents, you process the left-most unprocessed dependent next.
- Front to back. If the segment you are accessing has no dependents (or has no more unprocessed dependents), check to see if it has twins. If it does, process the twins next in a front-to-back sequence.

At any point, all three rules must be applied before going no to another segment. When you have satisfied rule three, you will have finished one hierarchic path in the structure. Then move up one level in the structure and reapply the rules.

It is important to remember that you have completed a separate hierarchical path each time you access a final segment at the bottom of a hierarchical structure.

Hierarchical Processing Exercise

Put a number inside each segment to indicate the sequential processing of the hierarchical structure..



IMS Environments

Index

Your application program can execute in different IMS environments. The three online environments are:

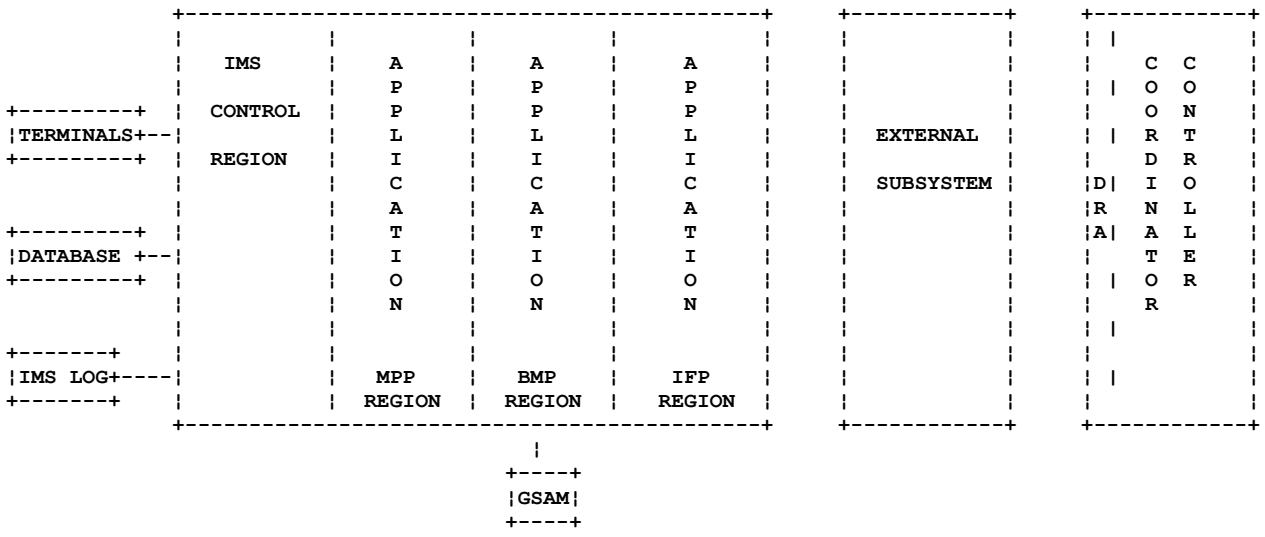
- DB/DC
- DBCTL
- DCCTL

The two batch environments are:

- DB Batch, which is generated from DB/DC and DBCTL class system generations.
- TM Batch, which is generated from DCCTL class system generations.

DB/DC Environment

An application program must reside in a dependent region of IMS. Figure below shows how an application can be positioned in a DB/DC environment.



Application View of DB/DC Environment

MPP=Message Processing Program region
BMP=Batch Message Processing Program region
IFP=IMS Fast Path region

Messages from terminals and application programs are processed by the IMS control region. The requested database segments or messages from other parts of the system (for example, status codes, system messages, or responses from terminals) are also processed by IMS and returned to the application.

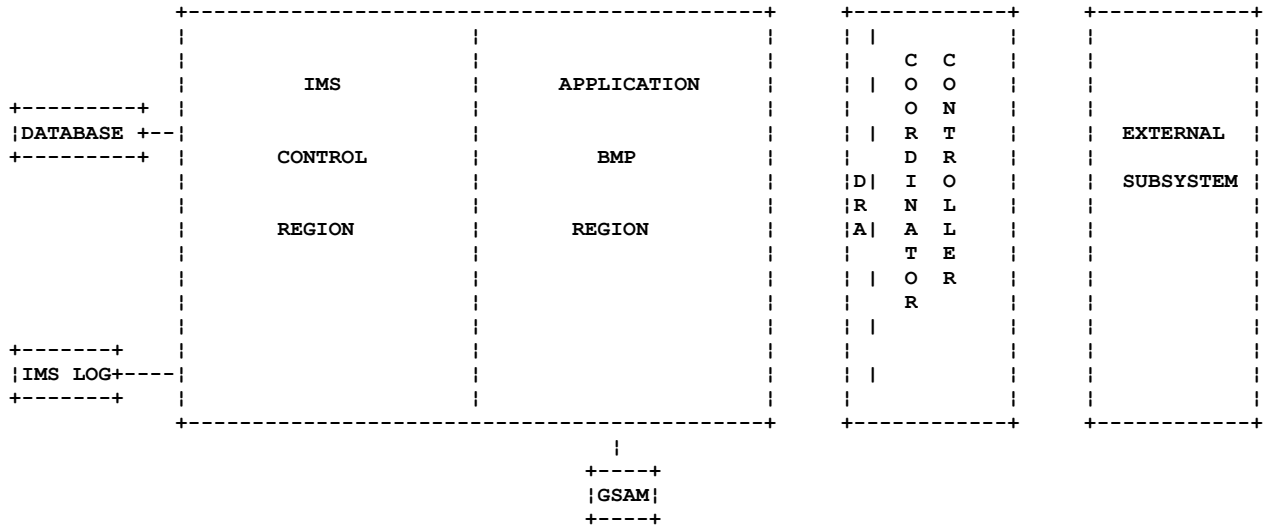
The online environment can be used to access other types of external subsystems using the External Subsystem Attach Facility (ESAF). It permits applications running with IMS to obtain data from external subsystems such as DB2.

The IMS DB portion of the IMS DB/DC environment can be used separately to provide database management for coordinator controllers (CCTLs). The IMS DB portion is termed the DBCTL environment.

The DBCTL Environment

DBCTL functions in the same manner as IMS DB in a DB/DC environment, but it has no inherent communications facilities. The communications and transaction management services are provided by a CCTL. A CCTL consists of the database resource adapter (DRA) and a transaction management subsystem, such as CICS. The DRA resides in the same address space as the transaction manager, allowing communication between the DBCTL environment and the "connected" transaction management subsystem.

The CCTL handles message traffic and schedules applications outside the DBCTL environment, and passes database calls through the DRA to DBCTL. DBCTL processes the DL/I call and returns the information to the CCTL through the DRA.



Application View of the DBCTL Environment

IMS application programs in the DBCTL environment can execute in non-message driven BMP regions. Application programs for DBCTL are the same as IMS DB applications. However, DBCTL applications cannot issue DL/I calls for communications or access MSDB databases. DBCTL BMPs may access DL/I, DEDB, and GSAM databases.

The DBCTL environment can also be used to attach to an external subsystem, such as DB2, using the External Subsystem Attach facility (ESAF). The DBCTL environment's ability to attach to external subsystems provides a BMP access to DB2 databases. Programs running under a CCTL do not have access to external subsystems or GSAM through the DRA interface.

The DB Batch Environment

DB Batch is the batch environment generated from DB/DC or DBCTL system generations. The DB Batch environment has a single address space that contains both IMS code and the application program. DB Batch application programs have access to DL/I and GSAM databases.

Access Methods Under IMS

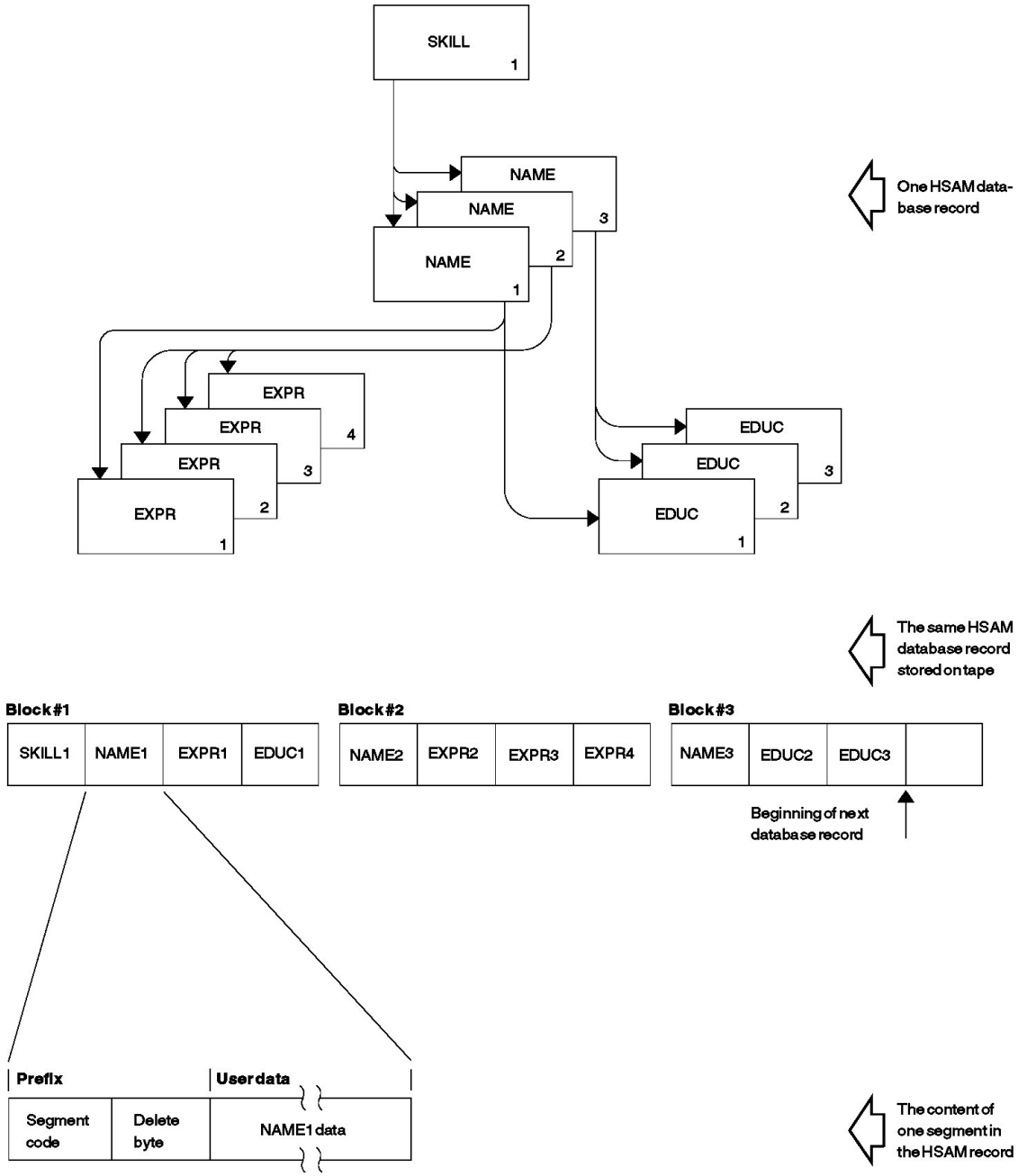
Index

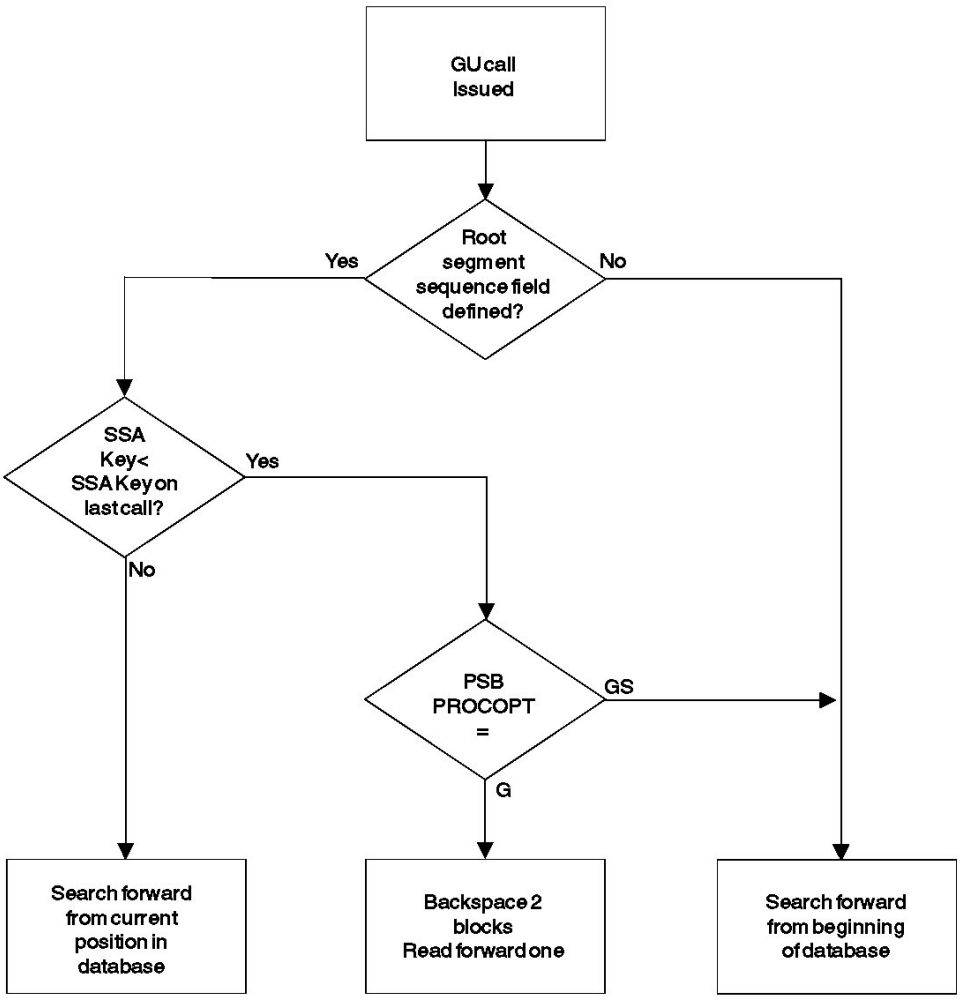
HSAM Databases

1) Hierarchical sequential access method (HSAM) databases use the sequential method of accessing data.
2) All database records and all segments within each database record are physically adjacent in storage.
3) An HSAM database can be stored on tape or on a direct-access storage device.
4) They are processed using either batch sequential access method (BSAM) or queued sequential access method (QSAM) specified by the PROCOPT= parameter in the PCB.
   - PROCOPT=GS for QSAM
   - PROCOPT=G for BSAM for batch
   - QSAM is always used when the system is online
5) HSAM data sets are loaded with root segments in ascending key sequence (if keys exist for the root) and dependent segments in hierarchic sequence.
6) Key field in root segments are not mandatory.
7) Segments must be presented to the load program in the order in which they must be loaded. HSAM data sets use a fixed-length, unblocked record format (RECFM=F)
8) HSAM databases can only be updated by rewriting them. Delete (DLET) and replace (REPL) calls are not allowed, and insert (ISRT) calls are only allowed when the database is being loaded. Although HSAM databases can use the field-level sensitivity option, they cannot use any of the following options:
   - Logical relationships
   - Secondary indexing
   - Variable-length segments
   - Logging, recovery, or reorganization
9) Multiple positioning and multiple PCBs cannot be used in HSAM databases.

When to Use HSAM

1) Uses of HSAM are limited
2) Used for applications requiring sequential processing only.
3) HSAM is used for low-use files like audit trails, statistical reports historical or archive data that has been purged from the main database.

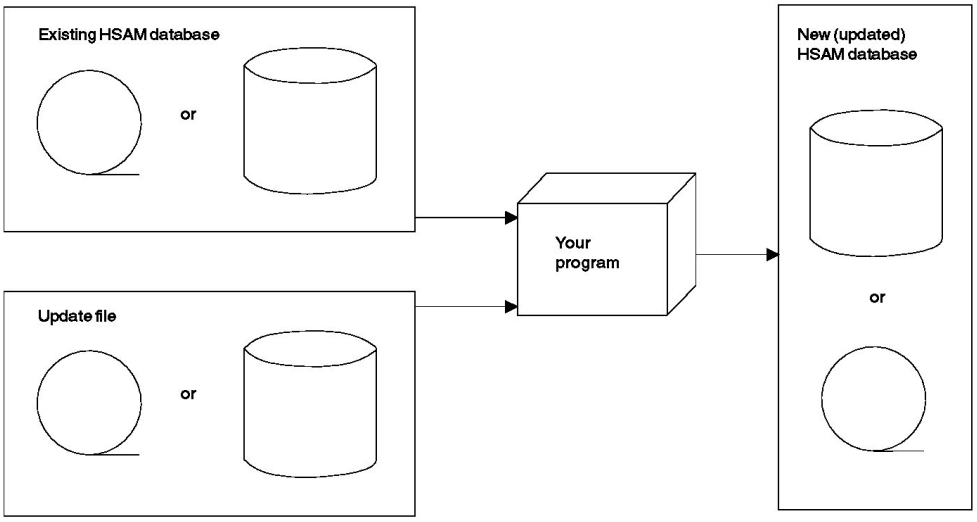




- For GU Calls against an HSAM Database where no sequence field has been defined, each GU call causes the search for the desired segment to start at the beginning of the database regardless of current position.

- This allows direct processing, although inefficiently, of the HSAM database. The processing, however, is restricted to one volume.

Fig below displays updating an HSAM Database



HISAM Databases

1) In a hierarchical indexed sequential access method (HISAM) database, as with an HSAM database, segments in each database record are related through physical adjacency in storage. Each HISAM database record is indexed, allowing direct access to a database record.
2) A unique sequence field in each root segment is mandatory.
3) These sequence fields are then used to construct an index to root segments (and therefore database records) in the database.
4) HISAM databases are stored on direct-access devices.
5) The virtual storage access method (VSAM) is used.
6) Unlike HSAM, all DL/I calls can be issued against a HISAM database. In addition, the following options are available for HISAM databases:
   - Logical relationships
   - Secondary indexing
   - Variable-length segments
   - Field-level sensitivity
   - Logging, recovery, and reorganization

When to Use HISAM

1) Used for databases that require direct access to database records and / or sequential processing of segments in a database record.

2) It is a good candidate for databases with the following characteristics:
   - Most database records are about the same size.
   - The database does not consist of relatively few root segments and a large number of dependent segments.
   - Applications do not depend on a heavy volume of root segments being inserted after the database is initially loaded.
   - Deletion of database records is minimal.

How a HISAM Record Is Stored

- HISAM database records are stored in two data sets.
  (a) The first data set (KSDS), called the primary data set, contains an index and all segments in a database record that can fit in one logical record. The index provides direct access to the root segment (and therefore to database records).
  (b) The second data set, called the overflow data set(ESDS), contains all segments in the database record that cannot fit in the primary data set.

- Record Lengths
  (a) You define the logical record length of both the primary and overflow data set.
  (b) The logical record length can be different for each data set.
  (c) Define the logical record length in the primary data set as large enough to hold an "average" database record or the most frequently accessed segments in the database record.

(d) Logical record length in the overflow data set can then be defined as whatever is most efficient given the characteristics of your database records.

- Logical records are grouped into control intervals (CIs). A control interval is the unit of data transferred between an I/O device and storage. You define the size of CIs.

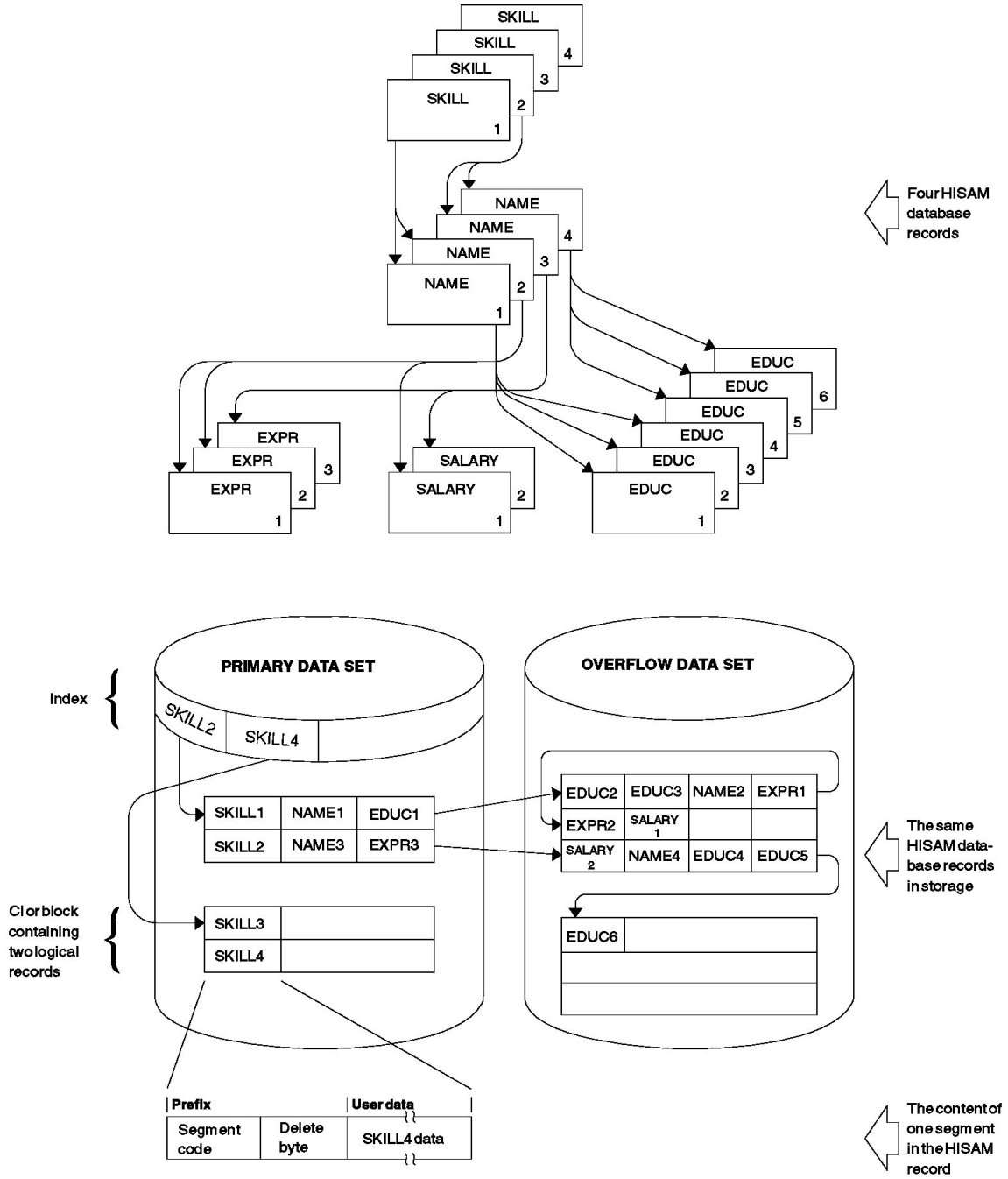
- Each database record starts at the beginning of a logical record in the primary data set. A database record can only occupy one logical record in the primary data set, but overflow segments of the database record can occupy more than one logical record in the overflow data set.

- Segments in a database record cannot be split and stored across two logical records. Because of this and because each database record starts a new logical record, unused space exists at the end of many logical records. When the database is initially loaded, IMS inserts a root segment with a key of all X'FF's as the last root segment in the database.

See following Figure which shows four HISAM database records as they are initially stored on the primary and overflow data sets.

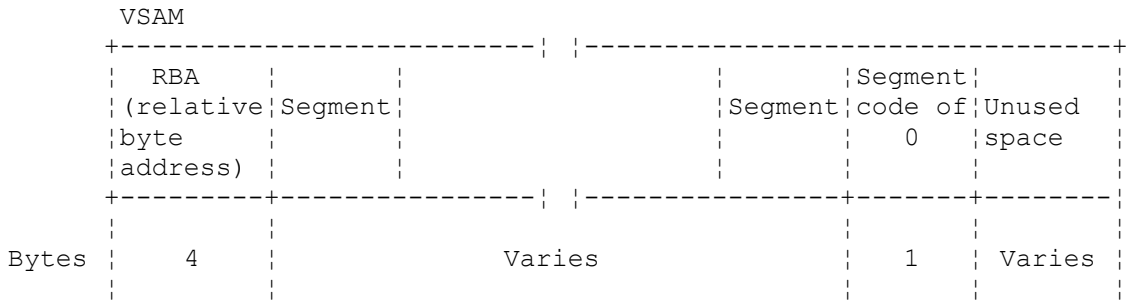
- Prefixes: In storage, a HISAM segment consists of a 2-byte prefix followed by user data.
  (a) The first byte of the prefix is the segment code (1 to 255), which identifies the segment type to IMS. This number can be from 1 to 255.
  (b) The segment code is assigned to the segment by IMS in ascending sequence, starting with the root segment and continuing through all dependents in hierarchic sequence.
  (c) The second byte of the prefix is the delete byte.

- Storage and chaining
  (a) Each logical record in the primary data set contains, in hierarchic sequence, the root plus all dependents of the root for which there's enough space.
  (b) The remaining segments of the database record are put in the overflow data set, again in hierarchic sequence.
  (c) The two "parts" of the database record are chained together with a direct-address pointer.
  (d) When overflow segments in a database record use more than one logical record in the overflow data set (the case for the first and second database record in following Figure), the logical records are also chained together with a direct-address pointer.
  (e) Note in the figure that HISAM indexes do not contain a pointer to each root segment in the database. Rather, they point to the highest root key in each block or CI.



HISAM DATA STORAGE STRUCTURE

- Format of a Logical Record in a HISAM Database



(a) In a VSAM logical record, the first 4 bytes are a direct-address pointer to the next logical record in the database record. This pointer maintains all logical records in a database record in correct sequence. The last logical record in a database record contains zeros in this field.

(b) Following the pointer are one or more segments of the database record in hierarchic sequence.

(c) Following the segments is a 1-byte segment code of 0. It says that the last segment in the logical record has been reached.

- Accessing Segments : In HISAM, when an application program issues a call with an SSA qualified on the key of the root segment, the segment is found by:

(a) Searching the index for the first pointer with a value greater than or equal to the specified root key (the index points to the highest root key in each CI)

(b) Following the index pointer to the correct CI

(c) Searching this CI for the correct logical record (the root key value is compared with each root key in the CI)

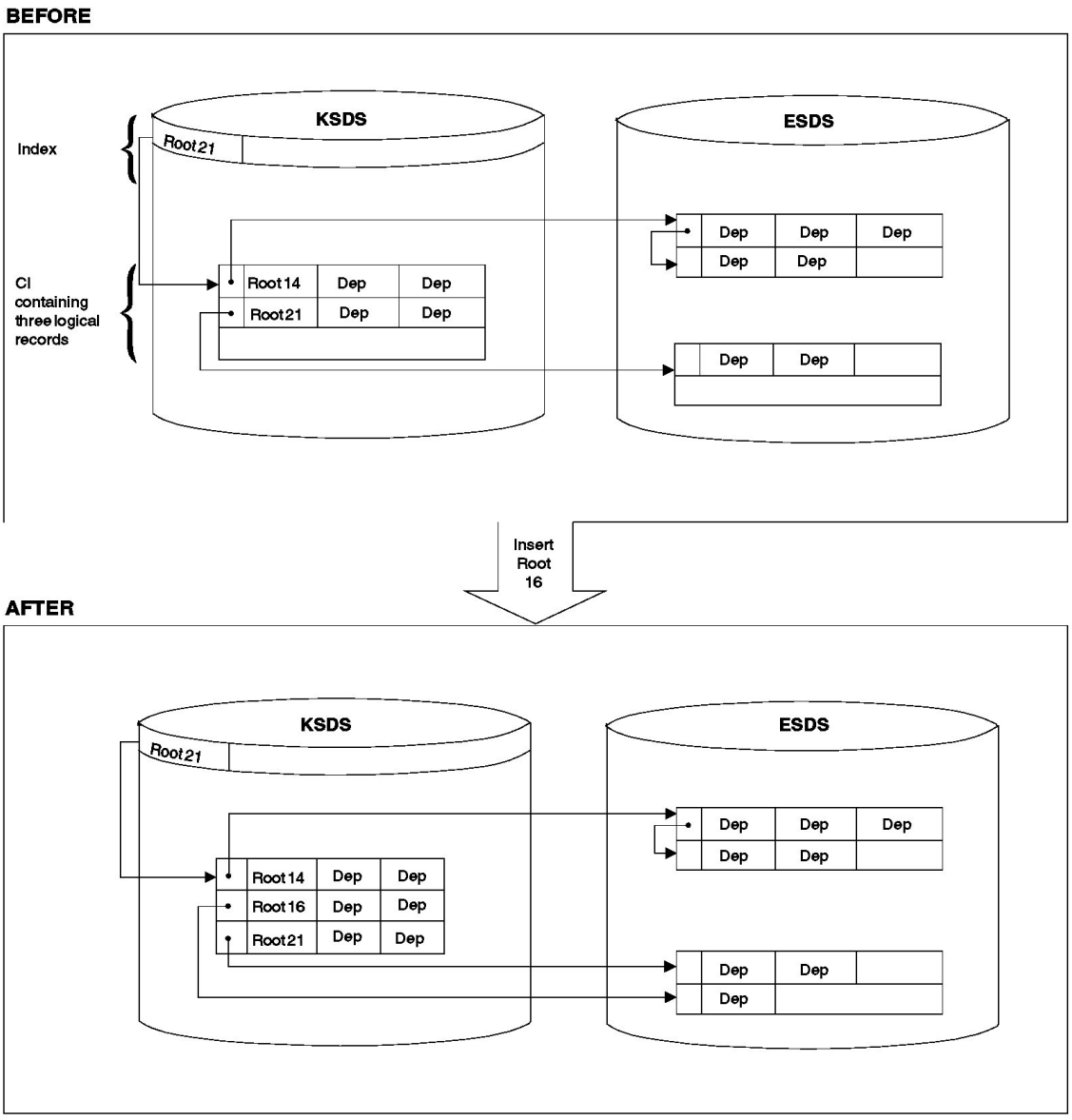
(d) When the correct logical record (and therefore database record) is found, searching sequentially through it for the specified segment.

- If an application program issues a GU call with an unqualified SSA for a root segment or with an SSA qualified on other than the root key, the HISAM index cannot be used. The search for the segment starts at the beginning of the database and proceeds sequentially until the specified segment is found.

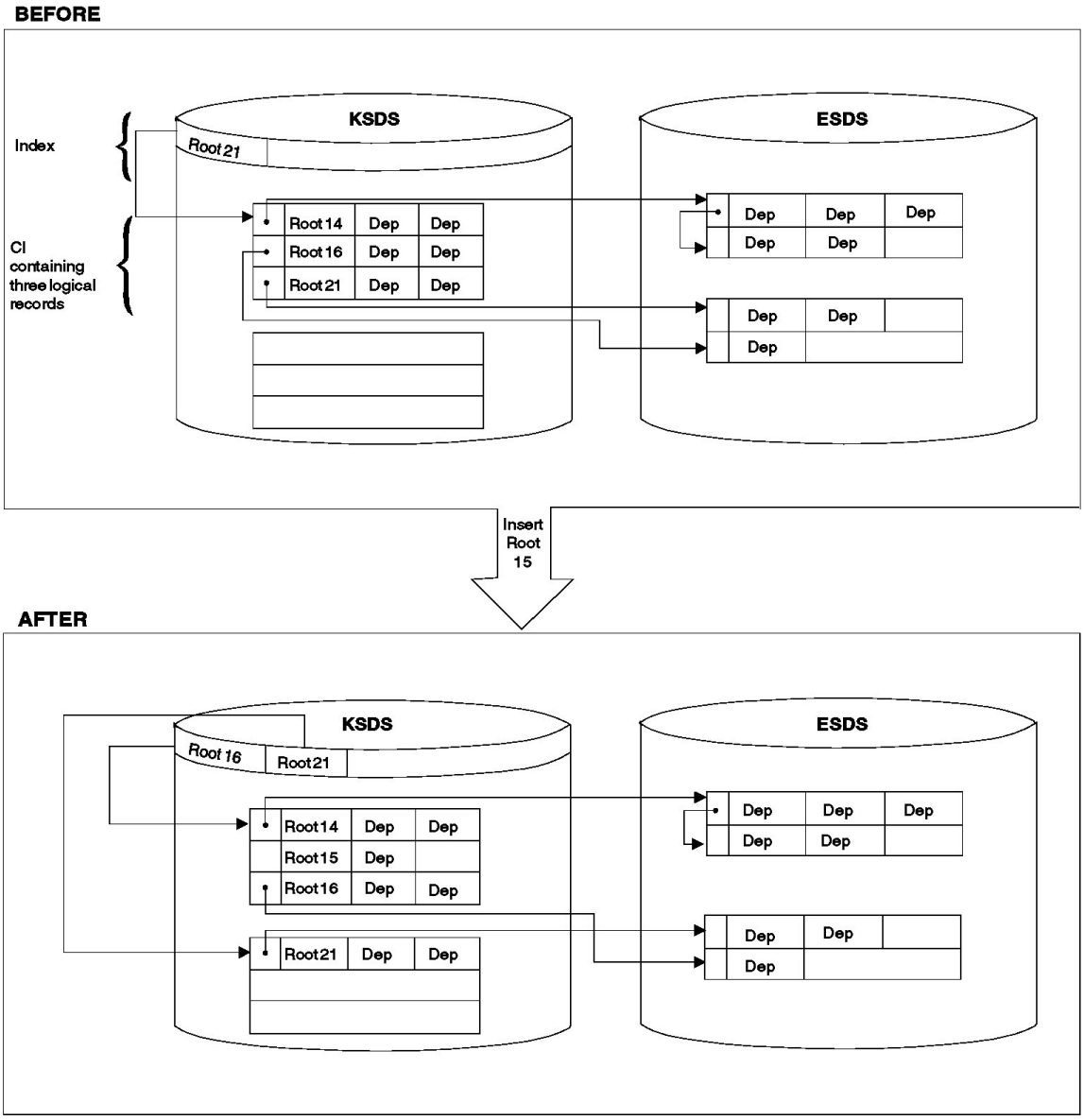
Inserting Root Segments Using VSAM

After an initial load, root segments inserted into a HISAM database are stored in the primary data set in ascending key sequence. The CI might or might not contain a free logical record into which the new root can be inserted. Both situations are described next.

The following figure depicts the situation when space exists for the additional record in the CI



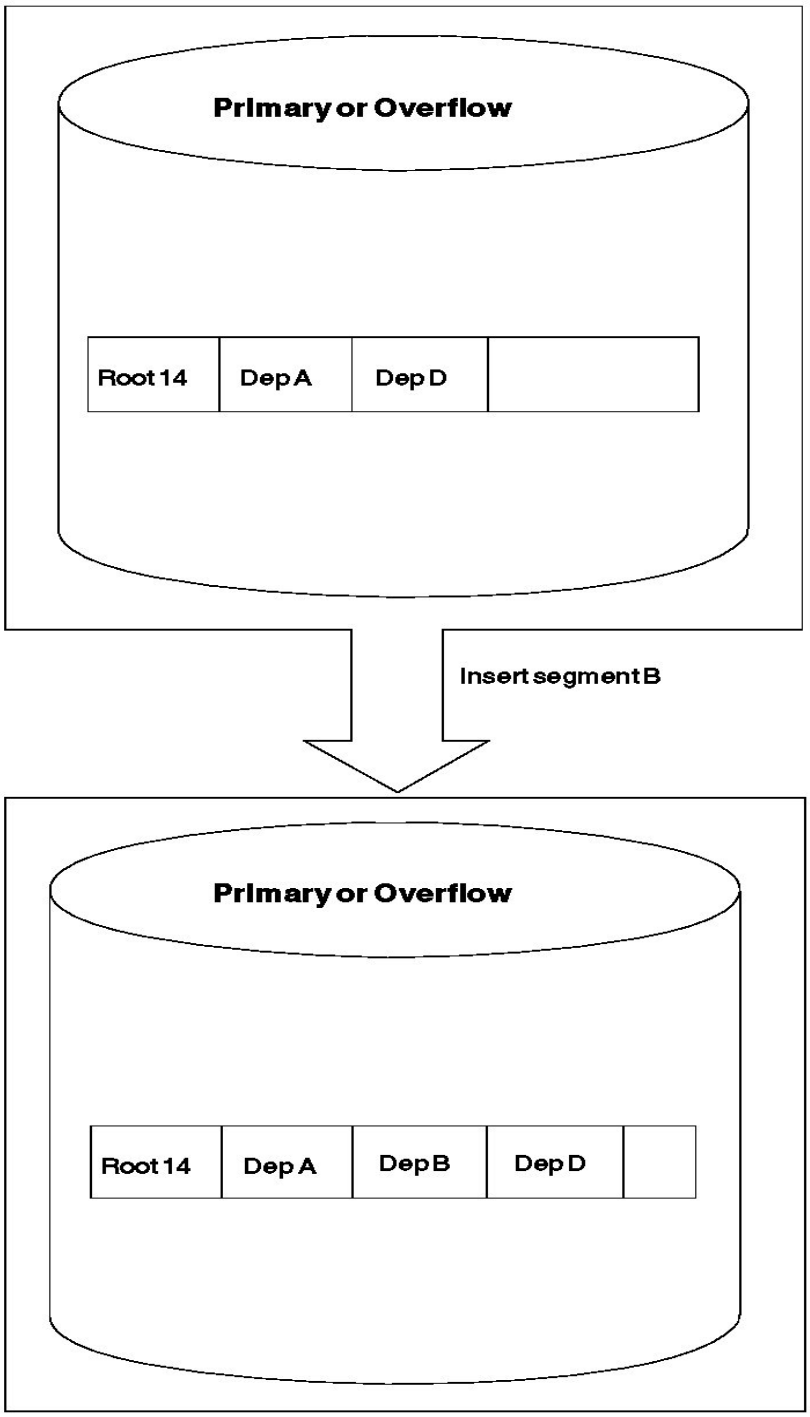
When No free logical record exists the following figure shows how insertion takes place when no free logical record exists in the CI. The CI is split forming two new CIs, both equal in size to the original one.



Inserting a Root Segment into a HISAM Database (No Free Logical Record Exists in the Control Interval)

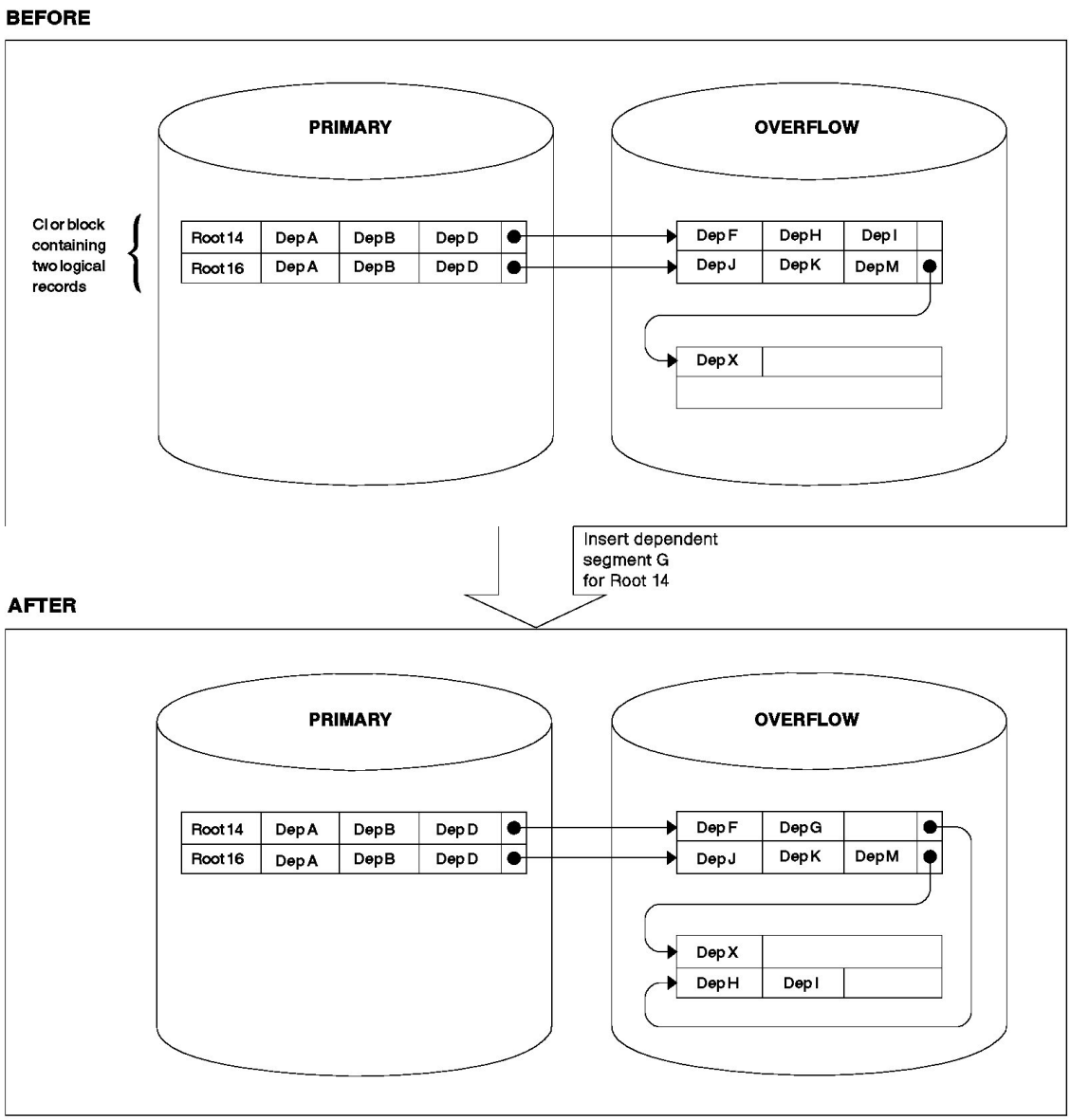
- Inserting Dependent Segments : Dependent segments inserted into a HISAM database after initial load are inserted in hierarchic sequence. IMS decides where in the appropriate logical record the new dependent should be inserted. Two situations are possible. Either there is enough space in the logical record for the new dependent or there is not.

(a) Insertion when there is enough space in the logical record. The new dependent is stored in its proper hierarchic position in the logical record by shifting the segments that hierarchically follow it to the right in the logical record.



Inserting a Dependent Segment into a HISAM Database (Space Exists in the Logical Record)

When no space exists in current record for the insertion



Inserting a Dependent Segment into a HISAM Database (No Space Exists in the Logical Record)

Deleting Segments

(a) When segments are deleted from a HISAM database, they are marked as deleted in the delete byte in their prefix.

(b) If the deleted segment is a root, the logical record containing the root is erased, provided neither the root nor any of its dependents is involved in a logical relationship.

(c) After the logical record is removed, its space is available for reuse. However, any overflow logical record containing dependents of this root is not available for reuse. Except for this

special condition, you must unload and reload a HISAM database to regain space occupied by deleted segments.

Replacing Segments

Replacing segments in a HISAM database is straightforward as long as fixed length segments are being used. The data in the segment, once changed, is returned to its original location in storage. The key field in a segment cannot be changed**.**

Criteria for Selecting HISAM

You should use HISAM when you need sequential or direct access to roots and sequential processing of dependent segments in a database record. HISAM is a good choice of data organization when your database has most, if not all, of the following characteristics.

(a) Each root has few dependents. Root segment access is indexed, and is therefore fast. Dependent segment access is sequential, and is therefore slower.

(b) You have a small number of delete operations against the database. Except for deleting root segments, all delete operations result in space that is unusable until the database is reorganized.

(c) Your applications depend on a small volume of root segments being inserted within a narrow key range (VSAM). Root segments inserted after initial load are inserted in root key sequence in the appropriate CI in the KSDS. If many roots have keys within a narrow key range, many CI splits might occur and decrease performance.

(d) Most of your database records are about the same size. This allows you to pick logical record lengths and CI sizes so most database records fit on the primary data set. You want most database records to fit on the primary data set, because additional read and seek operations are required to access those parts of a database record on the overflow data set.

(e) Having most of your database records be the same size also saves space. Each database record starts at the beginning of a logical record. All space in the logical records not used by the database record is unusable space. This is true of logical records in both the primary and overflow data set. If the size of your database records varies tremendously, large gaps of unused space can occur at the end of many logical records.

SHSAM, SHISAM, and GSAM Databases

You typically use simple hierarchical sequential access method (SHSAM), simple hierarchical indexed sequential access method (SHISAM), and generalized sequential access method (GSAM) databases in two situations.

- Situation 1 - Converting from a non database system to IMS
  SHSAM, SHISAM, and GSAM databases allow existing programs, using MVS access methods, to remain usable during the conversion to IMS. This is possible because the format of the data in these databases is the same as it is in MVS data sets.

- Situation 2 - Passing data
  When a database (or non-database) application program passes data to a database (or non-database) application program, it first puts the data in a SHSAM, SHISAM, or GSAM database. The database (or non-database) application program then accesses the data from these databases.

- SHSAM Databases
  A simple HSAM (SHSAM) database is an HSAM database containing only one type of segment, a root segment. The segment has no prefix, because no need exists for a segment code (there is only one segment type) and for a delete byte (deletes are not allowed).

  (a) Because SHSAM segments contain user data only (no IMS prefixes), they can be accessed by the MVS access methods BSAM and QSAM.

  (b) The ISRT, DLET, and REPL calls cannot be used to update. However, ISRT can be used to load a SHSAM database.

  (c) The only valid calls for processing a SHSAM database are the get calls, which only allow retrieval of segments from the database.

  (d) To update an SHSAM database, it must be reloaded.

  (e) Unlike SHSAM, however, GSAM files cannot be accessed from a message processing region. GSAM does allow you to take checkpoints and perform restart.

  (f) Although SHSAM databases can use the field-level sensitivity option, they cannot use any of the following options:

    - Logical relationships
    - Secondary indexing
    - Variable-length segments
    - Logging, recovery, or reorganization

- SHISAM Databases
  (a) A simple HISAM (SHISAM) database is a HISAM database containing only one type of segment, a root segment.

(b) The segment has no prefix, because no need exists for a segment code (there is only one segment type) and for a delete byte (deletes are done using a VSAM erase operation).

(c) SHISAM databases must use VSAM as their access method, and the data set used must be a KSDS.

(d) Because SHISAM segments contain user data only (no IMS prefixes), they can be accessed by VSAM macros as well as DL/I calls.

(e) All DL/I calls can be issued against SHISAM databases.

(f) In addition to those situations described in the introduction to this section, SHISAM is useful if you need an application program that accesses MVS data sets to use the IMS symbolic checkpoint call.

(g) SHISAM databases can use field-level sensitivity, but they cannot use any of the following options:

- Logical relationships
- Secondary indexing
- Variable-length segments

GSAM Databases

(a) GSAM databases are sequentially organized databases designed to be compatible with MVS data sets.

(b) GSAM databases can be on a data set previously created or one later accessed by the MVS access methods VSAM or QSAM / BSAM. GSAM data sets can use fixed-length or variable-length records when VSAM is used, or fixed-length, variable-length or undefined-length records when QSAM / BSAM is used.

(c) If VSAM is used to process a GSAM database, the VSAM data set must be entry sequenced and on a DASD.

(d) If QSAM / BSAM is used, the physical sequential (DSORG=PS) data set can be on a DASD, tape, or unit record device.

(e) Because GSAM is designed to be compatible with MVS data sets, the GSAM database has no hierarchy, database records, segments, or keys.

(f) In addition to those situations described in the introduction to this section, ***GSAM is useful if you need an application program that accesses MVS data sets to use the IMS symbolic checkpoint call.***

(g) GSAM databases are loaded in the order in which you present records to the load program. You cannot issue DLET and REPL calls against GSAM databases; however, you can issue ISRT calls after the database is loaded but only to add records to the end of the data set. Records cannot be randomly added to a GSAM data set.

(h) Although random processing of GSAM and SHSAM databases is possible, random processing of a GSAM database is done using a GU call qualified with a record search argument (RSA). This processing is primarily useful for establishing position in the database before issuing a series of GN calls.

(i) ***Although SHSAM and SHISAM databases can be processed in any processing region, GSAM databases can only be processed in a batch or batch message processing region***.

(j) The following IMS options do not apply to GSAM databases:
- Logical relationships
- Secondary indexing
- Field-level sensitivity
- Logging or reorganization

HDAM and HIDAM Databases

Hierarchical direct access method (HDAM) and hierarchical indexed direct access method (HIDAM) databases have many similarities and are referred to as HD databases.

(a) HD databases differ from sequentially organized databases in two important ways.
- They use the direct method of storing data, and the ***hierarchic sequence of segments in the database is maintained by having segments point to one another***. Except for a few special cases, each segment has one or more direct-address pointers in its prefix. When direct-address pointers are used, database records and segments can be stored anywhere in the database. Their position, once stored, is fixed, and they do not "move around" in the database when subsequent processing takes place. Instead, pointers are updated to reflect processing changes.

- ***HD databases also differ from sequentially organized ones in that space in HD databases can be reused***. If part or all of a database record is deleted, the deleted space can be reused when new database records or segments are inserted.

(b) HD databases are stored on direct-access devices in a VSAM ESDS (HDAM) or KSDS and ESDS (HIDAM).

(c) ***In HDAM, each root segment's storage location is found using a randomizing module. The randomizing module examines the root's key and, through an arithmetic technique, computes the address of a pointer to the root segment(In terms of CI and Root Anchor Point).***

(d) ***In HIDAM, each root segment's storage location is found by searching an index***. This index, unlike the index used with HISAM, requires use of a second database, an index database.

(e) IMS loads and maintains this index database.

(f) The advantage of the HDAM randomizing module is that the I/O operations required to search an index are eliminated. In addition, no need exists to update an index after a root segment is inserted or deleted.

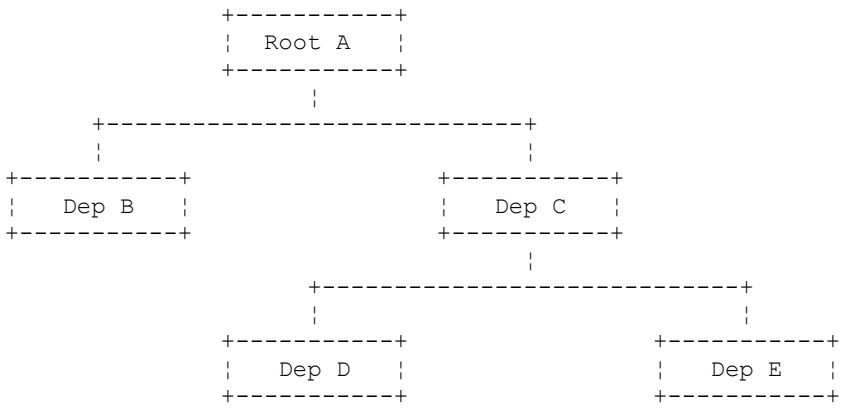
(g) The ***disadvantage of HDAM is that root segments cannot be accessed in key sequence***

(h) All DL/I calls can be issued against HD databases. In addition, the following options are available:
- Logical relationships
- Secondary indexing
- Variable-length segments
- Field-level sensitivity
- Logging, recovery, and reorganization

When to Use HDAM

(a) HDAM databases are typically used when you need primarily direct access to database records. The randomizing module provides fast access to the root segment (and therefore the database record).

(b) HDAM databases also give you fast access to paths of segments as specified in the DBD in a database record. For example, in the following database record, if physical child pointers are used they can be followed to reach segments B, C, D, or E. A hierarchic search of segments in the database record is bypassed. Segment B does not need to be accessed to get to segments C, D, or E. And segment D does not need to be accessed to get to segment E. Only segment A must be accessed to get to segment B or C. And only segments A and C must be accessed to get to segments D or E.



(c) You cannot process HDAM database records in key sequence unless the randomizing module you use stores root segments in physical key sequence.

When to Use HIDAM

(a) HIDAM databases are typically used when you need both random and sequential access to database records and random access to paths of segment in a database record.

(b) Access to root segments (and therefore database records) is not as fast as with HDAM, because the HIDAM index database has to be searched for a root segment's address.

Types of Pointers You Can Specify (DBA responsibility)

(a) The hierarchic sequence of segments in a database record using the sequential access methods is maintained by keeping segments physically adjacent to each other in storage.

(b) In the HD access methods, segments in a database record are kept in hierarchic sequence using direct-address pointers.

(c) Each prefix in an HD segment contains one or more pointers. Each pointer is 4 bytes long and consists of the relative byte address of the segment to which it points. Relative, in this case, means relative to the beginning of the data set.

(d) Three basic types of direct-address pointers exist:

- Hierarchic pointers, which point from one segment to the next in either forward or forward and backward hierarchic sequence

- Physical child pointers, which point from a parent to each of its first or first and last children, for each child segment type

- Physical twin pointers, which point forward or forward and backward from one segment occurrence of a segment type to the next, under the same parent

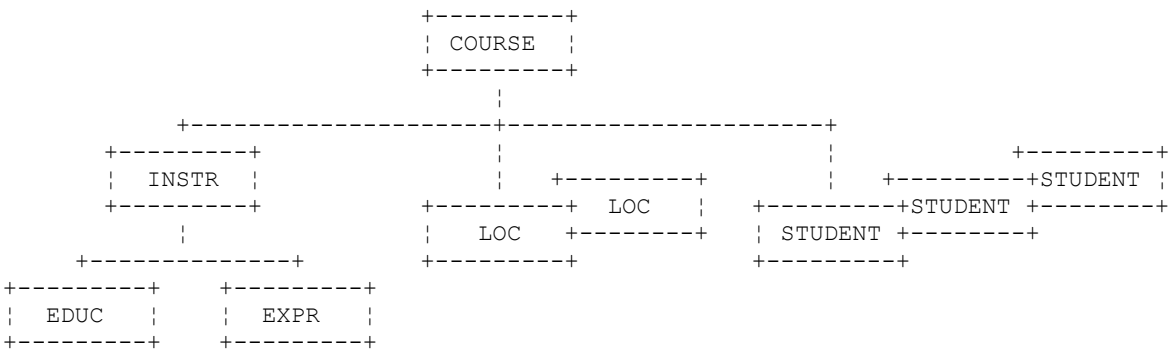
(e) When segments in a database record are typically processed in hierarchic sequence the hierarchic pointers are used.

(f) When segments in a database record are typically processed randomly, a combination of physical child and physical twin pointers.

(g) One thing to keep in mind while reading about pointers is that the different types, subject to some rules, can be mixed within a database record.

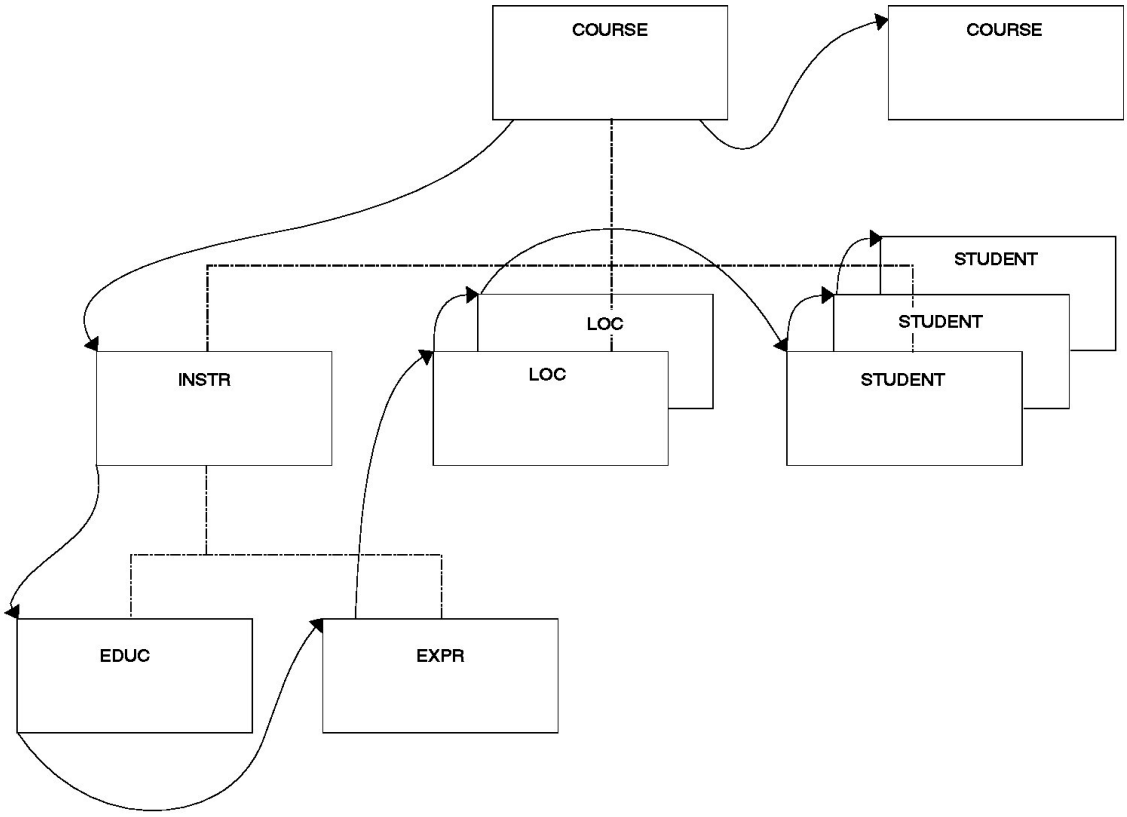
(h) Pointers are specified by segment type and all occurrences of the same segment type have the same type of pointer.

In the following sections, each type of pointer is illustrated, and the database record on which each illustration is based is as follows:



Hierarchic Forward Pointers

With hierarchic forward (HF) pointers, each segment in a database record points to the segment that follows it in the hierarchy. Figure below shows hierarchic forward pointers:



HIERARCHIC FORWARD POINTERS

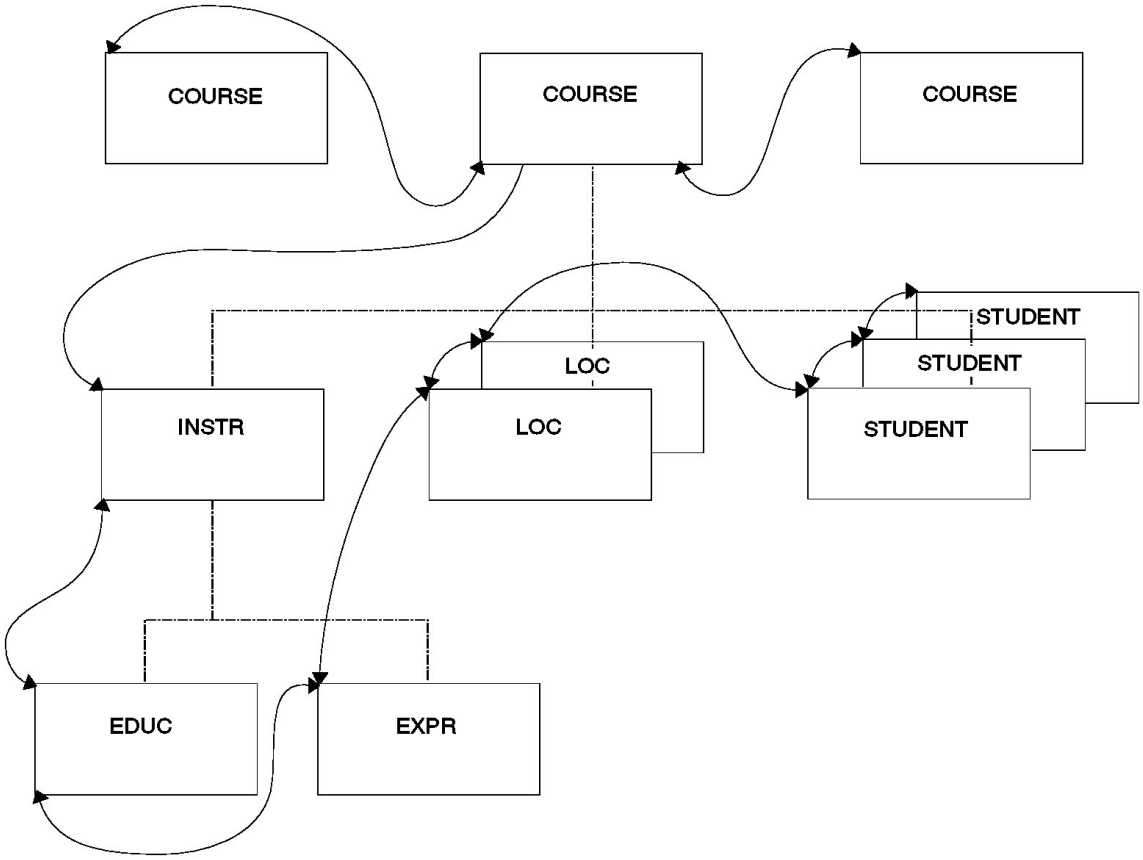
When an application program issues a call for a segment, HF pointers are followed until the specified segment is found. In this sense, the use of HF pointers in an HD database is similar to using a sequentially organized database. In both, to reach a dependent segment all segments that hierarchically precede it in the database record must be examined. HF pointers should be used when segments in a database record are typically processed in hierarchic sequence and processing does not require a significant number of delete operations. If there are a lot of delete operations, hierarchic forward and backward pointers (explained next) might be a better choice.

Four bytes are needed in each dependent segment's prefix for the HF pointer. Eight bytes are needed in the root segment. *More bytes are needed in the root segment because the root points to both the next root segment and first dependent segment in the database record*.

HF pointers are specified by coding PTR=H in the SEGM statement in the DBD.

Hierarchic Forward and Backward Pointers

With hierarchic forward and backward pointers (HF and HB), each segment in a database record points to both the segment that follows and the one that precedes it in the hierarchy (except dependent segments do not point back to root segments). HF and HB pointers must be used together, since you cannot use HB pointers alone.



(a) HF pointers work in the same way as the HF pointers described in the preceding section.

(b) HB pointers point from a segment to one immediately preceding it in the hierarchy. In most cases, HB pointers are not required for delete processing. IMS saves the location of the previous segment retrieved on the chain and uses this information for delete processing. ***The backward pointers are useful for delete processing if the previous segment on the chain has not been accessed. This happens when the segment to be deleted is entered by a logical relationship***.

(c) The backward pointers are useful only when all of the following are true:

- Direct pointers from logical relationships or secondary indexes point to the segment being deleted or one of its dependent segments.
- These pointers are used to access the segment.
- The segment is deleted.

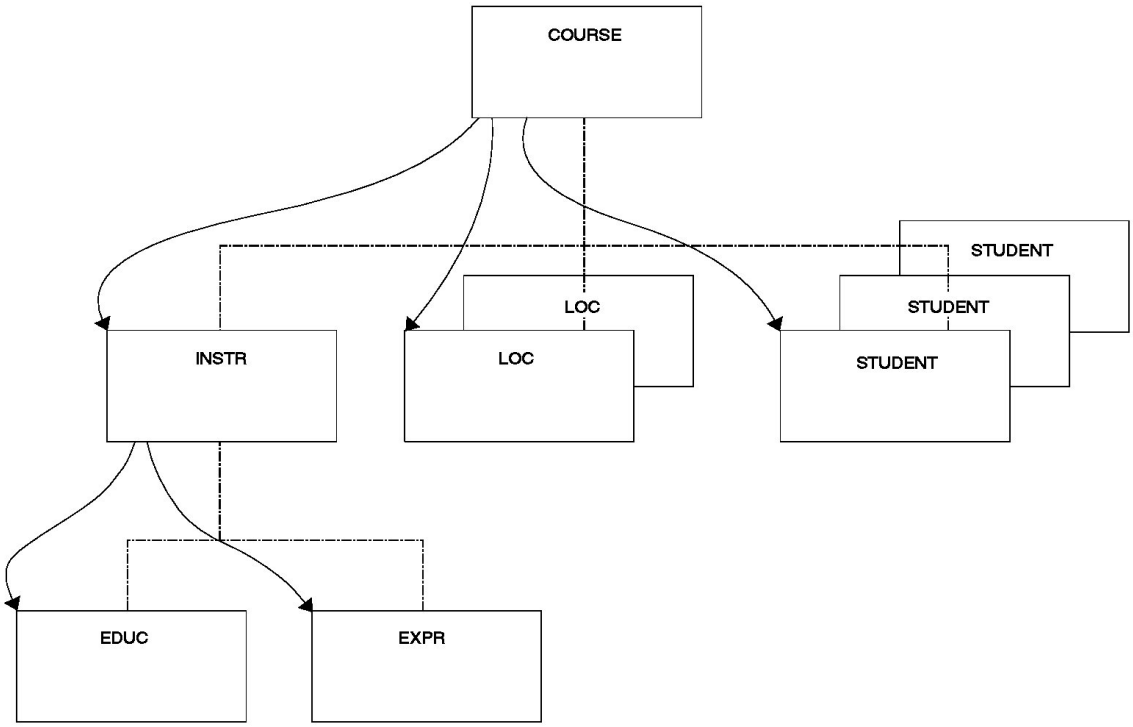
(d) Eight bytes are needed in each dependent segment's prefix to contain HF and HB pointers. Twelve bytes are needed in the root segment. More bytes are needed in the root segment because the root points:

1) Forward to a dependent segment
2) Forward to the next root segment in the database
3) Backward to the preceding root segment in the database

HF and HB pointers are specified by coding PTR=HB in the SEGM statement in the DBD.

Physical Child First Pointers

With physical child first (PCF) pointers, each parent segment in a database record points to the first occurrence of each of its immediately dependent child segment types.



Physical Child First Pointers

a) With PCF pointers, the hierarchy is only partly connected. No pointers exist to connect occurrences of the same segment type under a parent. Physical twin pointers (explained later) can be used to form this connection.

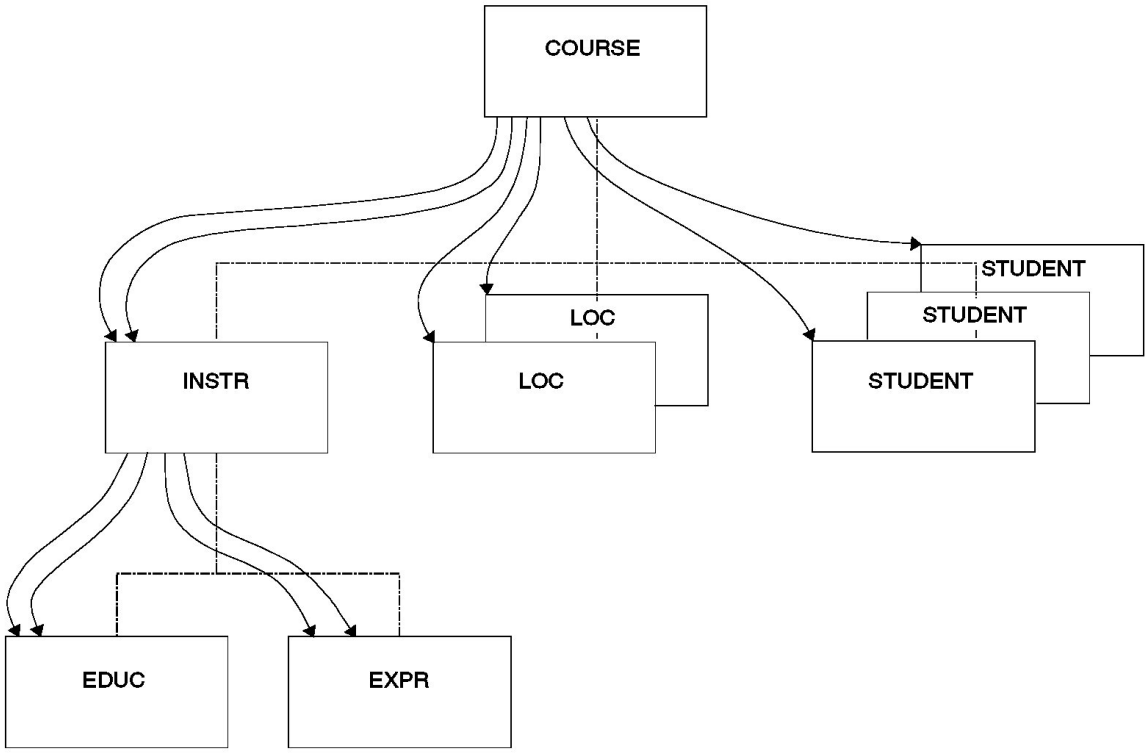
b) PCF pointers should be used when segments in a database record are typically processed randomly and sequence fields are defined for the segment type.

c) If sequence fields are not defined and new segments are inserted at the end of existing segment occurrences, the combination of PCF and physical child last (PCL) pointers (explained next) can be a better choice.

d) Four bytes are needed in each parent segment for each PCF pointer. PCF pointers are specified by coding PARENT=((name,SNGL)) in the SEGM statement in the DBD. This is the SEGM statement for the child being pointed to, not the SEGM statement for the parent. Note, however, that the pointer is stored in the parent segment.

Physical Child First and Last Pointers

With physical child first and last pointers (PCF and PCL), each parent segment in a database record points to both the first and last occurrence of its immediately dependent child segment types. PCF and PCL pointers must be used together, since you cannot use PCL pointers alone.



Physical Child First and Last Pointers

(a) Notice that if only one physical child of a particular parent segment exists, the PCF and PCL pointers both point to the same segment. As with PCF pointers, PCF and PCL pointers leave the hierarchy only partly connected, and no pointers exist to connect occurrences of the same segment type under a parent. Physical twin pointers (explained later) can be used to form this connection.

(b) PCF and PCL pointers (as opposed to just PCF pointers) are typically used when:
- No sequence field is defined for the segment type
- New segment occurrences of a segment type are inserted at the end of all existing segment occurrences.

(c) On insert operations, if the ISRT rule of LAST has been specified, segments are inserted at the end of all existing segment occurrences for that segment type. When PCL pointers are used, fast access to the place where the segment will be inserted is possible. This is because there is no need to search forward through all segment occurrences stored before the last occurrence.

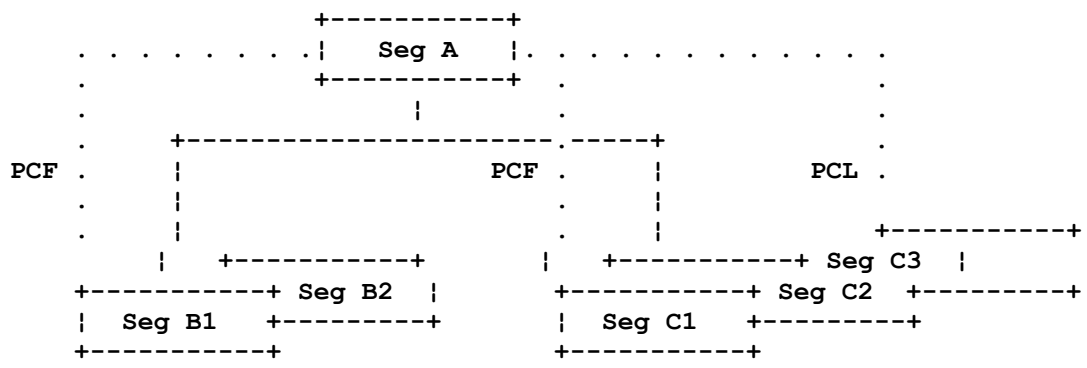
(d) PCL pointers also give application programs fast retrieval of the last segment in a chain of segment occurrences. Application programs can issue calls to retrieve the last segment by using an unqualified SSA with the command code L. When a PCL pointer is followed to get the last segment occurrence, any further movement in the database is forward.

(e) A PCL pointer does not enable you to search from the last to the first occurrence of a series of dependent child segment occurrences.

(f) Four bytes are needed in each parent segment for each PCF and PCL pointer. PCF and PCL pointers are specified by coding the PARENT= operand in the SEGM statement in the DBD as PARENT=((name,DBLE)). This is the SEGM statement for the child being pointed to, not the SEGM statement for the parent. Note, however, that the pointers are stored in the parent segment.

(g) A parent segment can have SNGL specified on one immediately dependent child segment type and DBLE specified on another. Figure below shows specifying PCF and PCL pointers.

```
DBD
SEGM A
SEGM B  PARENT=((name,SNGL))    (specifies PCF pointer only)
SEGM C  PARENT=((name,DBLE))    (specified PCF and PCL pointers)

Results in these pointers being created:
```



Specifying PCF and PCL Pointers

Physical Twin Forward Pointers

(a) With physical twin forward (PTF) pointers, each segment occurrence of a given segment type under the same parent points forward to the next segment occurrence. Figure below illustrates this.

(b) Notice that PTF pointers can be specified for root segments. When this is done in an HDAM database, the root segment points to the next root in the database chained off the same root anchor points (RAP). (RAPs are explained later). If no more root segments are chained from this RAP, the PTF pointer is zero.

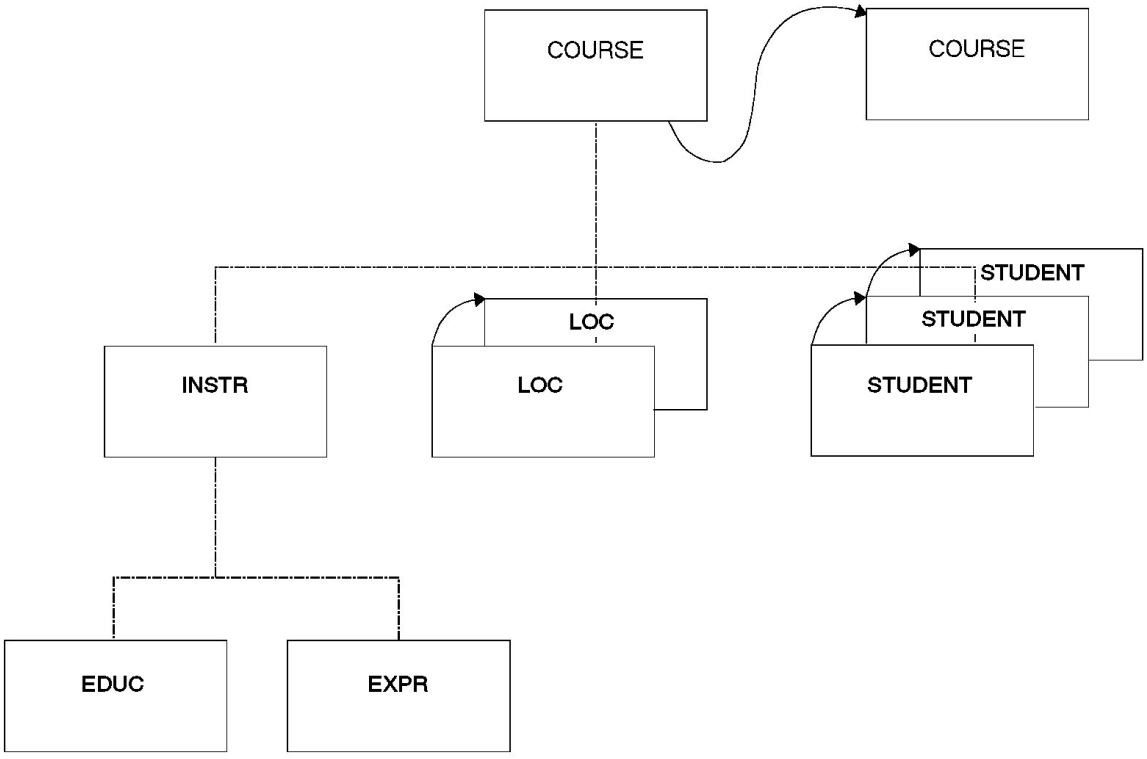
(c) When PTF pointers are specified for root segments in HIDAM database, the root segment does not point to the next root in the database. What happens is explained in a subsequent section called "Use of RAPs in a HIDAM Database." The important thing for you to know now is that if you specify PTF pointers on a root segment in a HIDAM database, the HIDAM index must be used for all sequential processing of root segments. This increases access time. This problem is eliminated if you specify PTF and physical twin backward (PTB) pointers (discussed next).

(d) With PTF pointers, the hierarchy is only partly connected. No pointers exist to connect parent and child segments. Physical child pointers can be used to form this connection. PTF pointers should be used when segments in a database record are typically processed randomly, and you do not need sequential processing of database records.

(e) Four bytes are needed for the PTF pointer in each segment occurrence of a given segment type.

(f) ***PTF pointers are specified by coding PTR=T in the SEGM statement in the DBD.*** This is the SEGM statement for the segment containing the pointer. The combination of PCF

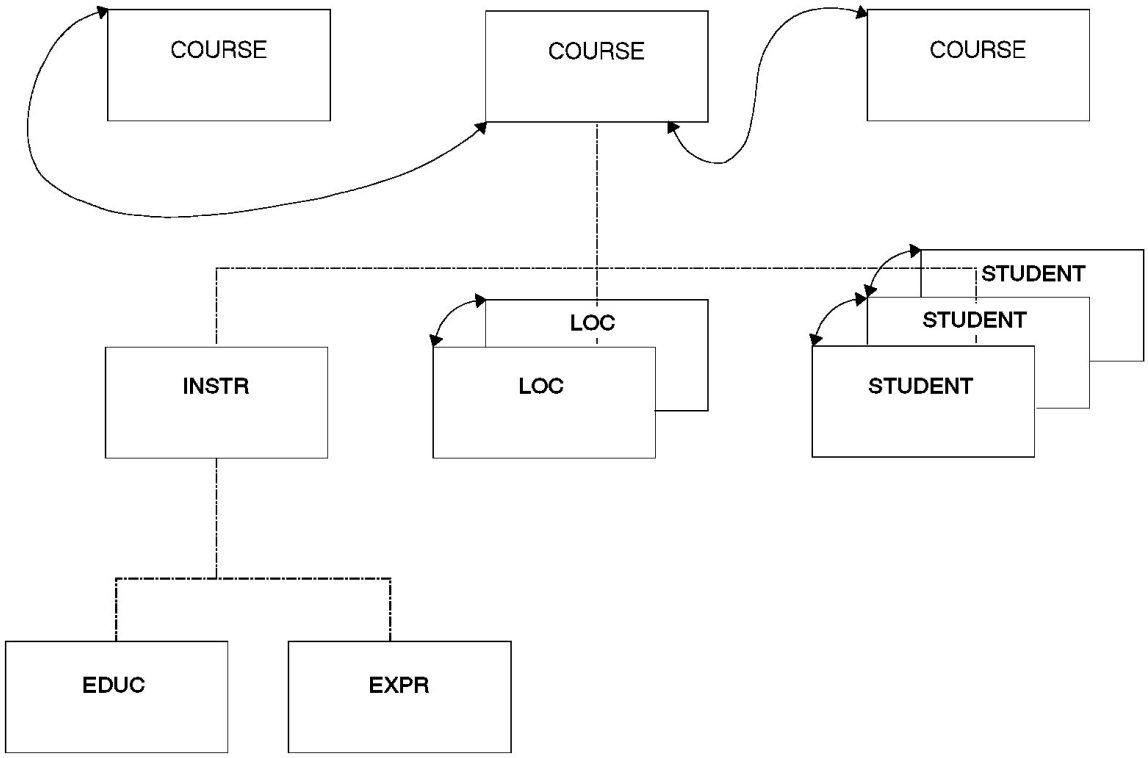
and PTF pointers is used as the default when pointers are not specified in the DBD.



Physical Twin Forward Pointers

Physical Twin Forward and Backward Pointers

With physical twin forward and backward (PTF and PTB) pointers, each segment occurrence of a given segment type under the same parent points both forward to the next segment occurrence and backward to the previous segment occurrence. PTF and PTB pointers must be used together, since you cannot use PTB pointers alone.



Physical Twin Forward and Backward Pointers

(a) Notice that PTF and PTB pointers can be specified for root segments. When this is done, the root segment points to both the next and the previous root segment in the database. As with PTF pointers, PTF and PTB pointers leave the hierarchy only partly connected. No pointers exist to connect parent and child segments. Physical child pointers (explained previously) can be used to form this connection.

(b) PTF and PTB pointers (as opposed to just PTF pointers) should be used on the root segment of a HIDAM database when you need fast sequential processing of database records. By using PTB pointers in root segments, database records can be sequentially processed without intervening references to the HIDAM index. ***PTB pointers improve performance when deleting a segment in a twin chain accessed by a virtually paired logical relationship***. This happens when the delete that causes DASD space to be released occurs on a delete from the logical access path.

(c) Eight bytes are needed for the PTF and PTB pointers in each segment occurrence of a given segment type.

(d) ***PTF and PTB pointers are specified by coding PTR=TB in the SEGM statement in the DBD.***

Summary of Database Types and Capabilities

	HSAM	HISAM	HDAM	HIDAM	DEDB	MSDB
Hierarchical Structures	yes	yes	yes	yes	yes	no
Direct Access Storage	yes	yes	yes	yes	yes	no
Multiple Data Set Groups	no	no	yes	yes	no	no
Logical Relationships	no	yes	yes	yes	no	no
Variable-Length Segments	no	yes	yes	yes	yes	no
Segment Edit/Compression	no	yes	yes	yes	yes	no
Field-Level Sensitivity	yes	yes	yes	yes	no	no
Primary Index	no	yes	no	yes	no	no
Secondary Index	no	yes	yes	yes	no	no
Logging, Recovery, Reorg	no	yes	yes	yes	yes	yes
VSAM	no	yes	yes	yes	yes	N/A
OSAM	no	no	yes	yes	no	N/A
QSAM/BSAM	yes	no	no	no	no	N/A
Boolean Operators	yes	yes	yes	yes	yes	no
Command Codes	yes	yes	yes	yes	yes	no
Online Utilities	no	no	no	no	yes	no
Batch	yes	yes	yes	yes	no	no

IMS Batch Environment

Index

When you complete this section, you will be able to identify all of the internal and external components of the IMS batch environment to successfully create an IMS batch application program

Batch system Environment

The correct IMS batch application program task sequence appears in the figure below. It involves the following seven steps;

- The IMS/VS batch control region (along with other IMS/VS modules) is loaded by the operating systems task management facility. Every IMS batch job that is run initiates a separate IMS/VS batch control region.

- The control region passes control to your IMS batch application program. Your application program must have been previously linked with the application language interface. There are separate language interfaces for COBOL, PL/I and IBM 370 Assembler Language.

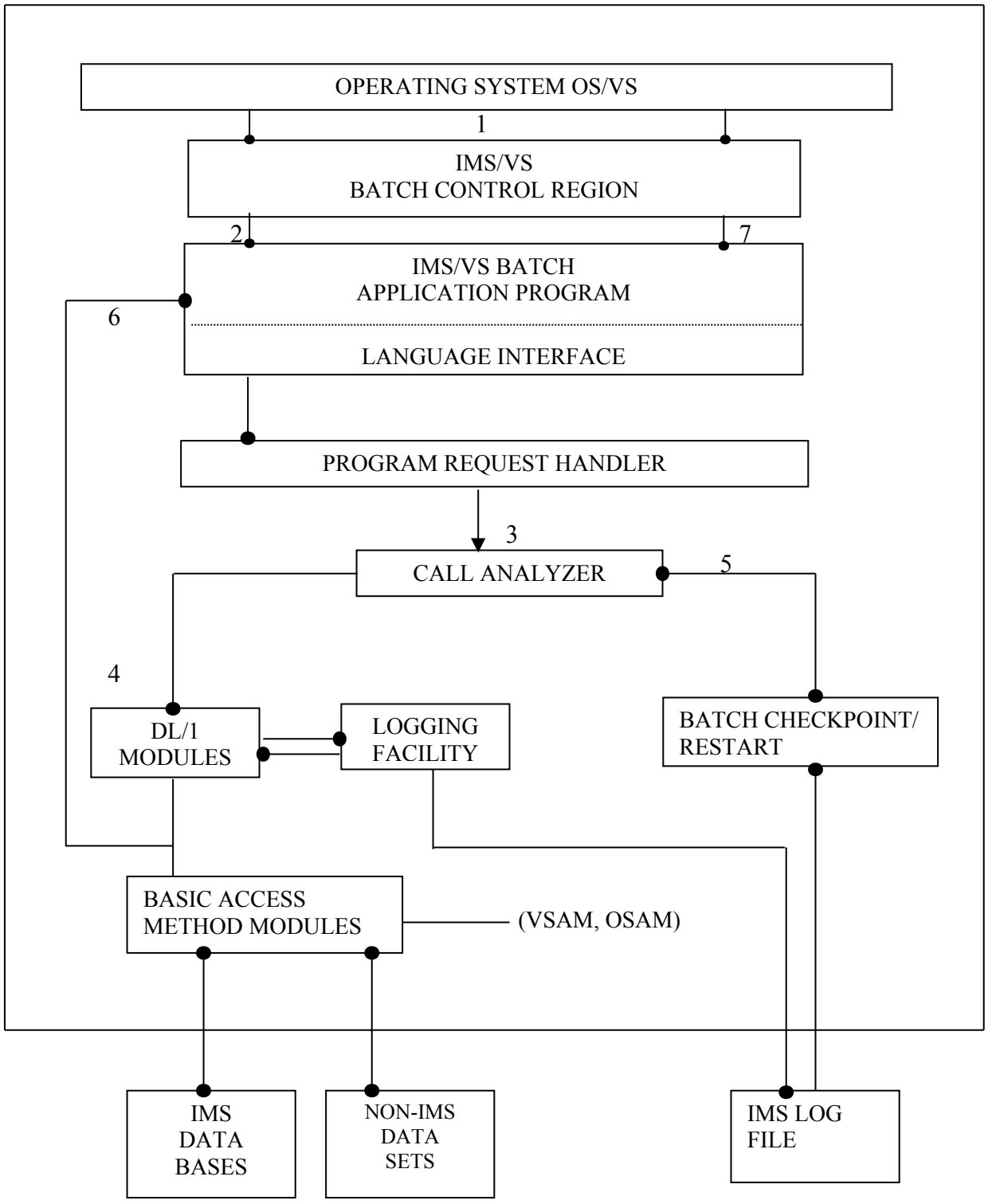
- When your application program issues Data Language/I (DL/I) calls, control is passed to the language interface. Then, the language interface passes control to the program request handler. DL/I is the data base language used to manipulate data on IMS databases. Once the program request handler is in control, the DL/I call parameters are checked for validity.

- The call analyzer receives control from the program request handler where, depending upon what type of call has been issued, control is passed to the appropriate call processor module. This activity is logged using the logging facility.

- If your program requests a checkpoint, them the batch checkpoint/restart module's services are requested and its information is logged via the logging facility.

- Unlike an IMS online application program, a batch IMS application program can access non –IMS data sets. In that case, your request is passed directly to the basic access method modules for processing.

- Upon termination of your application program, control is passed back to the IMS/VS batch control region and your program is terminated. The control region concludes its processing, and control returns to the operating system that will terminate the IMS/VS batch control region.



IMS Batch Environment

It is important to know that IMS, like all good data base management systems, provides online capabilities in addition to batch. The online environment is significantly different from the batch environment. This section addresses only the batch IMS environment.

Batch application Environment

IMS was designed to keep the physical and logical aspects of an application program separate from one another. This means data storage and access methods usually do not concern the application programmer. However, the access method can make a difference in terms of efficiency. For example, sequential processing is very inefficient when done with HDAM databases. Typically, you only want to access the data by reading from or writing to the database. In a non-IMS program, if new data elements are added to the record or if the access method is changed from QSAM to VSAM, you are required to change your program, even if these changes have no direct effect on the output of your program. An IMS program allows you to add new segments to a structure, change the data storage method, or change the access method without modifying the application program. The only consideration that arises from these changes is whether you need to access the new segment that has been added to the data structure.

Let's Examine the IMS/VS Batch Application Program/Language Interface block shown in the Figure above. There are internal and external components in an IMS application program.

External components

Several components external to your application program are required by IMS and DL/I to ensure proper execution of the program.

Data Language/1

This component is the data manipulation language required to access IMS databases. It is not a programming language; it is an addition to a programming language. DL/I has interfaces to COBOL, PL/I and 370 assembler language.

IMS control Blocks

IMS control blocks are required by IMS and DL/I to define the physical and logical structures of databases. IMS control blocks also define a program's access rights to databases and segments within those databases.

The data base description (DBD) defines the physical and logical hierarchical structures in IMS. The type of data storage and access method is also defined here. Segment key and search fields are defined in the DBD to aid in fast and efficient data access. The code below is an example of a DBD.

```
DBD       NAME=DPSPAYRP,ACCESS= (HISAM, VSAM)
DATASET   DD1=FPSPAYRP,OVFLW=FPSPAYRO
SEGM      NAME=PSEMPLYR,PARENT=0,BYTES =50
FIELD     NAME =(EMPLYCOD,SEQ),BYTES=9,                          X
          START=1,TYPE=C
SEGM      NAME=PSPAYCKR, PARENT = PSEMPLYR,  BYTES =50
FIELD     NAME=(PAYDATE, SEQ), BYTES =6,                         X
          START=1,TYPE=C
FIELD     NAME=PAYAMT,BYTES=4,START=7,TYPE=P
•
•

DBDGEN
END
```

DBD Control Block Example

Program Specification Block

The program specification block (PSB) defines which IMS data bases the application program can access, the program's logical view of the data base structure, and the type of access available to each data base segment (such as the Ability to ADD, CHANGE and DELETE). ALL IMS application programs must have a PSB.

The program communication block (PCB) is a part of the PSB that specifies the DBD to be used by the program. The application program will access at least one PCB for every database. The code below is an example of a PSB and Associated PCBs.

370 IMS ASSEMBLER MACROS CODED BY DATA BASE ADMINISTRATOR

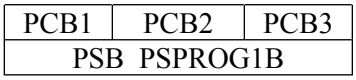
```
PCB       TYPE=DB,NAME=DPSPERSP ……
SENSEG    NAME=PSEMPLRR,PARENT=0,PROCOPT=G
SENSEG    NAME=PSDEPMTR,PARENT=PSEMPLRR,PROCOPT=GI
•
•
PCB       TYPE=DB,NAME=DPSPAYRP…….
SENSEG    NAME=PSEMPLYR,PARENT=0,PROCOPT=G
SENSEG    NAME=PSPAYCKR,PARENT=PSEMPLYR,PROCOPT=K
•
•
PCB       TYPE=DB,NAME=DPSBENFP…….
SENSEG    NAME=PSEMPLBR,PARENT=0,PROCOPT=A
•
•
PSBGEN    LANG=COBOL,PSBNAME=PSPROG1B,……
END
```

PCB1

PCB2

PCB3

PSB CONTROL BLOCK AFTER LINKEDIT



PSB Control Block Example

The DBD and PSB control blocks are usually created and maintained by the Database Administration group, but many sites have programmers who are responsible for coding their PSBs. Control blocks are written in 370 assembler Language using IMS macros.

When you are designing an IMS application, you should involve the DBA group early in the design process to ensure that a workable data base structure is developed and implemented.

Internal Components

An application program can be prepared in several specific ways to make sure it will function properly in the IMS environment.

Application Program Entry point

To make sure an IMS application program functions properly, you must first provide a link that enables DL/I to communicate with the application program. This is accomplished by specifying a point of entry into your program or an entry point. Figure below shows how this entry point is established within COBOL and PL/I.

PL/I

```
DLITPLI : PROC (  pcb _name1_PTR,
                  pcb _name2_PTR,
                  .
                  .
                  pcb_namex_PTR )  OPTIONS (MAIN);
```

COBOL

```
ENTRY   'DLITCBL'  USING  L-pcb –name1,
                          L-pcb –name2,
                          .
                          .
                          L-pcb –namex.
```

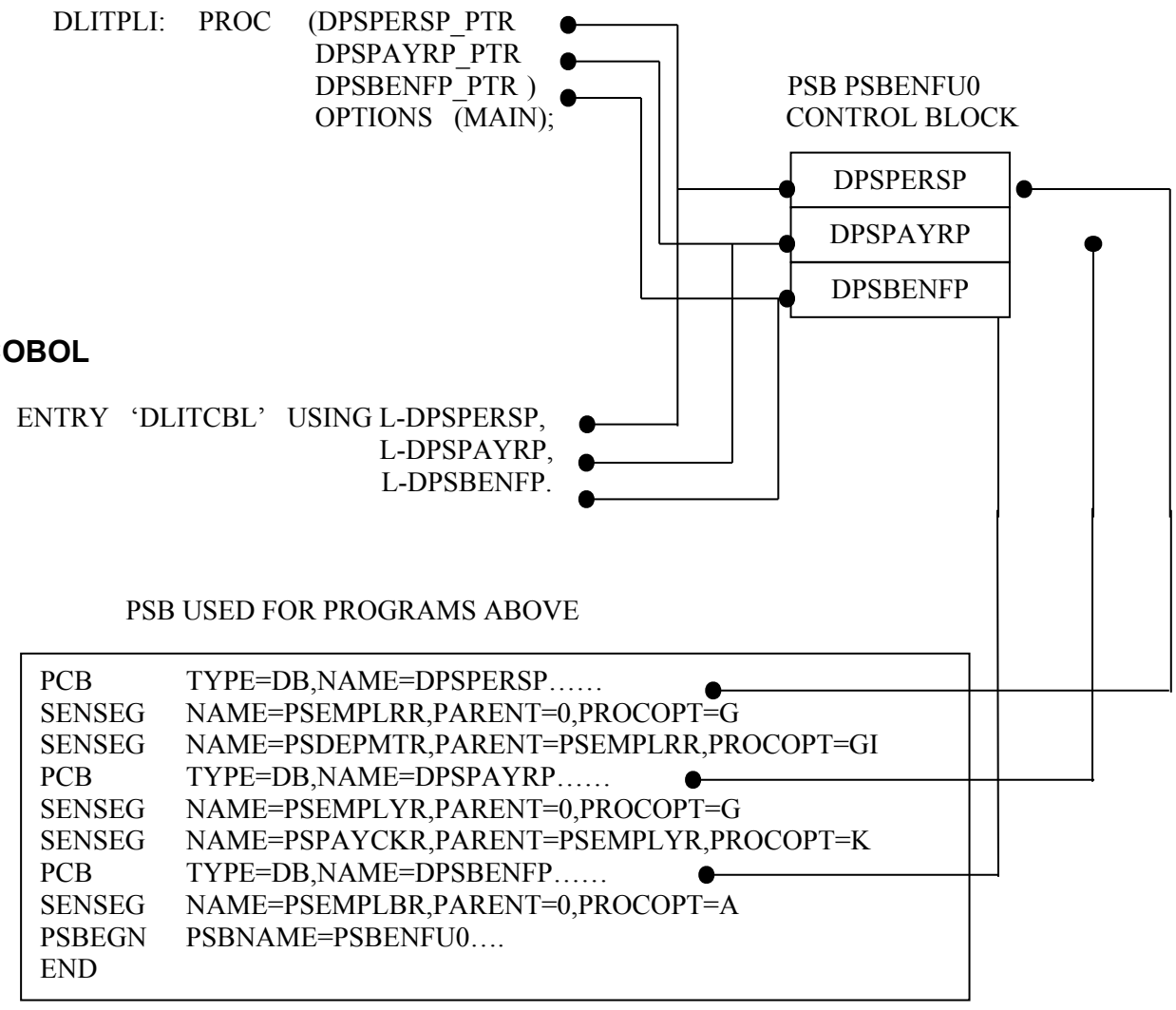
Application program Entry point Examples

In COBOL, the ENTRY statement must be the first statement in the procedure division, and the entry name must be DLITCBL. In PL/I, the main procedure name, DLITPLI, is optional.

The variables with pcb-name1, pcb-name2, and so on, represent addresses passed by IMS to your program. These addresses contain the location of each PCB specified in your PSB. The names you use for the variables are not critical, but it is recommended that you use some form of the DBD name so it can be easily identified in your program.

Figure below shows the relationship between your application program, the PCB Pointers, and the PSB.

PL/1



COBOL

```
ENTRY 'DLITCBL' USING L-DPSPERSP,
                      L-DPSPAYRP,
                      L-DPSBENFP.
```

PSB USED FOR PROGRAMS ABOVE

```
PCB       TYPE=DB,NAME=DPSPERSP……
SENSEG    NAME=PSEMPLRR,PARENT=0,PROCOPT=G
SENSEG    NAME=PSDEPMTR,PARENT=PSEMPLRR,PROCOPT=GI
PCB       TYPE=DB,NAME=DPSPAYRP……
SENSEG    NAME=PSEMPLYR,PARENT=0,PROCOPT=G
SENSEG    NAME=PSPAYCKR,PARENT=PSEMPLYR,PROCOPT=K
PCB       TYPE=DB,NAME=DPSBENFP……
SENSEG    NAME=PSEMPLBR,PARENT=0,PROCOPT=A
PSBEGN    PSBNAME=PSBENFU0….
END
```

Program and PCB,PSB Relationship

Although the names you choose in your program to represent the PCBs are not critical. However the order or position of each PCB Pointer in your program is important. This because IMS will pass the PCB addresses to your program in the same order they are specified in the PSB. In the Figure above, the order of the PCBs in the PSB are DPSPERSP, DPSPAYRP, and DPSBENFP. IMS will map the first PCB address, DPSPERSP, to the

first address specified in your program (via the pointer in PL/1, or via the 01 linkage name in COBOL), the second to the second, and so on.

Linkage Editor Entry Point

The application program entry point was established so that DL/I can communicate with your program. You need to allow the application program to communicate with DL/I so that your program can request its services to access IMS databases. This is done through the linkage editor when your program is linked. The figure below shows the SYSLIN control statements required to link your application program with the DL/I language interface.

PL/I

```
// LINKEDIT     JOB…
// STEP010      EXEC    PGM =IEWL….
      .
      .
// RESLIB       DD      DSN =IMS RESIDENT LIBRARY….
// SYSLIN       DD      DSN = OBJECT MODULE FROM COMPILE,….
//               DD      *
     LIBRARY RESLIB (PLITDLI)        dl/i language interface
     ENTRY PLICALLA                  not needed in LE compliant systems
/*
```

Note : you must also include the following statement in your application source code for PL/I

DCL PLITDLI EXTERNAL ENTRY;

COBOL

```
//LINKEDIT     JOB…
//JOBPARM      ….
//STEP010      EXEC    PGM =IEWL,…
•
•
//RESLIB  DD   DSN =IMS resident library….
//SYSLIN  DD   DSN =object module from compile…
//           DD   *
     LIBRARY RESLIB (CBLTDLI)        dl/i language interface
     ENTRY DLITCBL
/*
```

Linkage Editor Entry Point Examples

The language interface name in the linkage editor control statements is CBLTDLI and PLITDLI. This is opposite of your program entry point of DLITCBL and DLITPLI. This is because in your program entry point, DLITCBL or DLITPLI allows DL/I to communicate with your application program. The linkage editor LIBRARY include of CBLTDLI or PLITDLI members also allows your program to communicate with DL/I.

The entry name DLITCBL in the linkage editor ENTRY statement is required for COBOL. The entry name PLICALLA is required for PL/I. (Not for systems with Language Environment)

PCB MASK

When your application program requires access to an IMS database, issue a DL/I call statement. When the call is completed, DL/I passes information back to your program so you can determine the success or failure of the call. To enable your program to receive this in formation, you must define an I/O area into which the information is placed. This I/O area is called a PCB mask. All PCB masks look alike, regardless of the database you access (except when the database is GSAM). This section does not address GSAM PCB masks because GSAM databases are used almost exclusively in application programs that use the IMS checkpoint facility.

Because all DB PCB masks contain the same information, a data structure is typically used to define them. Figure below shows the structure layout of an IMS data base PCB mask.

PL/I

```
DCL   PCB_PTR                                  POINTER;
DCL   01   PCB_NAME                  BASED (PCB_PTR),
      05   DBD_NAME                    CHAR (08),
      05   SEGMEN_HIERARCHY_LEVEL_ID   CHAR (02),
      05   DLI_STATUS _CODE            CHAR (02),
      05   DLI_PROCESSING_OPTIONS      CHAR (04),
      05   RESERVED_FOR_DLI            CHAR (04),
      05   SEGMENT_NAME _FEEDBACK_AREA CHAR (08),
      05   LENGTH _OF _KEY_FEEDBACK_AREA FIXED  BIN (31),
      05   NUMBER_OF_SENSITIVE_SEGMENTS  FIXED BIN (31),
      05   KEY_FEEDBACK_AREA           CHAR (nn);
```

COBOL

```
01 L-PCB-NAME.
    05   L–DBD–NAME                        PIC   X(08).
    05   L–SEGMENT–HIERARCHY–LEVEL–ID      PIC   X(02).
    05   L–DLI-STATUS–CODE                 PIC   X(02).
    05   L–DLI–PROCESSING–OPTIONS          PIC   X(04).
    05   FILLER                            PIC   X(04).
    05   L–SEGMENT–NAME–FEEDBACK–AREA      PIC   X(08).
    05   L–LENGTH–OF–KEY–FEEDBACK–AREA     PIC  S9(05) COMP.
    05   L–NUMBER–OF–SENSITIVE–SEGMENTS    PIC  S9(05) COMP.
    05   L–KEY–FEEDBACK–AREA               PIC   X(nn).
```

In COBOL, define the PCB mask structure in the LINKAGE SECTION. The 01 level name must match the name, or names, specified in the USING part of the ENTRY statement.

The PCB mask structure must be based on a pointer in PL/I. The 01 level name does not need to be the same name, or names, specified in the main PROCEDURE statement. However, it is recommended that the two names be identical to facilitate program maintenance.

The following code snippets show the relationship between the PCB mask structure and the PCB pointers specified in your program.

PL/I

```
DLITPLI ; PROC (   DPSPERSP_PTR,
                   DPSPAYRP_PTR,
                   DPSBENFP_PTR ) OPTIONS (MAIN);
DCL  (DPSPERSP_PTR, DPSPAYRP _PTR, DPSBENFP _PTR)   POINTER;
DCL   01   PERSONNEL_PCB BSED (DPSPERSP_PTR),
              05  DBD_NAME                    CHAR(8),
              .
              .
DCL  01    PAYROLL_PCB BASED  (DPSPAYRP_PTR),
              05 DBD _NAME                    CHAR (8),
              .
              .
DCL  01   BENEFITS_PCB BASED (DPSBENFP_PTR),
              05  DBD _NAME                   CHAR (8),
              .
              .
```

COBOL

```
LINKAGE SECTION.
01     L-DPSPERSP.
       05   DBD-NAME                PIC  X (8).
        .
        .
01   L-DPSPAYRP.
       05   DBD-NAME                PIC X (8).
        .
        .
01   L-DPSBENFP
       05  DBD-NAME                 PIC X (8).
        .
        .

PROCEDURE  DIVISION.
  ENTRY DLITCBL USING   L-DPSPERSP,
```

L-DPSPAYRP,
L-DPSBENFP.

PCB Mask to PCB Pointer Relationship

Segment I/O Areas

When the application program requests that data be retrieved from an IMS data base, DL/I not only supplies information about the call, but also supplies information about the data retrieved from the data base. The data retrieved is the data contained in the segment requested in the call. Your program must provide an I/O area to store the data retrieved. This I/O area is the same as the I/O areas that are specified for any ordinary file. It is usually a structure. This structure defines the data elements contained in the segment just as you would define data elements in a record for a sequential file.

Here is a PL/I and COBOL example.

PL/I

```
/*****************************************************/
/*I/O AREA FOR DATABASE ROOT SEGMENT                 */
/*(UNSTRUCTURED)                                     */
/*****************************************************/
DCL   ROOT _SEGMENT   CHAR (77);
/*****************************************************/
/*I/O AREA FOR PAYROLL DATABASE ROOT SEGMENT  */
/* (STRUCTURED)                                      */
/*****************************************************/
DCL  01   PAYROLL_ROOT_SEGMENT_STRUC,
          05      EMPLOYEE _ID                CHAR (6),
          05      EMPLOYEE_NAME               CHAR (30),
          05      EMPLOYEE_DEPT _CODE         CHAR (4),
          05      FILLER                      CHAR (37);
```

COBOL

```
WORKING STORAGE SECTION.
*****************************************************************
*  I/O AREA FOR PAYROLL DATABASE ROOT SEGMENT                   *
*  (UNSTRUCTURED)                                               *
*****************************************************************
01    PAYROLL-ROOT-SEGMENT                  PIC  X(77).
*****************************************************************
*  I/O AREA FOR PAYROLL DATABASE ROOT SEGMENT                   *
*  (STRUCTURED)                                                 *
*****************************************************************
01   PAYROLL-ROOT-SEGMENT-STRUC.
     05    PR-EMPLOYEE-ID                   PIC  X(6).
     05    PR-EMPLOYEE-NAME                 PIC  X(30).
     05    PR-EMPLOYEE-DEPT-CODE            PIC  X(4).
```

```
05      PR-FILLER                                   PIC  X(37).
```

Segment Structure I/O Areas

Usually, you have a structure for every segment type accessed in your program, and are include into the source by COBOL COPY statements or %INCLUDE statement in PL/I. This is the same approach for structures which form the I/O area for records read from or written into a file.

Batch JCL Requirements

Another topic that pertains to the external features of your application program is the JCL required to actually run your job and execute your program. Figure below shows an example of the typical JCL required to support a batch IMS application program.

```
// IMSBATCH    JOB   …
// STEP010     EXEC  PGM=DFSRRC00,                                  <- - 1
//                   PARM='DLI,pgmname,psbname,…..'                <- - 2
// STEPLIB     DD    DSN=….  IMS AUTHORIZED LIBRARY, DISP=SHR        <- -3
//             DD    DSN=…. LOAD LIBRARY ,DISP=SHR                 <- -3
// IMS         DD    DSN=…. DBD LIBRARY,DISP=SHR                   <- -4
//             DD    DSN=…. PSB  LIBRARY ,DISP=SHR                 <- -4
// DFSRESLB    DD    DSN=…. IMS AUTHORIZE LIBRARY,DISP=SHR         <- -5
// DFSVSAMP    DD    DSN=…. IMS BUFFER INFORMATION, DISP=SHR       <- -6
// dbddd       DD    DSN=…. IMS  DATA BASES, DISP=OLD              <- -7
// osfile      DD    DSN=…. NON- IMS FILES, DISP=OLD               <- -8
// IMSLOGR     DD    DUMMY                                         <- -9
// IEFRDER     DD    DSN=….  IMS  LOG  DATASET,                    <- -10
//         DISP=(NEW,CATLG,CATLG),UNIT=tape/dasd,
//         DCB=(RECFM=VB,LRECL=4148,BLKSIZE=4152)
// SYSPRINT    DD  SYSOUT=…
```

JCL Sample for Batch IMS Program

The following information examines JCL in more detail:

- 1. The program executed in an IMS batch job is not your application program. It is the IMS batch control region. DFSRRC00.

- 2. Many parameters are passed to DFSRRC00, but the following three are the most often used.

DL/I – This parameter is required. It tells IMS that it will be responsible for the application control block (ACB) building process. Other possible values are DBB and BMP.

pgmname – This parameter is required. It must be the load module name your application program was linked under.

psbname – This parameter is optional. It is where you specify the PSB name that your program is going to use. If it is omitted, the PSB name will default to the same name specified for the application program.

- 3. The STEPLIB DDNAME specifies IMS resident libraries concatenated with the load library to which your program has been linked (optional, depending on system setup).

- 4. The IMS DDNAME is required. It specifies the concatenation of the DBD and PSB control block libraries. These two libraries are used by IMS to dynamically build the ACB.

- 5. The DFSRESLB DDNAME is optional and required at others (dependent on system setup). It is the library that contains the IMS modules. If you do not include this DDNAME in your execution JCL and it abends with a S306, then you will be required to include this DDNAME before you rerun the job.

- 6. The DFSVSAMP DDNAME is only required if the execution JCL contains DDNAMES for IMS databases that use the VSAM access method. If used, it specifies the database buffer information to IMS. Typically, this is a data set predefined by the DBA.

- 7. The DDNAME used here reflects the DDNAME defined in the DBD for this database. There is at least one DDNAME for every database defined in the PSB in the PARM= on the EXEC statement.

ß 8. If your application program uses any non-IMS data sets, you must include a DDNAME for each of those as indicated by the FILE declaration in PL/I, or the FD in COBOL.

- 9. The IMSLOGR DDNAME is used for recovery and restart.

- 10. The IEFRDER DDNAME specifies the log file to which IMS writes log records. If your application program updates anything in a database, IMS will attempt to open this file. Otherwise, it is an optional DDNAME. This data set is usually a generation data set, but it does not have to be.

JCL

Here is a working JCL and DLIBATCH procedure tested on S/390 V2R9. Copy this procedure from IMS.PROCLIB into your own userid.PROCLIB and tailor it as below. Your userid is assumed to be USER01:-

DLIBATCH

```
//        PROC MBR=TEMPNAME,PSB=,BUF=7,
//             SPIE=0,TEST=0,EXCPVR=0,RST=0,PRLD=,
//             SRCH=0,CKPTID=,MON=N,LOGA=0,FMTO=T,
//             IMSID=,SWAP=,DBRC=,IRLM=,IRLMNM=,
//              BKO=N,IOB=,SSM=,APARM=,
//             RGN=2048K,
```

```
//*            SOUT=A,LOGT=2400,SYS2=,
//             SOUT=A,SYS2=,
//             LOCKMAX=,GSGNAME=,TMINAME=
//G       EXEC PGM=DFSRRC00,REGION=&RGN,
//             PARM=(DLI,&MBR,&PSB,&BUF,
//             &SPIE&TEST&EXCPVR&RST,&PRLD,
//             &SRCH,&CKPTID,&MON,&LOGA,&FMTO,
//             &IMSID,&SWAP,&DBRC,&IRLM,&IRLMNM,
//             &BKO,&IOB,&SSM,'&APARM',
//             &LOCKMAX,&GSGNAME,&TMINAME)
//STEPLIB  DD DSN=IMS.&SYS2.RESLIB,DISP=SHR
//          DD DSN=IMS.&SYS2.PGMLIB,DISP=SHR
//          DD DSN=USER01.LOADLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.&SYS2.RESLIB,DISP=SHR
//IMS       DD DSN=USER01.PSBLIB,DISP=SHR
//          DD DSN=USER01.DBDLIB,DISP=SHR
//PROCLIB  DD DSN=IMS.&SYS2.PROCLIB,DISP=SHR
//IEFRDER  DD DSN=USER01.IMSLOG,DISP=(NEW,KEEP),
//          DCB=(RECFM=VB,BLKSIZE=1920,
//          LRECL=1916,BUFNO=2),SPACE=(TRK,(1,1))
//SYSUDUMP DD SYSOUT=&SOUT,
//          DCB=(RECFM=FBA,LRECL=121,BLKSIZE=605),
//          SPACE=(605,(500,500),RLSE,,ROUND)
//IMSMON   DD DUMMY
```

Here is a sample Job stream to run your IMS DB Job

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
//    JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC DLIBATCH,MBR=pgmname,PSB=psbname,DBRC=N
//G.SYSPRINT DD SYSOUT=*
//*
//*  THIS IS FOR THE HIDAM OR HISAM PATIENT DATABASE
//G.ddname1 DD DSN=USER01.database1,DISP=SHR
//G.ddname2 DD DSN=USER01.database2,DISP=SHR
//*
//* THIS IS THE FOR LOAD OR UPDAT DATA
//G.SYSIN DD DSN=USER01.loadorupdate.DATA,DISP=SHR
/*
//G.DFSVSAMP DD *
VSRBF=2048,4
/*
//
```

Here is a sample Job Stream to Prepare your PL/I program JOB

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC IMSPLI,REGION=0M
//C.SYSIN  DD DSN=USER01.PLI.SOURCE(memname),DISP=SHR
//L.SYSLMOD DD DSN=USER01.LOADLIB(memname),DISP=SHR
//
```

IMSPLI PROC

```
//        PROC MBR=TEMPNAME,PAGES=50,SYS2=,
//             SOUT=A
//C       EXEC PGM=IEL0AA,REGION=114K,
//             PARM=(XREF,A,OBJ,NODECK,NOMACRO,,
//             'OPT(TIME)',SYSTEM(IMS))
//STEPLIB DD DSN=IEL.V1R1M1.SIELCOMP,DISP=SHR
//         DD DSN=CEE.SCEERUN,DISP=SHR
//SYSUT1   DD UNIT=SYSDA,
//          SPACE=(1024,(60,60),RLSE,,ROUND),
//          DCB=BLKSIZE=1024,DISP=(,DELETE)
//SYSPRINT DD SYSOUT=&SOUT,
//          DCB=(LRECL=125,BLKSIZE=629,RECFM=VBA),
//          SPACE=(605,(&PAGES.0,&PAGES),RLSE)
//SYSLIN   DD UNIT=SYSDA,SPACE=(80,(250,80),RLSE),
//          DCB=(IMS.&SYS2.PROCLIB),
//          DISP=(,PASS)
//L       EXEC PGM=IEWL,PARM='XREF,LIST,LET',
//             COND=(4,LT,C),REGION=120K
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//RESLIB   DD DSN=IMS.&SYS2.RESLIB,DISP=SHR
//SYSLIN   DD DSN=*.C.SYSLIN,DISP=(OLD,DELETE)
//          DD DISP=SHR,
//          DSN=USER01.PROCLIB(PLITDLI)
//          DD DDNAME=SYSIN
//SYSLMOD  DD DISP=SHR,
```

```
//          DSN=IMS.&SYS2.PGMLIB(&MBR)
//SYSPRINT DD SYSOUT=&SOUT,
//          DCB=(LRECL=121,RECFM=FBA,BLKSIZE=605),
//          SPACE=(605,(&PAGES.0,&PAGES),RLSE)
//SYSUT1    DD UNIT=SYSDA,DISP=(,DELETE),
//          SPACE=(CYL,(5,1),RLSE)
```

Note !!!:
You have to copy the member PLITDLI from IMS.PROCLIB to USER01.PROCLIB and then edit it to remove the line ENTRY PLICALLA

Sample Programs

You will use later. Compile and link edit these into your Load Library. You may not understand all the code at this point in this training course.

PL/I Programs PLIPGM4

```
*PROCESS NOT('~');
 PLIPGM4: PROC(PCBPTR) OPTIONS(MAIN);
 DCL ADDR BUILTIN;
 DCL PCBPTR POINTER;
 DCL PLITDLI EXTERNAL ENTRY;
 DCL 01 PCBMASK BASED(PCBPTR),
     02 DBD_NAME    CHAR(8),
     02 SEG_ID      CHAR(2),
     02 STATUS      CHAR(2),
     02 PROCOPT     CHAR(4),
     02 RESERVED    CHAR(4),
     02 SEGMENT_NAME CHAR(8),
     02 LENGTH_FDBK FIXED BIN(31),
     02 NUM_SENSEGS FIXED BIN(31),
     02 KEY_FDBK_AREA CHAR(21);
 DCL SYSPRINT FILE PRINT STREAM OUTPUT ENV(VB BLKSIZE(129));
 DCL SYSIN FILE STREAM INPUT ENV(FB BLKSIZE(800) RECSIZE(80));
 DCL BUFF CHAR(80)INIT('');
 DCL FLAG BIT(1) INIT('1'B);
 DCL PARM_COUNT FIXED BIN(31);
 DCL FUNCTION CHAR(4);
 DCL SEG_IO_AREA CHAR(35) BASED(ADDR(BUFF));
 PUT SKIP LIST('STARTING PLIPGM4');
 FUNCTION='GN  ';PARM_COUNT=3;
 DCL SSA  CHAR(9) INIT('PATIENT');
 DO WHILE (FLAG='1'B);
     SEG_IO_AREA='';
     CALL PLITDLI(PARM_COUNT,FUNCTION,PCBPTR,SEG_IO_AREA);
     IF STATUS~='GB' THEN DO;
     PUT SKIP EDIT('SEGMENT ',SEGMENT_NAME,' : ',SEG_IO_AREA)
       (A,A,A,A);
     END;
     IF STATUS='GB' THEN DO;
       PUT SKIP LIST('NON BLANK FROM GN CALL ',STATUS);
       RETURN;
     END;
 END;
```

```
 PUT SKIP LIST('PLIPGM4 ENDING');
 END PLIPGM4;
```

PL/I Program PLIPGM5

```
PLIPGM5: PROC(PCBPTR) OPTIONS(MAIN);
 DCL ADDR BUILTIN;
 DCL PCBPTR POINTER;
 DCL PLITDLI EXTERNAL ENTRY;
 DCL 01 PCBMASK BASED(PCBPTR),
     02 DBD_NAME    CHAR(8),
     02 SEG_ID      CHAR(2),
     02 STATUS      CHAR(2),
     02 PROCOPT     CHAR(4),
     02 RESERVED    CHAR(4),
     02 SEGMENT_NAME CHAR(8),
     02 LENGTH_FDBK FIXED BIN(31),
     02 NUM_SENSEGS FIXED BIN(31),
     02 KEY_FDBK_AREA CHAR(21);
 DCL SYSPRINT FILE PRINT STREAM OUTPUT ENV(VB BLKSIZE(129));
 DCL SYSIN FILE STREAM INPUT ENV(FB BLKSIZE(800) RECSIZE(80));
 DCL BUFF CHAR(80);
 DCL PARM_COUNT FIXED BIN(31);
 DCL FUNCTION CHAR(4);
 DCL 01 BUFFER BASED(ADDR(BUFF)),
        02  SSA CHAR(10),
        02  SEG_IO_AREA CHAR(35);
 DCL FLAG BIT(1) INIT('1'B);
 ON ENDFILE(SYSIN) FLAG='0'B;
 ON UNDEFINEDFILE(SYSIN) BEGIN;
   PUT SKIP LIST('UNABLE TO OPEN SYSIN');
   EXIT;
 END;
 PUT SKIP LIST('STARTING PLIPGM5');
 FUNCTION='ISRT';PARM_COUNT=4;
 DO WHILE (FLAG='1'B);
  GET EDIT(BUFF)(A(80));
  IF FLAG='1'B THEN DO;
     PUT SKIP LIST(BUFF);
     CALL PLITDLI(PARM_COUNT,FUNCTION,PCBPTR,SEG_IO_AREA,SSA);
     IF STATUS='  ' THEN; ELSE DO;
       PUT SKIP LIST('NON BLANK FROM INSERT CALL ',STATUS);
       RETURN;
     END;
  END;
 END;
 PUT SKIP LIST('PLIPGM5 ENDING');
 END PLIPGM5;
```

IMS Control Blocks-Basic

Index

When you complete this section, you will understand the following:

- The various IMS control blocks.
- The Contents of the DBD and PSB control blocks.

Data Base Description

IMS allows the definition of databases outside your application program through the use of the data base description. The DBD is written in IBM 370 assembler Language using IMS macro statements. While you will have to wait till the next chapter to learn how to code a DBD, this chapter does discuss those aspects of the DBD that are relevant to application programming, such as the following:

- DBD name
- IMS and MVS Access methods
- DDNAMES for JCL
- The hierarchical structure of the data base
- Key and search field names

After a DBD is created, it is assembled and link edited to a PDS library (often called DBDLIB) for storage. DBDLIB provides IMS with a common library from which you can access the information stored in the DBD

Here is an example of the source code for an IMS data base.

```
DBD       NAME=CT7DPIS1,ACCESS=(HISAM,VSAM)
DATASET   DD1=CT7DPISP,OVFLW=CT7DPISO
SEGM      NAME=RSITEMSR,BYTES=62,PARENT=0
FIELD     NAME=(ITEMNUMB,SEQ,U), BYTES=006,START=001,TYPE=C
```

```
FIELD       NAME=COLOR,BYTES=008,START=008,TYPE=C
FIELD       NAME=MANFNAME,BYTES=015, START=016,TYPE=C
SEGM        NAME=RSRENTER,BYTES=119,PARENT=RSITEMSR
FIELD       NAME=(SOCSECNM,SEQ,U),BYTES=009,START=001,TYPE=C
FIELD       NAME=NAME,BYTES=030,START=071,TYPE=C
SEGM        NAME=RSSTATSR,BYTES=33,PARENT=RSITEMSR
FIELD       NAME=STATUS,BYTES=001,START=001,TYPE=C
SEGM        NAME=RSFINANR,BYTES=22,PARENT=RSITEMSR
FIELD       NAME=FINCDBY,BYTES=004,START=009,TYPE=C
FIELD       NAME=PURCDATE,BYTES=006,START=013,TYPE=C
DBDGEN
FINISH
END
```

DBD Source Code Example

Let's examine the DBD closely and identify the information it provides that is necessary for application programming.

DBD Name

The DBD is the first macro specified, as shown in the following example:

```
DBD         NAME=CT7DPIS1,ACCESS=(HISAM.VSAM)
```

The keyword NAME = specifies the name of the DBD (in this case, it is CT7DPIS1).

It also is the name under which the DBD is link edited. If you access this data base in an application program, this is the name that must appear in the PCB of your PSB. You should also use (from the programming standards point of view) it as the pointer name in your application program's main PROCEDURE statement for PL/I, or in the USING part of the ENTRY statement for COBOL. This simplifies program maintenance.

Storage and Access Methods

The DBD macro contains the storage and access methods used for the database. This is specified in the DBD with the ACCESS = keyword, as shown in the following example:

```
DBD       NAME=CT7DPIS1,ACCESS=(HISAM,VSAM)
```

IMS provides many different storage and access methods, such as HIDAM, HISAM, HDAM, HSAM, SHISAM, AND GSAM. The details about each of these have been discussed in an earlier chapter and the following are important to you as an application programmer:-

- If your data base is stored as HDAM, you must be aware that you cannot process root segments sequentially in key order. This access method stores root segments randomly throughout the database on the basis of the root key value. If you sequentially process the roots, you will get the key values in ascending or any other logical sequence.

- If your database is stored as HIDAM, you need to be aware that two DBDs are required to support it. One for the index and one for the primary and overflow datasets. This means that there will be two DD statements required in the JCL. HIDAM also provides direct access to any root segment through the use of its associated index database.

The next macro appearing in a DBD is the DATASET macro. It contains several useful pieces of information.

JCL DD Names

To access an IMS database, it is necessary to allocate the database to your job. This is done in batch through JCL. In an IMS batch application program, there is no specification of a FD as in a file oriented COBOL program, or a FILE declaration in PL/I, for IMS data bases. The data bases you access are referenced in your program's associated PSB via the PCB macro.

Refer to "Program Specification Block" for more detail on the PSB. The example of a DATASET macro appears in the following illustration.

```
DATASET      DD1=CT7DPISP,OVFLW=CT7DPIS0. . . .
```

The DD1=keyword specifies the DDNAME of the primary index dataset to be allocated in your JCL. In the example above, it is CT7DPISP.

Depending on the access method used, you may have an additional DD required because the database may be composed of multiple data sets. such an example is shown in the following illustration:

```
DATASET DD1=CT7DPISP,OVFLW=CT7DPIS0,. . . .
```

The OVFLW = keyword specifies the DDNAME of the overflow dataset to be allocated in your JCL. CT7DPIS0 is the DDNAME in the above example.

An example of the JCL statements needed to support the DBD in the DBD Source Code Example is shown in the following example:

```
//CT7DPISP DD DSN=....PRIMARY / INDEX DATASET
//CT7DPIS0 DD DSN=....OVERFLOW DATASET
```

Hierarchical Structure

One of the most important pieces of information provided by the DBD is the physical or logical (if ACCESS=LOGICAL ON DBD macro) structure of the data base. This is accomplished by specifying from 1 to 255 SEGM macros (255 is the maximum number of segment types allowed in a hierarchical structure). SEGM represents SEGMENT. An example of a typical SEGM macro found in a DBD is shown in the following illustration:

```
SEGM      NAME=RSITEMSR,BYTES=62,PARENT=0
```

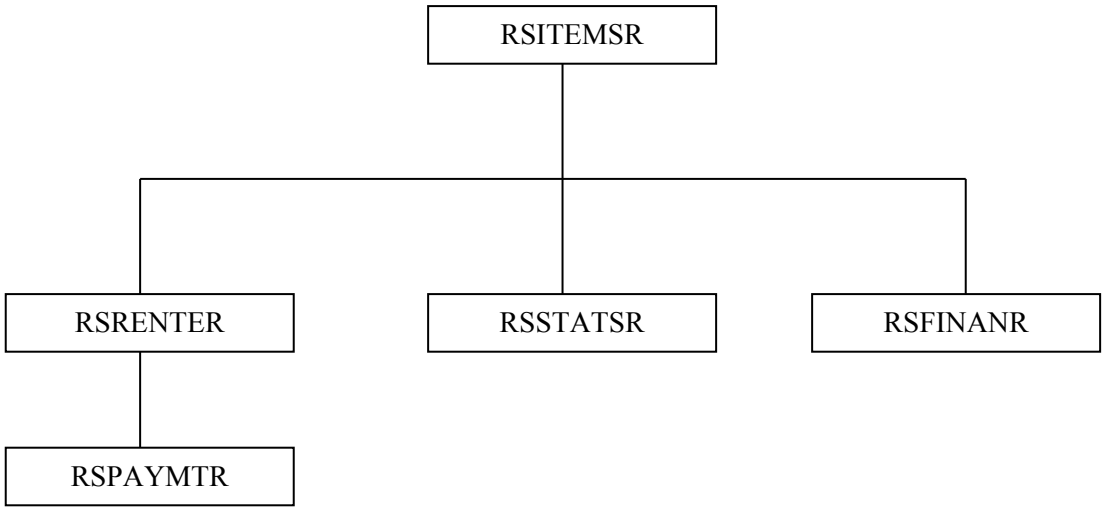
The NAME = keyword specifies the segment name. If the application program requires access to this segment, that name must appear in your PSB.

The SEGM macro also contains information about the hierarchical structure of the data base. This is indicated by the PARENT = keyword, as shown in the following example:

```
SEGM     NAME=RSITEMSR,BYTES=62,PARENT=0
  .
  .
SEGM  NAME=RSRENTER,BYTES=119,PARENT=RSITEMSR
```

You may recall that a hierarchical structure represents a group of related segments. These relationships are communicated in parent-to-child terms. The example above shows two SEGM macros. The first specifies a PARENT=0. The 0 (zero) means that this segment is the root. The second, however, specifies PARENT=RSITEMSR. This means that segment RSITEMSR is the parent of segment RSRENTER. Based on this relationship, you can begin to build a graphical representation of the database, which is very helpful.

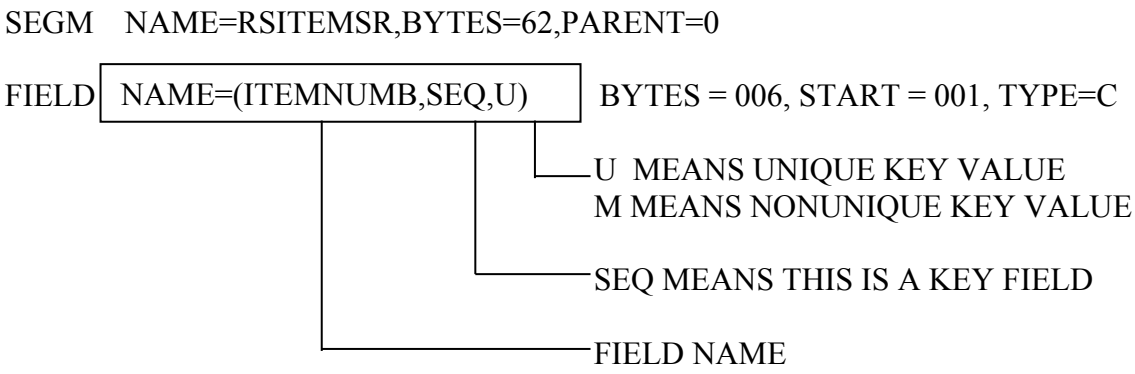
The order of SEGM macros specified in a DBD always represents the hierarchical sequence of the data base. That is, the root, SEGM, is first, followed by dependent SEGMENTS in a top-to-bottom, left-to-right fashion. To better understand how to extract the hierarchical structure from a DBD, study the DBD example code shown earlier. The following hierarchical structure can be determined based on the sequence of SEGM macros in the DBD.



When you are working with an unfamiliar database, it helps to look at the DBD source. The DBA can provide this. Using the DBD source as a reference point, you can draw the data base structure, which is much easier to understand than the DBD source code alone.

Key and Search Fields

The key and search fields contained in the DBD are also relevant to application programming. When access is required for a segment, it often is for a specific segment occurrence. For example, you might want information on item number 160769 from your item database. To avoid searching every segment for item number 160769, you can specify a key data element or field for the segment. Do this by specifying a FIELD macro under the associated SEGM macro relating to the segment for which you are defining the key field. This is shown in the following example:



The NAME = keyword of the FIELD macro specifies the name of the key or search field. Notice the breakdown of the NAME = keyword. Also notice that this is the key field for segment RSITEMSR because the FIELD macro appears under segment RSITEMSR SEGM macro. Only one key field may be specified per segment. If one is specified, it must be the first FIELD = macro under the SEGM. It is not always required to specify a key field for a segment.

You can also specify FIELD macros that do not represent key fields, but are fields through which you would like to search the segment. Let's look at the following example:

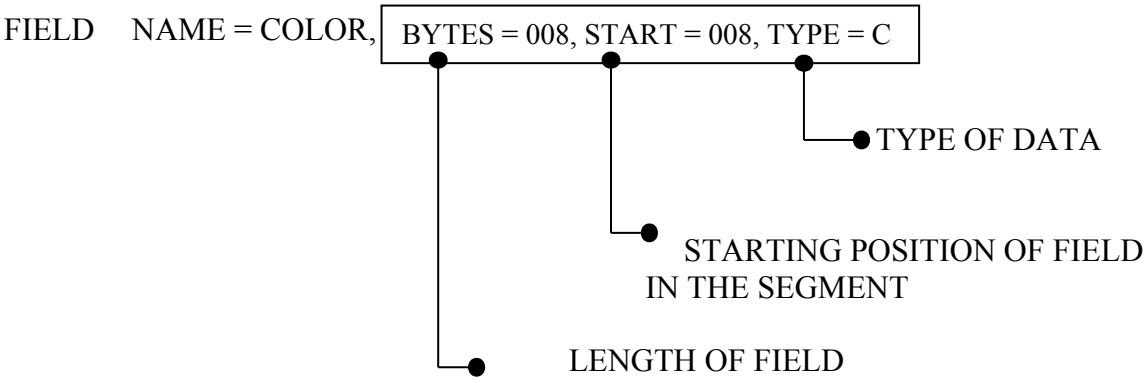
```
SEGM    NAME=RSITEMSR,BYTES=62,PARENT=0
FIELD   NAME=(ITEMNUMB,SEQ,U),BYTES=006,START=001,TYPE=C
FIELD   NAME=COLOR,BYTES=008,START=008,TYPE=C
FIELD   NAME=MANFNAME,BYTES=015,START=016,TYPE=C
```

Notice in the preceding example that, in addition to the key field, there are two other FIELD macros, one for COLOR and one for MANFNAME. These fields are search fields, not key fields. The fact that the NAME = keyword does not contain the SEQ parameter, as does field ITEMNUMB, indicates to IMS that the field is a search field, not a key field. The difference is that during the insertion of new occurrences of a segment, they will be inserted in ascending key sequence if a key field is specified. Otherwise, they are inserted one after the other as they occur.

The TYPE field of the FIELD macro indicates how IMS is to initialize that field if the application has insert authority for the segment but is not sensitive to the field. The various values possible are:-

Type field possible values
C Character
P Packed Decimal
Z Zoned DEC
X Hexadecimal
H Half-word binary
F Full-word binary

The last piece of information in the FIELD macro is shown in the following example:



Information, such as the length of the field, where it starts in the segments, and the type of data that it is, is found in the FIELD macro statement.

More detailed information on the DBD can be found in the IBM manual, IMS/VS *Utilities Reference manual.*

Program Specification Block

IMS keeps the data base description separate from your application program through the use of the database description control block. However, when you need access to one or more IMS databases, you need to communicate your requirements to IMS. Like the data base description, this information is kept separate from the application program through the use of the program specification block. You use the PSB to define the following:

- The data bases to which you need access
- The segments within each data base to which you need access
- The processing requirements for the segments

After a PSB is created, it is assembled and link edited to a PDS library (also called PSBLIB) for storage. PSBLIB then provides IMS with a common library from which to access the information stored in the PSB. A code snippet for a PSB follows.

```
PCB         TYPE=DB,NAME=CT7DPIS1,PROCOPT=AP,KEYLEN=21
```

```
SENSEG    NAME=RSITEMSR,PARENT=0
SENSEG    NAME=RSRENTER,PARENT=RSITEMSR
SENSEG    NAME=RSPAYMTR,PARENT=RSRENTER
SENSEG    NAME=RSSTATSR,PARENT=RSITEMSR
SENSEG    NAME=RSFINANR,PARENT=RSITEMSR,PROCOPT=G
PCB       TYPE=DB,NAME=CT7DPIS1,PROCOPT=G,KEYLEN=21,                X
          PROCSEQ=C7DKLNM
SENSEG    NAME=RSITEMSR,PARENT=0
SENSEG    NAME=RSRENTER,PARENT=RSITEMSR
SENSEG    NAME=RSPAYMTR,PARENT=RSRENTER
PSBGEN    PSBNAME=RSITEMUP,LANG=PL/I
END
```

PSB Source Code Example

The PSB control block contains a wealth of information for the application programmer. Let's examine the first macro in a PSB,the PCB macro.

Program Communication Block

The program communication block is a mini control block contained within the PSB control block. The PCB specifies the Database for which access is required. The following example shows the PCB macro and its components.

```
PCB       TYPE=DB,NAME=CT7DPIS1,PROCOPT=AP,KEYLEN=21
```

The TYPE = keyword on the PCB indicates the type of PCB it its. DB stands for database. Other code abbreviations are TP, for teleprocessing (Used for online application programs or batch message processing programs), and GSAM, for generalized sequential access method databases (used in IMS checkpoint programs).

The next keyword, shown in the following example, specifies the database name.

```
PCB   TYPE=DB,NAME=CT7DPIS1,PROCOPT=AP, KEYLEN=21
```

The NAME = keyword on the PCB specifies the DBD name of the database when the keyword TYPE=DB. This must be the same name with which the date base's DBD was link edited. It also is the recommended name you use as a PCB pointer name in your application program. The PSB will contain at least one PCB for every database you access in your program.

Logical View

The next macro in a PSB after the PCB is the SENSEG macro. This macro identifies the specific segment within the database to which you need access, and it produces the program logical view of the database. SENSEG stands for SENsitive SEGment.

```
SENSEG    NAME=RSITEMSR,PARENT=0
```

The NAME = keyword of the SENSEG macro identifies the segment name to which your program has access (sensitivity). This must be the same name that is in the SEGM macro of the DBD to which this PCB points. The order of the SENSEG macros under each PCB is critical. It must represent a valid hierarchical structure as defined by the corresponding DBD. It is not required that every segment defined in the DBD be represented by a SENSEG in the PSB, only those segments required for processing by your application program.

If more than one SENSEG is specified within a PCB, how does IMS know that a valid hierarchical structure is being specified? The keyword PARENT= on the SENSEG macro is used to establish the hierarchical relationships between the specified SENSEG macros. The following example illustrates this:

```
PCB       TYPE=DB, NAME=CT7DPIS1, PROCOPT=AP,KEYLEN=21
SENSEG    NAME=RSITEMSR, PARENT=0
SENSEG    NAME=RSRENTER, PARENT=RSITEMSR
```

There are two SENSEG macros specified in the example above. The first, segment name RSITEMSR, specifies PARENT=0. A parent of 0 (zero) indicates the root segment. The second macro identifies the segment, RSRENTER, with a PARENT=RSITEMSR. This means that the segment RSITEMSR is the parent of segment RSRENTER. This relationship is verified against the DBD specified in the PCB. In this example, it is DBD CT7DPIS1. The SENSEG relationships are not validated until the block building process is performed by IMS during the execution of the application program.

Processing Options

An important piece of information required by IMS is how you wish to process the databases and their segments. The following example shows how this is accomplished by using the PROCOPT = keyword on either the PCB or SENSEG macros, or on both.

```
PCB       TYPE=DB,NAME=CT7DPIS1,PROCOPT=AP, KEYLEN=21
SENSEG    NAME=RSITEMSR,PARENT=0
SENSEG    NAME=RSRENTER,PARENT=RSITEMSR
SENSEG    NAME=RSPAYMTR,PARENT=RSRENTER
SENSEG    NAME=RSSTATSR,PARENT=RSITEMSR
SENSEG    NAME=RSFINANR,PARENT=RSITEMSR,PROCOPT=G
PSBGEN    PSBNAME=RSITEMUP,LANG=PL/I
END
```

The keyword PROCOPT= stands for PROCessing OPtions. It indicates the type of processing required for either the database or the segments specified for those databases, or both.

The following Figure is a table of common processing options and the function of each. Not all possible processing options are shown here, only those most commonly used.

Certain processing options (up to a maximum of four) can be used. In the example above, you see that the keyword PROCOPT= can be specified on either the PCB macro or the SENSEG macro, or both. If the processing options are specified in the PCB, IMS considers them defaults, and they will be used by every SENSEG specified under the PCB. Any overrides at SEGMENT level can be specified in the PROCOPT at the SENSEG level. If no processing options are specified in the PCB macro, they must be specified at every SENSEG macro under the PCB. In the previous example, the PCB had PROCOPT=AP and the SENSEG had keyword PROCOPT=G. This means that for all SENSEGs that do not have the PROCOPT=keyword specified, IMS will use the processing options specified in the PCB(in this case PROCOPT=AP). The SENSEG identified as PROCOPT=G will not take the default processing options of AP. Instead, it will take the G option. This means that access to this segment is Get (retrieval) only. Access to all other segments will default to AP (which is All), or Get, Insert, Replace and Delete. You can also do path calls using these segments.

Option	Function
G	Get – Allows read only access to the segment.
I	Insert – Allows add only access to the segment.
R	Replace – Allows replacement and read of the segment.
D	Delete – Allows deletion and read of the segment.
A	All – Allows Gets, Insertion, Replacement and Deletion of the segment
K	Key – Allows access only to the key field in the segments in the I/O area
L(S)	Load Sequential – Used to initially load a database.
GO	Get Only – Will not enqueue (lock) segment.

Table of Common Processing Options

Processing Sequence

Earlier, it was mentioned that you could process a database via a secondary index. If you wish to process a database through the secondary index, you need to have the DBA define the PROCSEQ= parameter in the PCB, as shown below. Every time you access the database with this PCB, it will be accessed via the secondary index instead of the root segment index.

```
PCB    TYPE = DB,NAME=CT7DPIS1,PROCOPT=G, KEYLEN=21,PROCSEQ=CT7DKLNM
```

PSBNAME

The last PSB macro discussed is the PSBGEN macro. It is as shown in the following example:

```
PSBGEN      PSBNAME=RSITEMUP,LANG = PL/I
```

The PSBNAME = keyword specifies the name that the PSB will be link edited under. This also is the name that will used in batch JCL to indicate to IMS which PSB control block to use during the execution of the application program. In the example above, the PSB name is RSITEMUP.

Application Program Language

The PSBGEN macro specifies the application programming language that can use the PSB. It does this through the LANG=keyword. LANG=stands for LANGuage. In the following example, LANG=PL/I indicates that this PSB can be used by PL/I programs only. Other values are COBOL and ASSEM. COBOL and Assembler application programs can use PSBs by identifying either LANG=COBOL or LANG=ASSEM. Identical control blocks are generated for COBOL and ASSEMBLER. PL/I is the exception to the rule. A PL/I program must use a PSB with LANG=PL/I and a PSB with LANG=PL/I can only be used by a PL/I application program.

```
PSBGEN  PSBNAME=RSITEMUP, LANG=PL/I
```

Security

The ability to specify processing options in a PSB also helps deal with data security issues. IMS uses the processing options specified in the PSB to request the proper services from DL/I, such as adding, changing, deleting, or retrieving segments. DL/I only provides those services, at a PCB or sensitive segment level, specified by the processing options.

The PSB (PCB) controls the level of access. The PSB is generated under control of the DBA by controlling access to the PSBLIB.

In addition to using the processing options, of other measure can be taken to ensure data security within a database by using the SENFLD macro (which stands for SENsitive FieLD). This macro is placed after the SENSEG macro and tells DL/I that access to the segment is limited only to the SENFLD macros specified for that segment. This is called field level sensitivity. This is a powerful option of the PSB. Not only does it allow access to certain fields in a segment, but it also allows you to restructure the I/O area when the segment is retrieved. If the SENFLD option is used, no other field from that segment is returned to the program's I/O area except for the SENFLD data.

Application Control Block

Before IMS runs your application program, it builds a single control block from the DBD and PSB control blocks. This block building process is the application control block (ACB). All the necessary information from the DBD and PSB control blocks is merged into the ACB so that IMS has only one control block to deal with. The ACB contains all the information necessary for the operation of IMS and DL/I. It is created either dynamically or statically.

Dynamic ACB

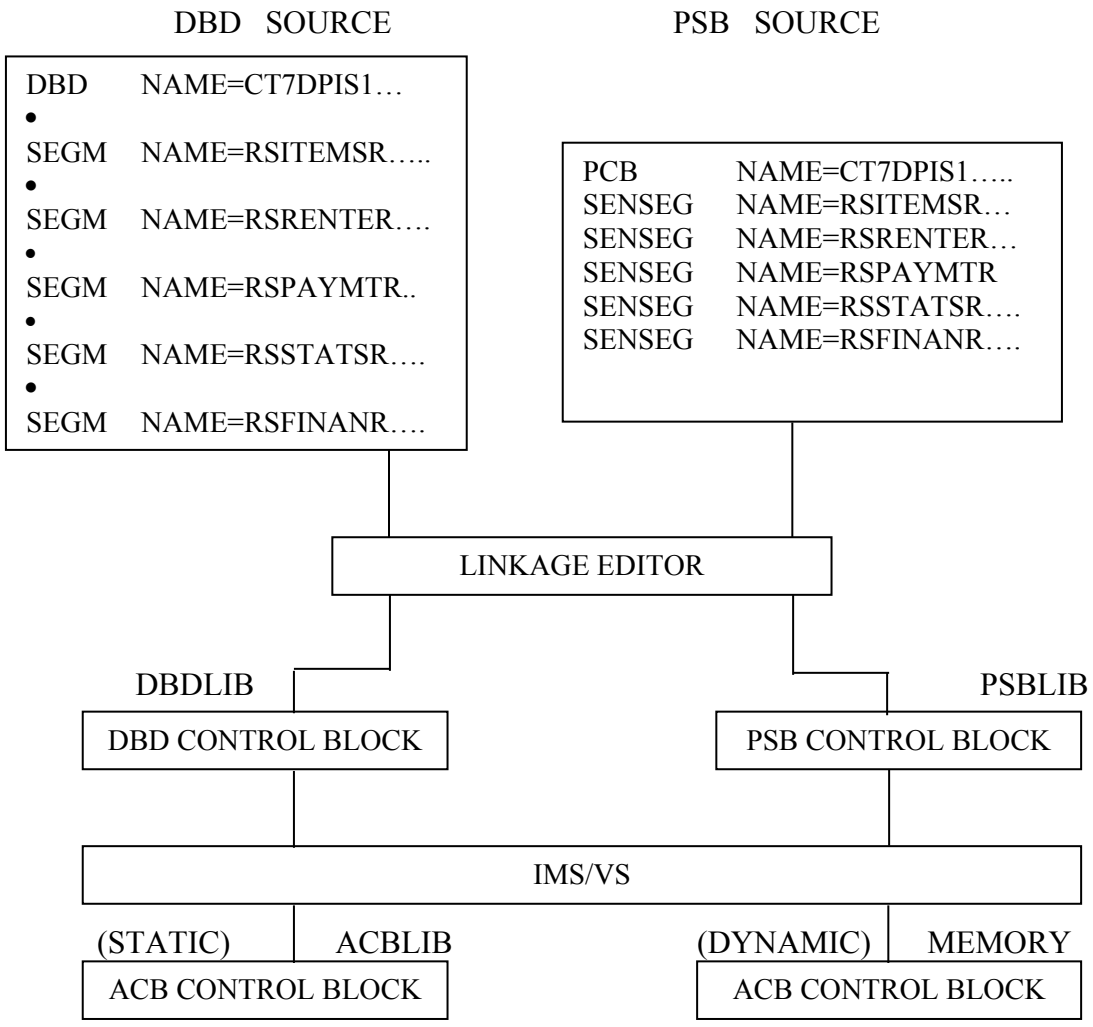
IMS dynamically builds an ACB each time a batch IMS application program is executed. since a batch program is usually run only once or twice a day, the overhead involved in block building is of little concern to the computer installation support teams.

Static ACB

The IMS block building process does not take a great deal of time. However, online programs that may be executed hundreds or thousands of times a day are costly. Dynamically building an ACB for each execution would become prohibitive. To solve this problem, IMS provides an internal facility that prebuilds ACBs. IMS requires that all online application programs have a prebuilt ACB for their PSBs. The DBA group is responsible for creating prebuilt ACBs. After it is created, an ACB is stored on a PDS library (often called ACBLIB) for access by IMS. Processing time is greatly reduced by retrieving, rather than creating, a prebuilt ACB for each online application program. ACBs are not usually created for IMS batch application programs because most batch programs are run only once or twice per day.

The DBA group is usually responsible for creating and maintaining all IMS control blocks. This centralized approach provides consistency and minimizes redundancy in the creation and maintenance of the IMS control blocks.

Figure below illustrates the block building process performed by the DBA and IMS.



Control Block Building

Summary

The following four IMS control blocks help keep data access and application program logic independent of one another.

- The data base description
- The program specification block
- The program communication block
- The application control block

The data base description block contains macros that specify the name of the data base description, the storage and access methods used for the data base, the physical or logical structure of the data base, and the names of key or search fields.

The program specification block contains the program communication block. It also contains macros that specify the program's logical view of the database, the processing options for segments within the data bases accessed, and the programming language(s) that can use the program specification block. Another major function of the program specification block is security. It decides which data bases can be accessed, as well as how each segment within the database can be processed.

The program communication block is a miniature control block that is contained within the PSB control block. The PCB specifies the IMS resources for which access is required by name and type.

The application control block is composed of the data base description and the program specification control blocks combined. Application control blocks are either dynamically or statically created. Each time a batch IMS application program is executed, IMS dynamically builds an ACB. To run online programs, which can be executed hundreds or thousands of time per day, prebuilt ACBs are stored in a PDS library and retrieved as needed. These prebuilt ACBs are called static application control blocks.

Control Block Exercise

Below is a DBD control block. Answer the following questions about is.

```
DBD       NAME=CT7DPPER,ACCESS=(HDAM,OSAM),                    X
          RMNAME=(DFSHDC40,004,00000007,0250)
DATASET   DD1=CT7DPISP,DEVICE=3380,SIZE=(02048)
SEGM      NAME=RSCOMMNR,BYTES =80,PARENT =0
FIELD     NAME=(SOCSECNM, SEQ,U),BYTES =009,START=001,TYPE =C
FIELD     NAME=LASTNAME,BYTES=015,START=010,TYPE =C
FIELD     NAME=STATE,BYTES =002,START=069,TYPE=C
SEGM      NAME= RSPAYRLR, BYTES =020, PARENT =RSCOMMNR
FIELD     NAME=(PAYDATE,SEQ,U),BYTES=006,START =001,TYPE =C
SEGM      NAME=RSPRSNLR, BYTES =025, PARENT =RSCOMMNR
FIELD     NAME=(JOBCODE,SEQ,U),BYTES=005,START=001,TYPE=C
SEGM      NAME=RSMEDICR,BYTES=090,PARENT=RSCOMMNR
FIELD     NAME=(VISITDTE,SEQ,M),BYTES=006,START=001,TYPE =C
DBDGEN
FINISH
END
```

1. What is the name of the DBD?
2. What is the DDNAME of the primary dataset?
3. What is the DDNAME of the overflow dataset?
4. What is the name of the key field for segment RSPRSNLR?
   - How long is it?
   - Where does it start in the segment?
   - Is it a unique key field?
5. How many search fields are defined in the DBD?
6. Draw the hierarchical structure of this database.

Below is a PSB control block. Answer the following questions about it.

```
PCB       TYPE =DB,NAME=CT7DPPER,PROCOPT=A,KEYLEN=15
SENSEG    NAME=RSCOMMNR,PARENT=0
SENSEG    NAME=RSMEDICR,PARENT=RSCOMMNR,PROCOPT=GRD
PSBGEN    PSBNAME=RSITEMUP,LANG=COBOL
END
```

1. What is the name of the DBD referenced in this PSB?
2. Can a new occurrence of RSMEDICR be added?
3. Can a new occurrence of RSCOMMNR be added?
4. Can this PSB be used for an assembler language program?
5. Can this PSB be used for a PL/I language program?
6. Draw the logical hierarchical structure for this PSB.

Control Blocks for a Sample Application

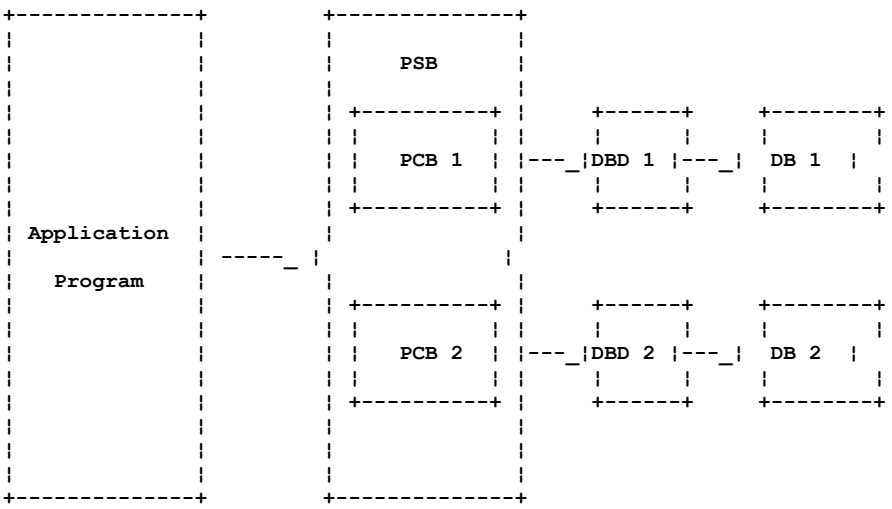
Index

Review of the DBD and PSB Basics

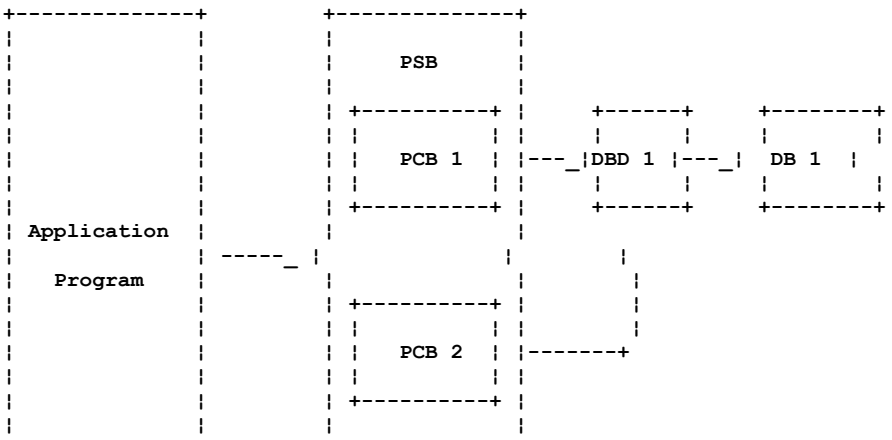
Application programs can communicate with the database without being aware of the physical location of the data they possess. To do this, database descriptors (DBD'S) and program specification blocks (PSB'S) are used.

A DBD describes the content and hierarchical structure of the physical or logical database. DBD'S also supply information to IMS to help in locating segments.

A PSB specifies the database segments an application program can access and the functions it can perform on the data, such as read only, update, or delete. Because an application program can access multiple databases, PSB'S are composed of one or more program control blocks (PCBs). The PSB describes the way a database is viewed by your application program.



The normal relationship between application programs, PSB'S, PCB'S, DBD'S, and databases.

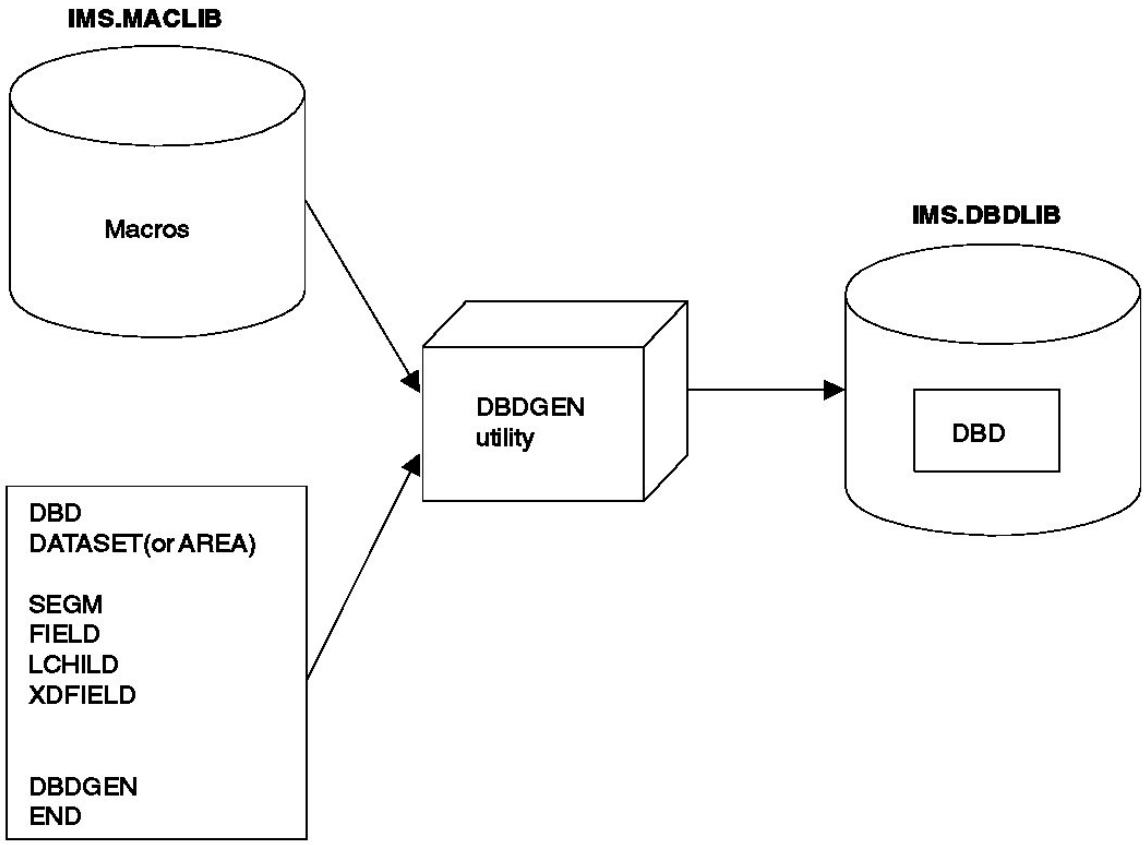


```
¦              ¦        ¦              ¦
¦              ¦        ¦              ¦
+--------------+        +--------------+
```

Relationship between Programs and Multiple PCBs (Concurrent Processing)

DBD

A DBD is a series of macro instructions that describes such things as a database's organization and access method, the segments and fields in a database record, and the relationships between types of segments. After you have coded the DBD macro instructions, they are used as input to the DBDGEN utility. This utility is a macro assembler that generates a DBD control block and stores it in the IMS.DBDLIB library for subsequent use during database processing.



DBDGEN PROCESS

JOB STREAM FOR DBDGEN

```
 //DBDGEN    JOB  MSGLEVEL=1
//                  EXEC        DBDGEN,MBR=APPLPGM1
//C.SYSIN        DD     *
    DBD              required for each DBD generation
    DATASET(or AREA) required for each data set group
                     (or AREA in a Fast Path DEDB)
    SEGM             required for each segment type
    FIELD            required for each DBD generation
    LCHILD           required for each secondary index or
                     logical relationship
    XDFIELD          required for each secondary index relationship
```

```
          .
          .
          .
     DBDGEN           required for each DBD generation
     END              required for each DBD generation
/*
```

Notes:-The following new macros used in definition of secondary indexes / Logical Relationships are explained.

(a) LCHILD Statement

The LCHILD statement defines a secondary index or logical relationship between two segment types, or the relationship between a HIDAM index database and the root segment type in the HIDAM database. LCHILD statements immediately follow the SEGM, FIELD, or XDFLD statement of the segment involved in the relationship. Up to 255 LCHILD statements can be defined for each database.

(b) XDFLD Statement

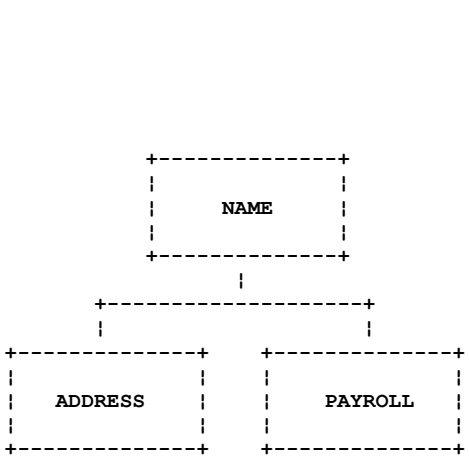
The XDFLD statement is used only when a secondary index exists. It is associated with the target segment and specifies:

- The name of an indexed field
- The name of the source segment
- The field(s) used to create the secondary index from the source segment

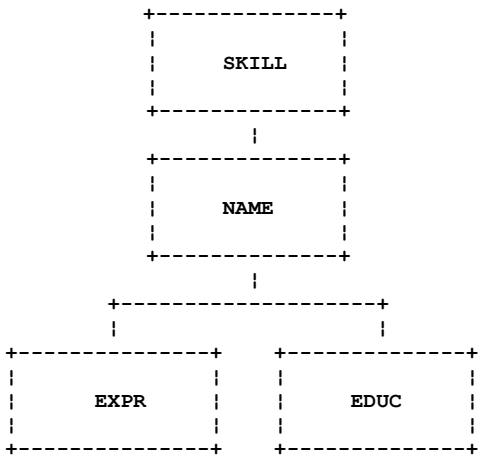
Up to 32 XDFLD statements can be defined per segment. However, the number of XDFLD and FIELD statements combined cannot exceed 255 per segment or 1000 per database.

Learn more from the following examples

The following examples are based on the two data bases indicated below:-



Payroll data structure



Skills inventory data structure

HSAM DBD Generation of Skills Inventory Database

```
DBD   NAME=SKILLINV,ACCESS=HSAM
 DATASET DD1=SKILHSAM,DD2=HSAMOUT,BLOCK=1,RECORD=3000

 SEGM  NAME=SKILL,BYTES=31,FREQ=100
 FIELD NAME=TYPE,BYTES=21,START=1,TYPE=C
 FIELD NAME=STDCODE,BYTES=10,START=22,TYPE=C

 SEGM  NAME=NAME,BYTES=20,FREQ=500,PARENT=SKILL
 FIELD NAME=STDCLEVL,BYTES=20,START=1,TYPE=C

 SEGM  NAME=EXPR,BYTES=20,FREQ=10,PARENT=NAME
 FIELD NAME=PREVJOB,BYTES=10,START=1,TYPE=C
 FIELD NAME=CLASSIF,BYTES=10,START=11,TYPE=C

 SEGM  NAME=EDUC,BYTES=75,FREQ=5,PARENT=NAME
 FIELD NAME=GRADLEVL,BYTES=10,START=1,TYPE=C
 FIELD NAME=SCHOOL,BYTES=65,START=11,TYPE=C

 DBDGEN
 FINISH
 END
```

Notes

1) FREQ= ,is only used for HSAM, HISAM, or INDEX databases. It specifies the estimated number of times that this segment is likely to occur for each occurrence of its physical parent.
2) DD2= , ddname of output data set

HSAM DBD Generation of Payroll Database

```
DBD   NAME=PAYROLDB,ACCESS=HSAM
 DATASET DD1=PAYROLL,DD2=PAYOUT,BLOCK=1,RECORD=1000,
```

```
SEGM  NAME=NAME,BYTES=150,FREQ=1000,PARENT=0
FIELD NAME=(EMPLOYEE,SEQ,U),BYTES=60,START=1,TYPE=C
FIELD NAME=MANNBR,BYTES=15,START=61,TYPE=C
FIELD NAME=ADDR,BYTES=75,START=76,TYPE=C

SEGM  NAME=ADDRESS,BYTES=200,FREQ=2,PARENT=NAME
FIELD NAME=HOMEADDR,BYTES=100,START=1,TYPE=C
FIELD NAME=COMAILOC,BYTES=100,START=101,TYPE=C

SEGM  NAME=PAYROLL,BYTES=100,FREQ=1,PARENT=NAME
FIELD NAME=HOURS,BYTES=15,START=51,TYPE=P
FIELD NAME=BASICPAY,BYTES=15,START=1,TYPE=P

DBDGEN
FINISH
END
```

HISAM DBD Generation of Skills Inventory SKILLINV Database

```
DBD   NAME=SKILLINV,ACCESS=HISAM
DATASET DD1=SKLHISAM,OVFLW=HISAMOVF,

SEGM  NAME=SKILL,BYTES=31,FREQ=100
FIELD NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
FIELD NAME=STDCODE,BYTES=10,START=22,TYPE=C

SEGM  NAME=NAME,BYTES=20,FREQ=500,PARENT=SKILL
FIELD NAME=(STDCLEVL,SEQ,U),BYTES=20,START=1,TYPE=C

SEGM  NAME=EXPR,BYTES=20,FREQ=10,PARENT=NAME
FIELD NAME=PREVJOB,BYTES=10,START=1,TYPE=C
FIELD NAME=CLASSIF,BYTES=10,START=11,TYPE=C

SEGM  NAME=EDUC,BYTES=75,FREQ=5,PARENT=NAME
FIELD NAME=GRADLEVL,BYTES=10,START=1,TYPE=C
FIELD NAME=SCHOOL,BYTES=65,START=11,TYPE=C

DBDGEN
FINISH
END
```

HISAM DBD Generation of Payroll Database

```
DBD   NAME=PAYROLDB,ACCESS=HISAM
DATASET DD1=PAYROLL,OVFLW=PAYROLOV,

SEGM  NAME=NAME,BYTES=150,FREQ=1000,PARENT=0
FIELD NAME=(EMPLOYEE,SEQ,U),BYTES=60,START=1,TYPE=C
FIELD NAME=MANNBR,BYTES=15,START=61,TYPE=C
FIELD NAME=ADDR,BYTES=75,START=76,TYPE=C

SEGM  NAME=ADDRESS,BYTES=200,FREQ=2,PARENT=NAME
FIELD NAME=HOMEADDR,BYTES=100,START=1,TYPE=C
FIELD NAME=COMAILOC,BYTES=100,START=101,TYPE=C

SEGM  NAME=PAYROLL,BYTES=100,FREQ=1,PARENT=NAME
FIELD NAME=HOURS,BYTES=15,START=51,TYPE=P
FIELD NAME=BASICPAY,BYTES=15,START=1,TYPE=P
```

```
DBDGEN
FINISH
END
```

HDAM DBD Generation of Skills Inventory SKILLINV Database with Hierarchic Pointers

```
DBD   NAME=SKILLINV,ACCESS=HDAM,RMNAME=(RAMDMODL,1,500,824)
DATASET   DD1=SKILHDAM,BLOCK=1648,SCAN=5

SEGM   NAME=SKILL,BYTES=31,PTR=H,PARENT=0
FIELD  NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
FIELD  NAME=STDCODE,BYTES=10,START=22,TYPE=C

SEGM   NAME=NAME,BYTES=20,PTR=H,PARENT=SKILL
FIELD  NAME=(STDCLEVL,SEQ,U),BYTES=20,START=1,TYPE=C

SEGM   NAME=EXPR,BYTES=20,PTR=H,PARENT=NAME
FIELD  NAME=PREVJOB,BYTES=10,START=1,TYPE=C
FIELD  NAME=CLASSIF,BYTES=10,START=11,TYPE=C

SEGM   NAME=EDUC,BYTES=75,PTR=H,PARENT=NAME
FIELD  NAME=GRADLEVL,BYTES=10,START=1,TYPE=C
FIELD  NAME=SCHOOL,BYTES=65,START=11,TYPE=C

DBDGEN
FINISH
END
```

Notes:-

RMNAME=(RAMDMODL,1,500,824)

- Parm1:- Name of randomizing module
- Parm2:- Number of root anchor points
- Parm3:- The highest control interval in the root addressable area is used by IMS.
- Parm4:- specifies the maximum number of bytes of database record that can be stored into the root addressable area in a series of inserts unbroken by a call to another database record.

HDAM DBD Generation of Skills Inventory Database with Physical Child and Physical Twin Pointers

```
DBD   NAME=SKILLINV,ACCESS=HDAM,RMNAME=(RAMDMODL,1,500,824)
DATASET   DD1=SKILHDAM,BLOCK=1648,SCAN=5

SEGM   NAME=SKILL,BYTES=31,PTR=T,PARENT=0
FIELD  NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
FIELD  NAME=STDCODE,BYTES=10,START=22,TYPE=C

SEGM   NAME=NAME,BYTES=20,PTR=T,PARENT=((SKILL,SNGL))
FIELD  NAME=(STDCLEVL,SEQ,U),BYTES=20,START=1,TYPE=C

SEGM   NAME=EXPR,BYTES=20,PTR=T,PARENT=((NAME,SNGL))
FIELD  NAME=PREVJOB,BYTES=10,START=1,TYPE=C
FIELD  NAME=CLASSIF,BYTES=10,START=11,TYPE=C

SEGM   NAME=EDUC,BYTES=75,PTR=T,PARENT=((NAME,SNGL))
FIELD  NAME=GRADLEVL,BYTES=10,START=1,TYPE=C
FIELD  NAME=SCHOOL,BYTES=65,START=11,TYPE=C
```

```
DBDGEN
FINISH
END
```

Notes:-

1) PTR=T ----Physical twin Forward Pointer in Segment NAME
2) PARENT=((SKILL,SNGL)) ----Physical child First Pointer in Segment SKILL

Primary HIDAM Index Relationship

```
HIDAM:                              INDEX:
 DBD NAME=dbd1,ACCESS=HIDAM          DBD NAME=dbd2,ACCESS=INDEX
 SEGM NAME=seg1,BYTES=,              SEGM NAME=seg2,BYTES=
      POINTER=
 LCHILD NAME=(seg2,dbd2),            LCHILD NAME=(seg1,dbd1),
        PTR=INDX                            INDEX=fld1
 FIELD NAME=(fld1,SEQ,U),            FIELD NAME=(fld2,SEQ,U),
       BYTES=,START=                       BYTES=,START=
```

Summary of Statements for the Primary HIDAM Index Relationship

The next two examples show the control statements that define the skills inventory data structure as two HIDAM databases. The first is defined with hierarchic pointers, and the second is defined with physical child and physical twin pointers. Since a HIDAM database is indexed on the sequence field of its root segment type, an INDEX DBD generation is required. Shown below are the control statements for the two HIDAM DBD generations and the index DBD generation.

INDEX DBD Generation for HIDAM Database SKILLINV

```
DBD   NAME=INDEXDB,ACCESS=INDEX
DATASET DD1=INDXDB1,

SEGM  NAME=INDEX,BYTES=21,FREQ=10000
LCHILD NAME=(SKILL,SKILLINV),INDEX=TYPE
FIELD NAME=(INDXSEQ,SEQ,U),BYTES=21,START=1

DBDGEN
FINISH
END
```

HIDAM DBD Generation of Skills Inventory Database with Hierarchic pointers

```
DBD    NAME=SKILLINV,ACCESS=HIDAM
DATASET   DD1=SKLHIDAM,BLOCK=1648,SCAN=5

SEGM  NAME=SKILL,BYTES=31,PTR=H,PARENT=0
FIELD NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
FIELD NAME=STDCODE,BYTES=10,START=22,TYPE=C
LCHILD NAME=(INDEX,INDEXDB),PTR=INDX

SEGM  NAME=NAME,BYTES=20,PTR=H,PARENT=SKILL
FIELD NAME=(STDCLEVL,SEQ,U),BYTES=20,START=1,TYPE=C

SEGM  NAME=EXPR,BYTES=20,PTR=H,PARENT=NAME
FIELD NAME=PREVJOB,BYTES=10,START=1,TYPE=C
FIELD NAME=CLASSIF,BYTES=10,START=11,TYPE=C
```

```
SEGM  NAME=EDUC,BYTES=75,PTR=H,PARENT=NAME
FIELD NAME=GRADLEVL,BYTES=10,START=1,TYPE=C
FIELD NAME=SCHOOL,BYTES=65,START=11,TYPE=C

DBDGEN
FINISH
END
```

HIDAM DBD Generation of Skills Inventory SKILLINV Database with Physical Child and Physical Twin Pointers

```
DBD    NAME=SKILLINV,ACCESS=HIDAM
DATASET   DD1=SKLHIDAM,BLOCK=1648,SCAN=5

SEGM  NAME=SKILL,BYTES=31,PTR=T,PARENT=0
LCHILD NAME=(INDEX,INDEXDB),PTR=INDX

FIELD NAME=(TYPE,SEQ,U),BYTES=21,START=1,TYPE=C
FIELD NAME=STDCODE,BYTES=10,START=22,TYPE=C

SEGM   NAME=NAME,BYTES=20,PTR=T,PARENT=((SKILL,SNGL))
FIELD NAME=(STDCLEVL,SEQ,U),BYTES=20,START=1,TYPE=C

SEGM  NAME=EXPR,BYTES=20,PTR=T,PARENT=((NAME,SNGL))
FIELD NAME=PREVJOB,BYTES=10,START=1,TYPE=C
FIELD NAME=CLASSIF,BYTES=10,START=11,TYPE=C

SEGM  NAME=EDUC,BYTES=75,PTR=T,PARENT=((NAME,SNGL))
FIELD NAME=GRADLEVL,BYTES=10,START=1,TYPE=C
FIELD NAME=SCHOOL,BYTES=65,START=11,TYPE=C

DBDGEN
FINISH
END
```

Example of GSAM DBD Generation

```
DBD   NAME=CARDS,ACCESS=(GSAM,BSAM)
DATASET  DD1=ICARDS,DD2=OCARDS,RECFM=F,RECORD=80
DBDGEN
FINISH
END
```

PSB OR PROGRAM SPECIFICATION BLOCK

Review of basics:-A PSB is a series of macro instructions that describes an application program's characteristics, its use of segments and fields within a database, and its use of logical terminals. A PSB consists of one or more PCBs (program communication blocks). Of the two types of PCBs, one is used for alternate message destinations, the other, for application access and operation definitions.

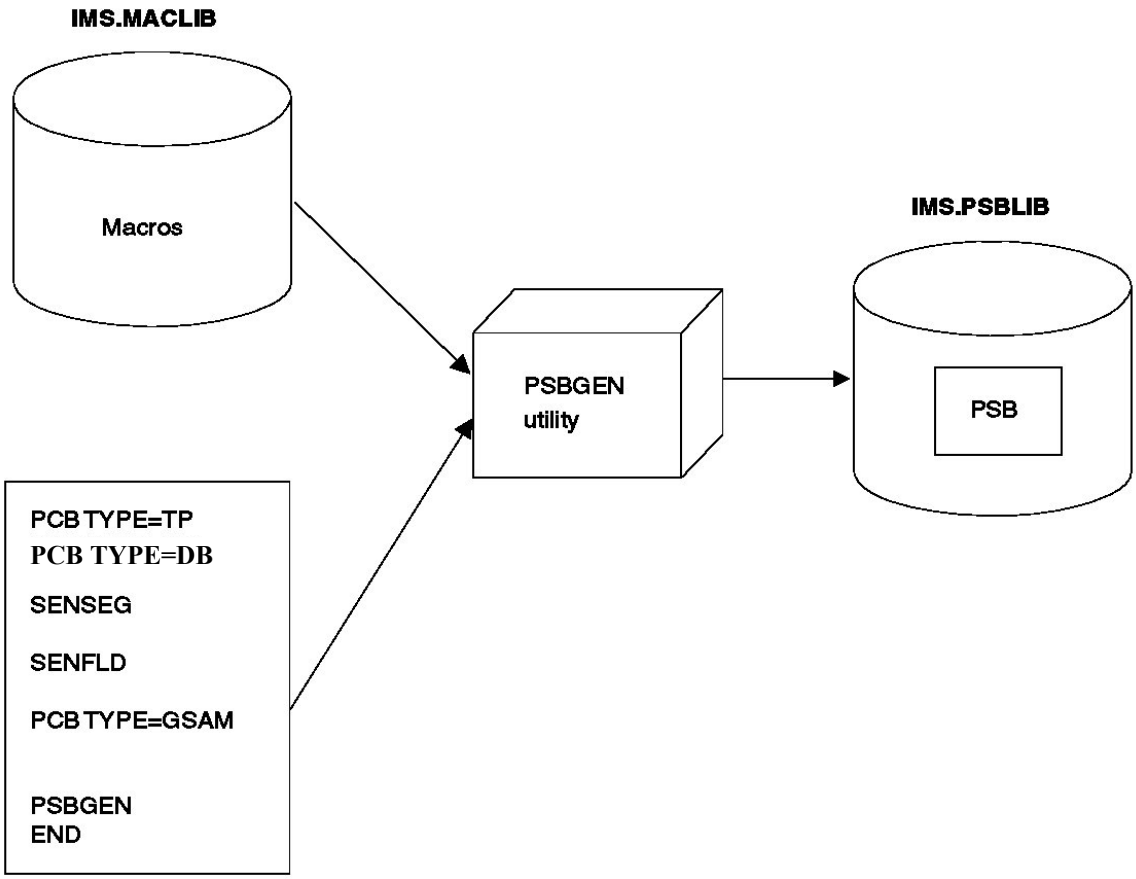
After you code the PSB macro instructions, they are used as input to the PSBGEN utility. This utility is a macro assembler that generates a PSB control block then stores it in the IMS.PSBLIB library for subsequent use during database processing.

Notes:-

1) Multiple PCB'S can be coded in a PSB
2) The PROCOPT can be coded at SENSEG level
3) KEYLEN specifies the longest possible concatenated key that is possible in the data base
4) Segment level sensitivity can be enforced using SENSEG macro
5) Field level sensitivity can be enforced using SENFLD macro
6) The SENSEG macros must be coded in hierarchical order
7) The PROCOPT at the PCB level will be used unless overridden by PROCOPT at segment level.
8) Type = GSAM can be coded for a GSAM data base.
9) You can ask for processing via a secondary index like below
10) PCB TYPE=DB,NAME=CT7DPIS1,PROCOPT=G,KEYLEN=21, PROCSEQ=CT7DKLNM , where CT7DKLNM is the DBD for the secondary index data base.
11) PROCOPT values

OPTION	FUNCTION
G	GET-Allows read only access to the segment
I	Insert-Allows add only (insert) access of the segment
R	Replace-Allows segment to be read and replaced
D	Delete-Allows segment to be read and deleted
A	All-Allows Get, Insertion, Replace, Delete of a segment
K	Key-Allows access to only the key of the segment
P	Path-Allows Path calls
L(S)	Load Sequential-Used to initially load the data base
GO	Get Only- Will not enqueue segments

PROCESS OF PSB CREATION



SKELETON JCL FOR PSB CREATION

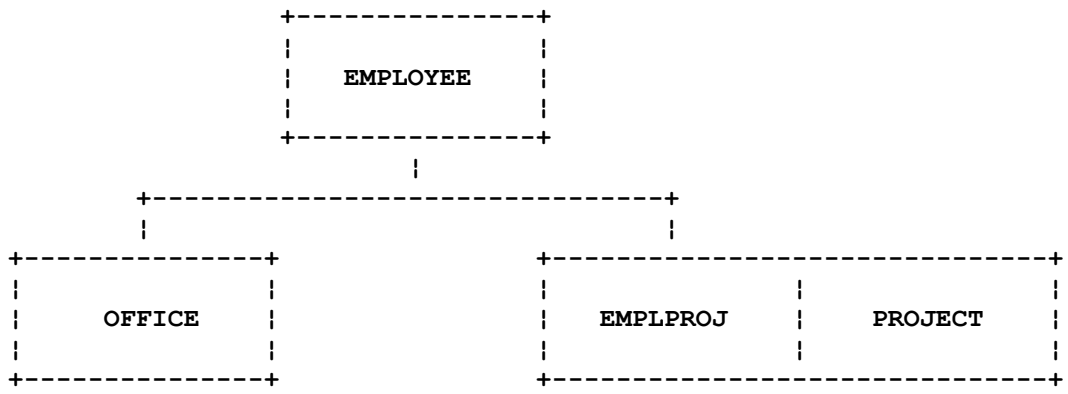
```
//PSBGEN    JOB  MSGLEVEL=1
//                EXEC        PSBGEN,MBR=APPLPGM1
//C.SYSIN       DD     *

    PCB TYPE=TP      required for output message destinations
    PCB TYPE=DP      required for each database the application program
                      can access
    SENSEG           required for each segment in the database the
                      application program can access
    SENFLD           required for each field in a segment that
                      the application program can access,
                      when field-level sensitivity is specified     .
    PCB TYPE=GSAM
        .
        .
        .
    PSBGEN           required for each PSB generation
    END              required for each PSB generation
/*
```

This example shows that a PSB generation is being performed for a batch message processing program. The GSAM PCB is used by the application program to generate a report file.

```
//PSBGEN   JOB  MSGLEVEL=1
//       EXEC   PSBGEN,MBR=APPLPGM3
//C.SYSIN   DD  *
     PCB       TYPE=DB,DBDNAME=PARTMSTR,PROCOPT=A,KEYLEN=100
     SENSEG    NAME=PARTMAST,PARENT=0,PROCOPT=A
     SENSEG    NAME=CPWS,PARENT=PARTMAST,PROCOPT=A
     PCB       TYPE=GSAM,DBDNAME=REPORT,PROCOPT=LS
     PSBGEN    LANG=COBOL,PSBNAME=APPLPGM3
     END
/*
```

Field Level Sensitivity

Figure below shows a PCB for a batch program using field level sensitivity.



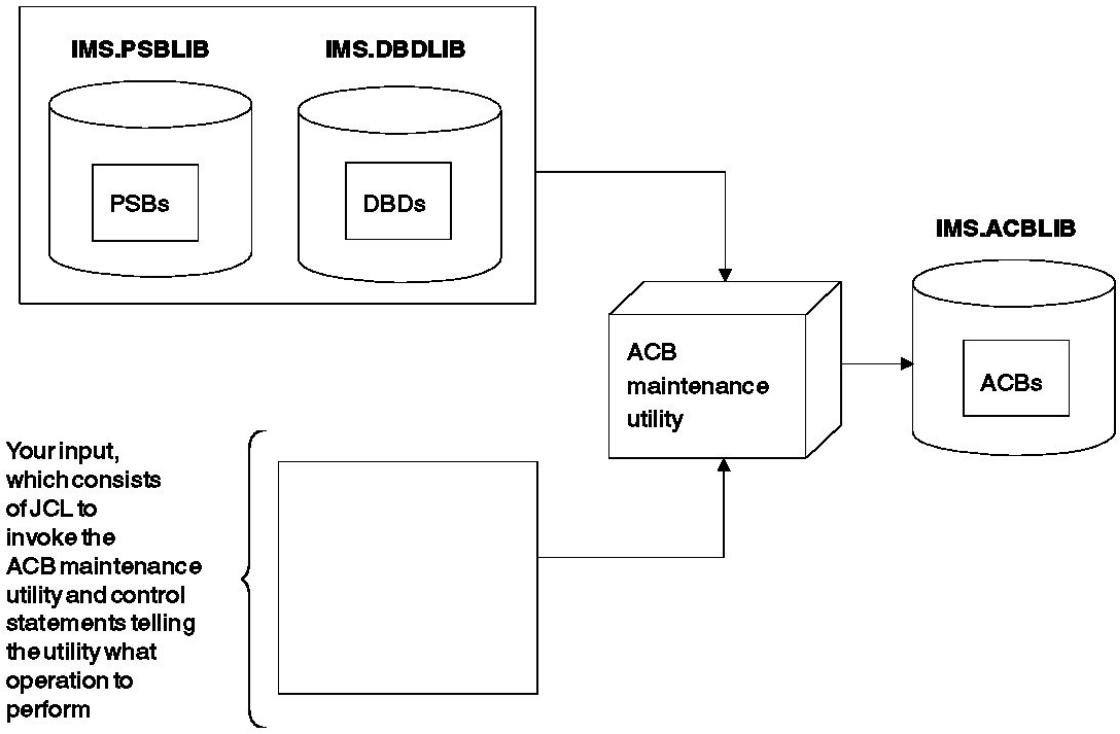
SEGMENT NAME	FIELD NAME	START LOCATION	LENGTH
EMPLOYEE	EMPSSN	1	9
	EMPLNAME	10	10
	EMPFNAME	20	9
	EMPMI	29	1
	EMPADDR	30	30
OFFICE	OFNUMBER	1	5
	OFPHONE	6	7
EMPLPROJ	EPFUNCTN	1	20
	EPTIMEST	21	5
	EPTIMCUR	26	5
PROJECT	PROJNUM	1	8
	PROJTTLE	9	20
	PROJSTRT	29	8
	PROJEND	37	8

```
PROJSTAT                45                  1
```

```
//PSBGEN   JOB  MSGLEVEL=1
//       EXEC    PSBGEN,MBR=APPLPGM1
//C.SYSIN   DD   *
     PCB     TYPE=DB,NAME=FISDBD1,PROCOPT=GRP,KEYLEN=20
     SENSEG   NAME=EMPLOYEE,PARENT=0
     SENFLD   NAME=EMPLNAME,START=13,REPL=NO
     SENFLD   NAME=EMPFNAME,START=1,REPL=NO
     SENFLD   NAME=EMPMI,START=11
     SENSEG   NAME=OFFICE,PARENT=EMPLOYEE
     SENSEG   NAME=EMPLPROJ,PARENT=EMPLOYEE
     SENFLD   NAME=PROJNUM,START=1
     SENFLD   NAME=PROJTTLE,START=10
     SENFLD   NAME=EPFUNCTN,START=35
     SENFLD   NAME=EPTIMEST,START=60
     SENFLD   NAME=EPTIMCUR,START =70
     PSBGEN   LANG=ASSEM,PSBNAME=FISPCB1
     END
/*
```

ACB Building the Application Control Blocks

IMS builds the ACB by merging information from the PSB and DBD. For execution in a batch environment, IMS can build ACBs either dynamically (PARM=DLI), or it can pre-build them using the ACB maintenance utility (PARM=DBB). ACB'S must be pre-built for use by online application programs.



You can have the utility pre build ACB'S for all PSB'S in IMS.PSBLIB, for a specific PSB, or for all PSB'S that reference a particular DBD. Pre built ACB'S are kept in the IMS.ACBLIB library. (IMS.ACBLIB is not used if ACB'S are not pre built.) When ACB'S are pre built and an application program is scheduled, the application program's ACB is read from IMS.ACBLIB directly into storage. This means that less time is required to schedule an application program. In addition, less storage is used if pre built ACB'S are used. Another advantage of using the ACB maintenance utility is the initial error checking it performs. It checks for errors in the names used in the PSB and the DBD'S associated with the PSB and, if erroneous cross-references are found, prints appropriate error messages.

IMS.ACBLIB has to be used exclusively. Because of this, the ACB maintenance utility can only be executed using an IMS.ACBLIB that is not currently allocated to an active IMS system. Also, because IMS.ACBLIB is modified, it cannot be used for any other purpose during execution of the ACB maintenance utility.

Control Block Building JCL

DBD Generation

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC DBDGEN,MBR=memname
//C.SYSIN  DD DSN=USER01.DBD.SOURCE(memname),DISP=SHR
//
```

PSB Generation

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC PSBGEN,MBR=memname
//C.SYSIN  DD DSN=USER01.PSB.SOURCE(memname),DISP=SHR
//
```

PSBGEN PROC

```
//        PROC MBR=TEMPNAME,SOUT=A,RGN=0M,SYS2=
//C       EXEC PGM=ASMA90,REGION=&RGN,
//             PARM='OBJECT,NODECK,NODBCS'
//SYSLIB   DD DSN=IMS.&SYS2.MACLIB,DISP=SHR
//SYSLIN   DD UNIT=SYSDA,DISP=(,PASS),
//         SPACE=(80,(100,100),RLSE),
//         DCB=(BLKSIZE=80,RECFM=F,LRECL=80)
//SYSPRINT DD SYSOUT=&SOUT,DCB=BLKSIZE=1089,
//         SPACE=(121,(300,300),RLSE,,ROUND)
//SYSUT1   DD UNIT=SYSDA,DISP=(,DELETE),
//         SPACE=(CYL,(10,5))
//L       EXEC PGM=IEWL,PARM='XREF,LIST',
//             COND=(0,LT,C),REGION=120K
//SYSLIN   DD DSN=*.C.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=&SOUT,DCB=BLKSIZE=1089,
//         SPACE=(121,(90,90),RLSE)
//SYSLMOD  DD DISP=SHR,
//         DSN=USER01.PSBLIB(&MBR)
//SYSUT1   DD UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),
//         SPACE=(1024,(100,10),RLSE),DISP=(,DELETE)
```

DBDGEN PROC

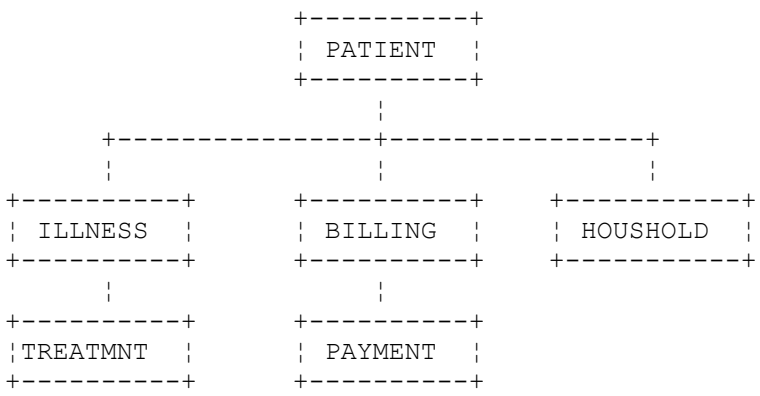
```
//        PROC MBR=TEMPNAME,SOUT=A,RGN=0M,SYS2=
//C       EXEC PGM=ASMA90,REGION=&RGN,
//             PARM='OBJECT,NODECK,NODBCS'
//SYSLIB   DD DSN=IMS.&SYS2.MACLIB,DISP=SHR
//SYSLIN   DD UNIT=SYSDA,DISP=(,PASS),
//         SPACE=(80,(100,100),RLSE),
//         DCB=(BLKSIZE=80,RECFM=F,LRECL=80)
//SYSPRINT DD SYSOUT=&SOUT,DCB=BLKSIZE=1089,
//         SPACE=(121,(300,300),RLSE,,ROUND)
//SYSUT1   DD UNIT=SYSDA,DISP=(,DELETE),
//         SPACE=(CYL,(10,5))
//L       EXEC PGM=IEWL,PARM='XREF,LIST',
//             COND=(0,LT,C),REGION=120K
```

```
//SYSLIN    DD DSN=*.C.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT  DD SYSOUT=&SOUT,DCB=BLKSIZE=1089,
//          SPACE=(121,(90,90),RLSE)
//SYSLMOD   DD DISP=SHR,
//          DSN=USER01.DBDLIB(&MBR)
//SYSUT1    DD UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),
//          SPACE=(1024,(100,10),RLSE),DISP=(,DELETE)
```

The Data Base you are going to create and use in the hands on

The examples are based on the Medical Database described in the IBM manual IMS/ESA V6 Appl Pgm: DB. The example on Logical Relationship described in a later chapter uses, in addition a Doctors data base. Both data bases are described below.

MEDICAL DATABASE



PATIENT SEGMENT, PATIENT has a unique key field: PATNO. PATIENT segments are stored in ascending order of their patient numbers.

PATNO	NAME	ADDR
5	10	30

ILLNESS SEGMENT, The key field is ILLDATE. Because it is possible for a patient to come to the clinic with more than one illness on the same date, this key field is not unique; For segments with equal keys or no keys, the RULES keyword determines where the segment is inserted. Where RULES=LAST, ILLNESS segments that have an equal key are stored on a first-in first-out basis among those with equal keys. ILLNESS segments with unique keys are stored in ascending order on the date field, regardless of RULES. ILLDATE is specified in the format YYYYMMDD.

ILLDATE	ILLNAME
8	10

TREATMNT SEGMENT, The key field of the TREATMNT segment is DATE. Because a patient may receive more than one treatment on the same date, DATE is a nonunique key field. TREATMNT, like ILLNESS, has been specified as having RULES=LAST. TREATMNT segments with equal date keys are also stored on a first-in-first-out basis. DATE is specified in the same format as ILLDATE--YYYYMMDD.

DATE	MEDICINE	QUANTITY	DOCTOR
8	10	4	10

BILLING SEGMENT, BILLING has no key field.

BILLING
6

PAYMENT SEGMENT,the PAYMENT segment has no key field.

PAYMENT
6

HOUSHOLD SEGMENT, RELNAME is the key field.

RELNAME	RELATN
10	8

The DBD for a HISAM Data Base

PNTDBHIS-----> DBD

```
DBD   NAME=PNTDBHIS,ACCESS=HISAM
DATASET DD1=PNTDBHIS,OVFLW=PNTOVFLW
SEGM NAME=PATIENT,BYTES=45,PARENT=0
FIELD NAME=(PATNO,SEQ,U),BYTES=5,START=1,TYPE=C
FIELD NAME=NAME,BYTES=10,START=6,TYPE=C
FIELD NAME=ADDR,BYTES=30,START=16,TYPE=C
SEGM  NAME=ILLNESS,BYTES=18,PARENT=PATIENT
FIELD NAME=(ILLDATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=ILLNAME,BYTES=10,START=9,TYPE=C
SEGM NAME=TREATMNT,BYTES=32,PARENT=ILLNESS
FIELD NAME=(DATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=MEDICINE,BYTES=10,START=9,TYPE=C
FIELD NAME=QUANTITY,BYTES=4,START=19,TYPE=C
FIELD NAME=DOCTOR,BYTES=10,START=23,TYPE=C
```

```
SEGM  NAME=BILLING,BYTES=6,PARENT=PATIENT
FIELD NAME=BILLING,BYTES=6,START=1,TYPE=C
SEGM  NAME=PAYMENT,BYTES=6,PARENT=BILLING
FIELD NAME=PAYMENT,BYTES=6,START=1,TYPE=C
SEGM  NAME=HOUSHLD,BYTES=18,PARENT=PATIENT
FIELD NAME=RELNAME,BYTES=10,START=1,TYPE=C
FIELD NAME=RELATN,BYTES=8,START=11,TYPE=C
DBDGEN
FINISH
END
```

PNTPHISL-----> Load PSB

```
PCB  TYPE=DB,NAME=PNTDBHIS,PROCOPT=LS,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PNTPHISL,LANG=PL/I
END
```

PNTPHISG-----> Add, Get, Replace, Delete PSB

```
PCB  TYPE=DB,NAME=PNTDBHIS,PROCOPT=A,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PNTPHISG,LANG=PL/I
END
```

IDCHIS-----> Idcams Job Stream to create the clusters

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
   DELETE USER01.PNTDBHIS CLUSTER
   DELETE USER01.PNTOVFLW CLUSTER
   DEFINE CLUSTER (NAME(USER01.PNTOVFLW) NONINDEXED -
   RECORDSIZE(52,52) TRACKS(1,1) CONTROLINTERVALSIZE(2048))
   DEFINE CLUSTER (NAME(USER01.PNTDBHIS) INDEXED KEYS(5,6) -
   RECORDSIZE(52,52) TRACKS(1,1)) DATA(CONTROLINTERVALSIZE(2048))
//
```

The DBD for a HDAM Data Base

PNTDBHD------> DBD

```
DBD  NAME=PNTDBHD,ACCESS=HDAM,RMNAME=(DFSHDC40,1,100,824)
DATASET DD1=PNTDBHD
SEGM NAME=PATIENT,BYTES=45,PARENT=0
FIELD NAME=(PATNO,SEQ,U),BYTES=5,START=1,TYPE=C
FIELD NAME=NAME,BYTES=10,START=6,TYPE=C
FIELD NAME=ADDR,BYTES=30,START=16,TYPE=C
SEGM  NAME=ILLNESS,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
```

```
FIELD NAME=(ILLDATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=ILLNAME,BYTES=10,START=9,TYPE=C
SEGM NAME=TREATMNT,BYTES=32,PTR=T,PARENT=((ILLNESS,SNGL))
FIELD NAME=(DATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=MEDICINE,BYTES=10,START=9,TYPE=C
FIELD NAME=QUANTITY,BYTES=4,START=19,TYPE=C
FIELD NAME=DOCTOR,BYTES=10,START=23,TYPE=C
SEGM  NAME=BILLING,BYTES=6,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=BILLING,BYTES=6,START=1,TYPE=C
SEGM  NAME=PAYMENT,BYTES=6,PTR=T,PARENT=((BILLING,SNGL))
FIELD NAME=PAYMENT,BYTES=6,START=1,TYPE=C
SEGM  NAME=HOUSHLD,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=RELNAME,BYTES=10,START=1,TYPE=C
FIELD NAME=RELATN,BYTES=8,START=11,TYPE=C
DBDGEN
FINISH
END
```

PNTPHDL------>The Load PSB

```
PCB  TYPE=DB,NAME=PNTDBHD,PROCOPT=L,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PSB2,LANG=PL/I
END
```

PNTPHDG------>The Add, Get, Replace and Delete PSB

```
PCB  TYPE=DB,NAME=PNTDBHD,PROCOPT=A,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PSB2,LANG=PL/I
END
```

IDCHD------>The Idcams Job Stream to create the cluster

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
   DELETE USER01.PATNTDB CLUSTER
   DEFINE CLUSTER (NAME(USER01.PATNTDB) NONINDEXED -
   RECORDSIZE(2041,2041) CONTROLINTERVALSIZE(2048) -
   TRACKS(2 2))
//
```

HIDAM EXAMPLE

PNTDBHI ------>The main data base DBD

```
DBD  NAME=PNTDBHI,ACCESS=(HIDAM,VSAM)
DATASET DD1=PNTDBHI
SEGM NAME=PATIENT,BYTES=45,PARENT=0
```

```
LCHILD NAME=(INDXSEG,PNTDBHII),PTR=INDX
FIELD NAME=(PATNO,SEQ,U),BYTES=5,START=1,TYPE=C
FIELD NAME=NAME,BYTES=10,START=6,TYPE=C
FIELD NAME=ADDR,BYTES=30,START=16,TYPE=C
SEGM  NAME=ILLNESS,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=(ILLDATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=ILLNAME,BYTES=10,START=9,TYPE=C
SEGM NAME=TREATMNT,BYTES=32,PTR=T,PARENT=((ILLNESS,SNGL))
FIELD NAME=(DATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=MEDICINE,BYTES=10,START=9,TYPE=C
FIELD NAME=QUANTITY,BYTES=4,START=19,TYPE=C
FIELD NAME=DOCTOR,BYTES=10,START=23,TYPE=C
SEGM  NAME=BILLING,BYTES=6,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=BILLING,BYTES=6,START=1,TYPE=C
SEGM  NAME=PAYMENT,BYTES=6,PTR=T,PARENT=((BILLING,SNGL))
FIELD NAME=PAYMENT,BYTES=6,START=1,TYPE=C
SEGM  NAME=HOUSHLD,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=RELNAME,BYTES=10,START=1,TYPE=C
FIELD NAME=RELATN,BYTES=8,START=11,TYPE=C
DBDGEN
FINISH
END
```

PNTDBHII------> The Index DBD

```
DBD  NAME=PNTDBHII,ACCESS=INDEX
DATASET DD1=PNTDBHII
SEGM NAME=INDXSEG,BYTES=5
LCHILD NAME=(PATIENT,PNTDBHI),INDEX=PATNO
FIELD NAME=(INDXSEQ,SEQ,U),BYTES=5,START=1,TYPE=C
DBDGEN
FINISH
END
```

PNTPHDIL ------> The load PSB

```
PCB  TYPE=DB,NAME=PNTDBHI,PROCOPT=LS,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PNTPHDIL,LANG=PL/I
END
```

PNTPHDIG ------> The Add, Get, Delete and Replace PSB

```
PCB  TYPE=DB,NAME=PNTDBHI,PROCOPT=A,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PNTPHDIG,LANG=PL/I
END
```

IDCHDI ------> The Idcams Job Stream

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
   DELETE USER01.PNTDBI CLUSTER
   DELETE USER01.PNTDBII CLUSTER
   DEFINE CLUSTER (NAME(USER01.PNTDBI) NONINDEXED -
   RECORDSIZE(2041,2041) CONTROLINTERVALSIZE(2048) -
   TRACKS(2 2))
   DEFINE CLUSTER (NAME(USER01.PNTDBII) INDEXED KEYS(5,5) -
   RECORDSIZE(10,10) TRACKS(1,1)) DATA(CONTROLINTERVALSIZE(1024))
//
```

Data Language/1

Index

When you complete this section, you will be able to do the following;

- Establish communication between your application program and IMS data bases through the use of entry points, PCB Points, and PCB mask structures.
- Request Data Language / I (DL/I) services as needed in your application program by issuing DL/I calls using the correct format.

To access IMS data bases, you must use a language to communicate your application program requests. The language developed by IBM to meet this need is Data Language /I.

DL/I is not a programming language, but a data access language. It is not unique to IMS. It can also be used within the CICS environment. DL/I provides all the necessary functions to support the maintenance of IMS data bases, such as add, changes and delete. Many functions, in addition to those for maintenance, are provided to take advantage of the IMS checkpoint, restart logging and online facilities, DL/I is designed to support hierarchicalal data base structures and associated relationships.

Before discussing D/I in detail, however, let's review some basic points.

Application program Linkage

The following two tasks must be performed to enable two way communication between DL/I and your application program;

- Your application program must establish an entry point and addressability to IMS resources.
- You must include the appropriate DL/I language interface in your program load module and provide the appropriate entry point.

Entry point and PCB Pointers

Figure below shows an example of how the entry points and addressability to IMS resources are specified in your application program.

PL/I

```
DLITPLI : PROC   ( pcb_name1_PTR.
                   pcb_name2_PTR.
                                •
                                •
                   pcb_namex_PTR ) OPTIONS (MATN);

DCL (pcb_name1_PTR, pcb_name2_PTR,….pcb_namex_PTR)  POINTER;
DCL   PLITDLI   EXTERNAL ENTRY;
```

COBOL

```
ENTRY  'DLITCBL"  USING L-pcb_name1,
                        L-pcb_name2,
                            •
                            •
                        L_pcb_namex.
```

Figure entry point and addressability to IMS resources

The entry name, DLITCBL, is required for COBOL, but the entry name of DLITPLI is optional for PL/I. The PCB Pointer names, pcb_name1 through pcb-namex , not only have a one to one relationship to the PCBs in the PSB used by this application program, but also must appear in the same order as the PCBs do in the PSB.

Linkage Editor

The control statements required for the Linkage Editor are reviewed below.

PL/I

```
//USER011   JOB…..
//STEPO10   EXEC     PGM =IEWL….
        •
        •
//RESLIB     DD   DSN=IMS  resident library….
//SYSLIN     DD   DSN=object module from compile
  LIBRARY RESLIB (PLITDLI)     ----dl/i  language interface
/*
```

COBOL

```
//USER011   JOB….
// STEP010  EXEC PGM =IEWL……
        •
        •
//RESLIB     DD   DSN=IMS RESIDENT LIBRARY,…
//SYSLIN     DD   DSN=object module from compile
//                  DD   *
  LIBRARY RESLIB (CBLTDLI)   -----dl/i language interface
  ENTRY DILITCBL
/*
```

FIGURE linkage Editor control statements

Remember that when DL/I services a request from your program, it places information into the PCB mask to which your program has addressability, and may also place data in the segment I/O area. It is up to the application program to provide I/O areas to store this information.

PCB Mask I/O Area

Figure below is an example of the data base PCB mask. The PCB Mask structure must appear in the linkage section of a COBOL program. There must be a structure for each PCB in the list. The 01 level name, for COBOL must be the same one used in the USING clause of the ENTRY statement. PL/I PCB mask structures must be based. You should use the pointer names specified in the main PROCEDURE statement as the based variable in the structure.

PL/I

```
DCL  PCB_PTR                                  POINTER;
DCL  01    PCB_NAME BASED (PCB_PTR),
       05 DBD_NAME                            CHAR (08),
       05 SEGMENT_HIERARCHY_LEVEL_IO          CHAR (02),
       05 DLI_STATUS_CODE                     CHAR (02),
       05 DLI_PROCESSING_OPTIONS              CHAR (04),
       05 RESERVED_FOR _DLI                   CHAR (04),
       05 SEGMENT_NAME_FEEDBACK_AREA          CHAR (08),
       05 LENGTH_OF_KEY_FEEDBACK_AREA         FIXED  BIN (31),
       05 NUMBER_OF _SENSITIVE _SEGMENTS      FIXED  BIN (31),
       05 KEY_FEEDBACK_AREA                   CHAR (NN);
```

COBOL

```
01   L-PCB-NAME,
       05     L-DBO-NAME                      PIC  X (08).
       05     L-SEGMENT-HIERARCHY-LEVEL-ID    PIC  X (02).
       05     L-DLI-STATUS-CODE               PIC  X (02).
       05     L-DLI-PROCESSING-OPTIONS        PIC  X (04).
       05      FILLER                         PIC  X (04).
       05     L-SEGMENT-NAM-FEEDBACK-AREA     PIC  X (08).
       05     L-LENGTH-OF-KEY-FEEDBACK-AREA   PIC  S9 (09)    COMP.
       05     L-NUMBER-OF-SENSITIVE-SEGMENTS  PIC  S9 (09)    COMP.
       05     L-KEY-FEEDBACK-AREA             PIC  X (nn).
```

PCB mask

The following information describes each data element in the PCB mask;

DBD Name – This field contains the DBD name of the data base accessed in the DL/I request The field length must be character eight bytes long.

Segment Hierarchy Level ID – This field contains the lowest level number in the hierarchy that DL/I accessed to satisfy your request It must be character two bytes long.

DL/I Status Code –This field contains a code that indicates the success or failure of the DL/I request. It must be character two bytes long.

DL/I Processing Options-This field contains the processing options specified in the PROCOPT=field of the PCB for the requested segment It must be character four bytes long. Processing options specified in the SENSEG macro will not show up here.

Reserved for DL/I – This field is reserved for DL/I use and is four bytes long. COBOL Programmers sometimes designate this as FILLER PIC X(4).

Segment Name Feedback Area – This field contains the name of the segment returned by DL/I. It must be character eight bytes long.

Length of Key Feedback Area – This field contains length of the key feedback area. It must be binary four bytes (full word) long;

Number of Sensitive Segments –This field contains the number of SENSEG macros found within the PCB (data base) accessed via the DL/I request It is binary four bytes (full Word)long.

Key Feedback area – This field contains the concatenated keys from all the segments retrieved down the hierarchical path up to and including the retrieved segment;

Your application will never insert or change any data in the PCB mask. DL/I always fills in the PCB mask after each request.

It is also important to check the status code of the call immediately after it is issued, In all cases, a status code of two blanks means the DL/I call was successful. If the status code is not blank you should take appropriate action in your application program. The common status codes for each type of DL/I call will be illustrated in the topics to follow.

PL/1

```
/*****************************************************/
/*   I/O AREA FOR ROOT SEGMENT (UNSTRUCTURED)      */
/*****************************************************/
DCL RENTAL_ITEM                                  CHAR (62);
/*****************************************************/
/*I / O AREA FOR ROOT SEGMENT STRUCTURE             */
/*****************************************************/
DCL 01 RENTAL_ITEM_STRUCTURE,
          05    RI_ITEM_NUMB                     CHAR (06),
          05    RI_CATEGORY                      CHAR (01),
          05    RI_COLOR                         CHAR (08),
          05    RI_MANUFAC_NAME                  CHAR (15),
          05    RI_MANUFAC_SERIAL                CHAR (20),
          05    RI_SIZE                          CHAR (10),
          05    RI_STORE                         CHAR (02);
```

COBOL

```
WORKING STORAGE SECTION
************************************************
* I/O AREA FOR ROOT SEGMENT (UNSTRUCTURED)   *
************************************************
 01    RENTAL –ITEM – STRUC                PIC X (62).
************************************************
* I/O AREA FOR ROOT SEGMENT (UNSTRUCTURED)   *
************************************************
01  RENTAL-ITEM-STRUC.
    05 RI_ITEM_NUMB                        PIC X (06).
    05 RI_CATEGORY                         PIC X (01).
    05 RI_COLOR                            PIC X (08).
    05 RI_MANUFAC_NAME                     PIC X (15).
    05 RI_MANUFAC_SERIAL                   PIC X (20).
    05 RI_SIZE                             PIC X (10).
    05 RI_STORE                            PIC X (02).
```

Segment I/O Areas

Many installations have predefined structures with INCLUDE or COPY members. You should determine whether such structures are available through your installation before you spend time coding them.

Figure below shows sample programs with all of the necessary components for communicating with DL/I.

PL/I

```
RSITH0I :  PROC (CT7DPDA1_PTR)     /* RENTAL  DATA BASE  PCB POINTER */
                  OPTIONS  (MAIN) REORDER;

DCL  CT7DPDA1_PTR        POINTER:
DCL  ADDR                BUILTIN:

/* ESTABLISH LINKAGE BETWEEN DL/I  AND  PROGRAM*/
DCL  PLITDLI                EXTERNAL   ENTRY;

/*   DATA BASE  PCB MASK*/
DCL  01  PCBNAME                          BASED   (CT7DPDA1_PTR),
          05     DBD_NAME                 CHAR (08),
          05     SEG_NAME                 CHAR (02),
          05     DLI_STATUS_CODE          CHAR (02),
          05     DLI_PROC_OPTIONS         CHAR (04),
          05     RESERVED_FOR_DLI         CHAR (04),
          05     SEGMENT _NAME            CHAR (08),
          05     LTH_KEY_FEEDBACK         FIXED BIN (31),
```

```
        05    NUM_SENSEGS              FIXED BIN (31),
        05    KEY_FEEDBACK_AREA        CHAR (21);

/* SEGMENT I/O  AREA DEFINITIONS    */
DCL SEGMENT_IO_AREA                CHAR (119)      INIT(‘  ’);

DCL  01  RENTAL_ITEM_SEG           BASED (ADDR(SEGMENT_IO_AREA)),
   05    RI_ITEM_NUMB               CHAR (06),
   05    RI_CATEGORY                CHAR (01),
   05    RI_COLOR                   CHAR (08),
   05    RI_MANUFAC_NAME            CHAR (15),
   05    RI_MANUFAC_SERIAL          CHAR (20),
   05    RI_SIZE                    CHAR (10),
   05    RI_STORE                   CHAR (02);

END;    /*  RSITM01 */
```

Part 1 of 2 – application program linkage

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID  RSITEMOI.
REMARKS.  SAMPLE COBOL IMS  PROGRAM.


ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE –CONTROL.

DATA DIVISION.
FILE SECTION.
WORKING –STORAGE SECTION.
01  SEGMENT–IO–AREA               PIC  X (119)  VALUE SPACES.

01  RENTAL–ITEM-SEG  REDEFINES SEGMENT–IO–AREA.
      05    RI-ITEM-NUMB           PIC X (06).
      05    RI-CATEGORY            PIC X (01).
      05    RI-COLOR               PIC X (08).
      05    RI-MANUFAC-NAME        PIC X (15).
      05    RI-MANUFAC-SERIAL      PIC X (20).
      05    RI-SIZE                PIC X (10).
      05    RI-STORE               PIC X (02).

LINKAGE SECTION.
01  L-CT7DPDA1.
      05    L-DBD-NAME             PIC X (08).
      05    L-SEG-HIER-LEVEL_ID    PIC X (02).
```

```
05    L-DLI-STATUS-CODE          PIC X (02).
05    L-DLI-PROC-OPTIONS         PIC X (04).
05    FILLER                     PIC X (04).
05    L-SEGMENT-NAME             PIC X (08).
05    L-LTH-KEY-FEEDBACK         PIC S9 (05)   COMP.
05    L-NUM-SENSEGS              PIC S9 (05)   COMP.
05    L-KEY-FEEDBACK-AREA        PIC X (21).

PROCEDURE  DIVISION.
*
*    ESTABLISH LINKAGE BETWEEN DL/I  AND JPROGRAM
*
       ENTRY  'DLITCBL'  USING L-CT7DPDA1.

 GOBACK.
```

Part 2 of 2-application Program Linkage

DL/I Call

A DL/I request in your application program is DL/I call. The actual DL/I call structure has a very specific format. Figure below is an example of those formats.

PL/I

```
CALL  PLITDLI (parm_count.
               function,
               pcb_name,
               segment_io,
               ssa_1,
                 •
                 •
               ssa_15);
```

COBOL

```
CALL   'CBLTDLI'   USING    parm_count,
                            function,
                            pcb-name,
                            segment-io,
                            ssa- 1,
                            •
                            •
                            ssa – 15.
```

structure of a DL/I call

These calls represent your application requesting DL/I services. You must pass the parameters to DL/I in the sequence shown in the Figure above.

In PL/I or COBOL, you must issue the CALL statement followed by the language interface. Use PLITDLI for PL/I and CBLTDLI for COBOL, you must define a series of parameters that are discussed on the following pages.

Parm Count

The parm count is a binary full word that has the value equal to the number of parameters passed to the DL/I call, not including the parm count itself. This is a required parameter for PL/I but is optional for COBOL.

Function code

The function code parameter is required in all languages, It must be a character field, four bytes long. It specifies the type of function DL/I should perform on the data base segment, for example retrieve, add, replace, delete, and so on. The table describes common DL/I function codes and their function descriptions.

DL/I Function code	Function Description
GU	Get Unique – Allows direct retrieval of segment.
GHU	Get Hold Unique – Allows direct retrieval of a segment with segment locking.
GN	Get Next – Allows sequential retrieval of a segment.
GHN	Get Hold Next – Allows Sequential retrieval of a segment with segment locking.
GNP	Get Next within Parent – Allows sequential retrieval of a segment under the previously retrieved parent segment.
GHNP	Get Hold Next within Parent – Allows sequential retrieval of a segment under the previously retrieved parent segment with segment locking.
ISRT	Insert – Allows the adding of a segment occurrence.
REPL	Replace – Allows the data in a segment to be replaced by new data.
DLET	Delete – Allows the deletion of a segment occurrence and all segments below it in the hierarchy.

Common DL/I Function Codes Table

PCB NAME

The PCB name identifies the data base DL/I is accessing. This name must be the same one used in the main PROCEDURE statement of PL/I, and the same ones used in the LINKAGE SECTON and ENTRY statement of a COBOL. This is a required parameter in all DL/I calls. When an application uses multiple data bases, it is through this parameter that DL/I knows against which data base the call is being issued.

Segment I/O

This specifies the I/O area variable (or structure), name for the call. DL/I will use this name to get segment data from the I/O area or to put segment data into it. This Parameter is required on all DL/I data base calls.

Segment Search Argument

This specifies the I/O area structure name for the segment search argument, (SSA) DL/I uses to satisfy your request. This parameter is optional. You can specify as many as 15 SSA'S in a single DL/I call.

Examples of two DL/I calls.

PL/I

```
/* GET UNIOUE CALL */
  CALL PLITDLI   (  C_FOUR,          <------  FIXED BIN (31) INIT (4)
                    C_GU,            <------  CHAR (4)       INIT (‘GU‘)
                    CT7DPOA1_PTR,    <------  POINTER
                    SEG_IO,
                    ROOT_SSA_Q   );

/*  INSERT CALL */
   CALL    PLITDLI   (  C_FOUR,
                        C_ISRT,      <------  CHAR (4)       INIT (‘ISRT‘)
                    CT7DPDA1_PTR,
                    SEG_IO,
                    ROOT_SSA_U   );

/*  DELETE CALL */
   CALL    PLITDLI   (  C_THREE,    <------  FIXED BIN (31) INIT (3)
                        C_DLET,     <------  CHAR (4)       INIT (‘ISRT‘)
                    CT7DPDA1_PTR,
                        SEG_IO );
```

COBOL

```
*  GET UNIOUE CALL
*
  CALL  ‘CBLTDLI’  USING  C-FOUR,      <------ S9(09) COMP VALUE +4 (IF USED)
                          C-GU,        <------ PIC  X (4)  VALUE ‘GU’
                          L-CT7DPDA1,  <------ LINKAGE SECTION VARIABLE
                          SEG-IO,
                          ROOT-SSA-Q.


*INSERT CALL
*
    CALL ‘CBLTDLI’ USING   C-ISRT,     <------  PIC  X (4)  VALUE ‘ISRT’
                           L-CT7DPDA1,
                       SEG-IO,
                           ROOT-SSA-U.
*DELETE CALL
*
CALL  ‘CBLTDLI’  USING     C-DLDT,     <------ PIC  X (4)  VALUE ‘DLET’
                           L-CT7DPDA1,
                           SEG-IO.
```

DL/I Call Examples

Unqualified Call

The SSA is an optional part of the DL/I call statement. If no SSA(S) are specified, DL/I considers the call to be unqualified. This means that you are not giving DL/I any specific segment type or segment occurrence. DL/I's response to this request is based on the following criteria;

- Sensitive segments defined in the PSB
- Processing options for defined segments in the PSB
- Its current Position in the database
- The type of call issued (GU, GN, DLET, or ISRT)

Unqualified sequential retrieval calls always retrieve the next segment occurrence in the data base in hierarchicalal sequence, based on DL/Is current position in the data base. An unqualified GU or GHU always retrieves the first segment occurrence in the data base, regardless of its current position. Stated simply, an unqualified GU or GHU call always retrieves the first root segment in the data base. This is a quick and practical method of repositioning DL/I to the beginning of a data base.

The ISRT must have at least one SSA. The SSA must be for the segment being inserted, called the TARGET segment It must be unqualified. If an unqualified ISRT call is issued, DL/I will return an AH (Missing SSA) Status code.

REPL and DLET calls usually do not require SSA'S. It is normal for these calls to be unqualified.

Qualified call

If one or more SSA s are specified in a call, DL/I considers the call to be qualified, DL/I will honor the SSA'S request for a specific segment type and return that segment's data to the application program.

It is good Programming practice to qualify all of your DL/I calls and to specify SSA'S for each segment type down the path to the target segment. The only exceptions are the replace and delete calls because they typically do not use SSA'S. It is also recommended that you qualify all of your SSA'S, when possible. By providing as much specific information in the call, you ensure that DL/I performs exactly as you wish.

The segment search argument is a set of parameters passed to DL/I. It specifies the segment type in the data base you wish to process. You can request a specific occurrence (Key value or search field value) using a qualified SSA. You can specify many SSA'S in a DL/I call. If you do, the sequence of the SSA list must represent a valid hierarchical sequence. When multiple SSA'S are used, the last SSA in the list is called the target SSA. When DL/I returns

segment data to your application program, it returns only the data for the target SSA. Similarly, when DL./I is performing maintenance on a data base segment (ISRT, REPL or DLET), only the target SSA segment data is affected. Exceptions to this rule are explained more fully in the later section on command Codes.

The basic format of a qualified and unqualified SSA is shown below.

PL/I

```
DCL  01  ITEM_SSA_Q,
           05  SEGMENT_NAME              CHAR (08)  INIT ('RSITEMSR'),
           05  QUALIFIER                 CHAR (01)  INIT  ('('),
           05  FIELD_NAME                CHAR (08)  INIT ('ITEMNUMB'),
           05  RELATIONAL_OPERATOR       CHAR (02)  INIT ('  ='),
           05  FIELD_VALUE               CHAR (06)  INIT ('  '),
           05  RIGHT_PAREN               CHAR (01)  INIT (')');

DCL  01  ITEM_SSA_U,
           05  SEGMNT_NAME               CHAR (08)  INIT ('RSITEMSR'),
           05  QUALIFIER                 CHAR (01)  INIT ('  ');
```

COBOL

```
01   ITEM-SSA-Q.
       05  IS-SEGMENT-NAME              PIC  X (08)  VALUE  'RSITEMSR'.
       05  IS-QUALIFIER                 PIC  X (01)  VALUE  '('.
       05  IS-FIELD-NAME                PIC  X (08) VALUE  'ITEMNUMB'.
       05  IS-RELATIONAL-OPERATOR       PIC  X (02)  VALUE  '  ='.
       05  IS-FIELD-VALUE               PIC  X (06)  VALUE  '  '.
       05  IS-RIGHT-PAREN               PIC  X (01)  VALUE  ')'.

01   ITEM-SSA-U.
       05  IS-SEGMENT-NAME              PIC  X (08)  VALUE 'RSITEMSR'.
       05  IS-QUALIFIER                 PIC  X (01)  VALUE  '  '.
```

Structure of a Basic SSA

A structure is normally used to define an SSA. Your application program typically contains at least one SSA for every segment that you need to access. Often, the SSA structures are predefined in INCLUDE or COPY members. Lets discuss each element of the SSA structure.

Segment Name- This field must be eight characters long and contain the name of the segment to be accessed in the data base. This name must be defined in the SEGM macro of the DBD.

DBD SOURCE CODE

```
DBD . . .
DATASET . . .
SEGM      NAME=RSITEMSR, . . . .
   •
   •
```

APPLICATION PROGRAM SOURCE

```
DCL  01  SSA_STRUCTURE,
           05  SEGN     CHAR (8)  INIT ('RSITEMSR'),
           05  .
           05  .
```

Qualifier – This field is one character long and must contain either a blank or a left parenthesis, '(' . If a blank is specified, DL/I will consider the SSA unqualified. If a left parenthesis is specified, DL/I will consider the SSA qualified.

Field Name-This field is eight characters long. It must contain the name of a field specified in a FIELD macro of the DBD for the segment specified in the segment name of the SSA

DBD SOURCE CODE

```
DBD . . . .
DATASET . . . .
SEGM      NAME=RSITEMSR . . . .
FIELD    NAME= (ITEMNUMB,SEQ,U) . . . .
   •
   •
```

APPLICATION PROGRAM SOURCE

```
DCL  01  SSA_STRUCTURE,
           05  SEGN     CHAR (8)  INIT ('RSITEMSR'),
           05  QUAL     CHAR (1)  INIT ('('),
           05  FIELD    CHAR (8)  INIT ('ITEMNUMB'),
           05
           05
```

Relational operator –This field is two characters long and specifies the relational value of the FIELD to its VALUE. A table of the allowable relational operators follows:-

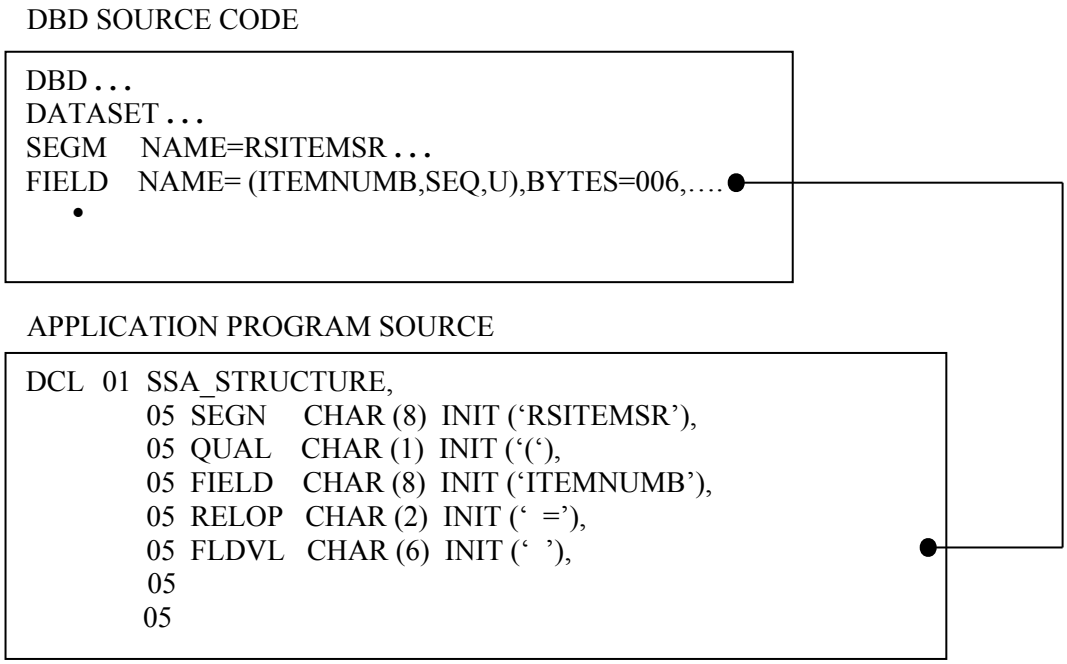
Operators	Relational Meaning
=b or b = or EQ	Must be equal to
>= or => or GE	Must be greater than or equal to
<= or =< or LE	Must be less than or equal to

>b or b> or GT	Must be greater than
<b or b< or LT	Must be less than
Ø= or =Øor NE	Must not be equal to

Note: Lowercase B (b) indicates a blank.

SSA Relational operators Table

Field value – This field specifies the precise value of the field name specified in the SSA. Its length depends upon that specified in the bytes keyword of the FIELD macro in the DBD. Its type depends on the TYPE= specified in the DBD. See the following figure.



Right Paren – This field is one character long and specifies the right parenthesis character,). This signals the termination of the SSA.

The basic SSA structures is used most of the time. However, more complex SSA structures Will be discussed in Advanced SSA Concepts.

Unqualified SSA

DL/I interprets an SSA as unqualified if position nine in a basic SSA is blank. This means that everything specified in the SSA after position nine is ignored by DL/I. This causes DL/I to move to the next segment occurrence of the specified SEGMENT name in the first eight positions in the SSA. The unqualified SSA is used when sequential processing of a segment type is required, or if the application does not have a field value to specify for the call.

Qualified SSA

If the left parenthesis character is in position nine in a basic SSA, DL/I considers the SSA qualified. All fields in the SSA following position nine are considered in the call. Qualified SSA'S are used to directly process a specific instance of the segment type. This means that a qualified SSA tells DL/I to find, not only a specific segment type, but also a specific instance of that type where the logical condition specified in the SSA is satisfied. The logical conditions are indicated by the FIELD NAME and FIELD VALUE fields of the SSA.

Adding Data to an IMS Data Base

Index

The DBA defines your data base, then loads it . In this class however **YOU** are going to define, create and load the data base. There are two methods of loading a data base. Both methods require using the insert call. The insert call is function code, ISRT, It adds a new occurrence of the segment type specified in the SSA. You must insert segments in hierarchical sequence, and the SSA must be unqualified for the segment type you are inserting. For example, you cannot add a dependent segment type, such as RSRENTER if its parent segment type, RSITEMSR, has not been previously added. This is because a valid child segment cannot exist without its parent Figure below shows some examples of the ISRT call.

Regardless of how the DL/I call is issued in your program, notice that each element of the DL/I call relates back to the application program.

PL/I

```
RSITMO1:  PROC  (CT7DPDA1_PTR)   /*ITEM DATA BASE  PCB  POINTER*/
              OPTIONS (MAIN);
          •
          •
/*****************************************************************/
/* PCB MASK I/O AREA                                             */
/*****************************************************************/
DCL  01  PCB_MASK BASED (CT7DPDA1_PTR),
             05  DBD_NAME                          CHAR(08),
             05  SEGMENT_HIERARCHY_LEVEL_ID        CHAR (02),
             05  DLI_STATUS_CODE                   CHAR (02),
             05  DLI_PROCESSING_OPTIONS            CHAR (04),
             05  RESERVED_FOR_DLI                  CHAR (04),
             05  SEGMENT_NAME_FEEDBACK_AREA        CHAR (08),
             05  LENGTH_OF_KEY_FEEDBACK_AREA       FIXED  BIN (31),
             05  NUMBER_OF_SENSITIVE_SEGMENTS      FIXED  BIN (31),
             05  KEY_FEEDBACK_AREA                 CHAR (21);

/*ESTABLISH LINKAGE BETWEEN  DL/I  AND  PL/I */
DCL  PLITDLI              EXTERNAL ENTRY;
DCL  CT7DPDA1_PTR         POINTER;
DCL  C_4                  FIXED  BIN (31,0)  INIT(4);
DCL  C_ISRT               CHAR (4)  INIT ('ISRT');
          •
/*************************************************/
/*SSA STRUCTURES                                 */
/*************************************************/
DCL  01  ITEM_SSA_U,
  05 SEGMENT_NAME      CHAR(08)   INIT ('RSITEMSR'),
  05 QUALIFIER         CHAR (01)  INIT ('  ');
```

```
        •
        •
/*************************************************/
/*SEGMENT   I/O AREA                             */
/*************************************************/
DCL   01    RENTAL_ITEM_SEG,
    05  RI_ITEM_NUMB              CHAR (06),
    05  RI_CATEFGORY              CHAR (01),
    05  RI_COLOR                  CHAR (08),
    05  RI_MANUFAC_NAME           CHAR (15),
    05  RI_MANUFAC_SERIAL         CHAR (20),
    05  RI_SIZE                   CHAR (10),
    05  RI_STORE                  CHAR (02),
        •
        •
/*****************************************************/
/*MAIN PROGRAM LOGIC                                 */
/*****************************************************/


/* ISRT  REOUIBES UNOUALIFIED TARFGET SSA */
CALL PLITDLI   (C_4,                    /* PARM COUNT                */
                C_ISRT,                 /* DL/I CALL FUNCTION CODE   */
                CT7DPDA1_PTR,           /* DATA BASE PCB POINTER     */
                RENTAL_ITEM_SEG,        /* SEGMENT  I/O  AREA        */
                ITEM_SSA_U);            /* SSA FOR ITEM SEGMENT      */

1F  DLI_STATUS_CODE =  ‘  ’THEN
        •
        •
ELSE
        •
        •
```

Part 1 of 2 DL/I Insert call (ISRT)

COBOL

```
IDENTIFICATION  DIVISION.
PROGRAM-ID. RSITEM01.
REMARKS. SAMPLE DL/I INSERT PROGRAM.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        •
        •
DATA DIVISION. FILE SECTION.
        •
        •
```

```
 WORKING-STORAGE SECTION.
 01   C-ISRT                PIC X(04)   VALUE 'ISRT'.
 01   C-4                   PIC S9(05)  COMP VALUE +4.
          •
          •
*****************************************************************
*   SSA STRUCTURES
*****************************************************************
  01  ITEM-SSA-U .
      05  SEGMENT-NAME              PIC X(08) VALUE 'RSITEMSR'.
      05  LEFT-PAREN                PIC X(01) VALUE  '  '.
*****************************************************************
*  SEGMENT I/0 AREA
*****************************************************************
 01   ITEM-SEGMENT-DATA.
      05  IS-ITEM-NUMB              PIC X(06).
      05  IS-CATEGORY               PIC X(01).
      05  IS-COLOR                  PIC X(08).
      05  IS-MANUFAC-NAME           PIC X(15).
      05  IS-MANUFAC-SERIAL         PIC X(20).
      05  IS-SIZE                   PIC X(10).
      05  IS-STORE                  PIC X(02).
          •
          •
 LINKAGE SECTION.
*****************************************************************
*     PCB MASK I/0 AREAS
*****************************************************************
 01   L-CT7DPDA1.
      05  DBD-NAME                        PIC X(08).
      05  SEGMENT-HIERARCHY-LEVEL-ID      PIC X(02).
      05  DLI-STATUS-CODE                 PIC X(02).
      05  DLI-PROCESSING-OPTIONS          PIC X(04).
      05  RESERVED-FOR-DLI                PIC X(04).
      05  SEGMENT-NAME-FEEDBACK-AREA        PIC X(08).
      05  LENGTH-OF-KEY-FEEDBACK-AREA       PIC S9(05) COMP.
      05  NUMBER-OF-SENSITIVE-SEGMENTS      PIC S9(05) COMP.
      05  KEY-FEEDBACK-AREA               PIC X (21).

 PROCEDURE DIVISION.
*****************************************************************
*     ESTABLISH LINKAGE BETWEEN DL/I ANND COBOL
*****************************************************************
         ENTRY  'DLITCBL'  USING   L-CT7DPDA1.
          •
          •
         MOVE  '  '  TO LEFT-PAREN.
*     PARM COUNT IS OPTIONAL FOR COBOL
```

```
CALL 'CBLTDLI' USING C-4,
                     C-ISRT,
                     L-CT7DPDA1,
                     ITEM-SEGMENT-DATA,
                     ITEM-SSA-U.
IF DLI-STATUS-CODE= SPACES
    •
    •
ELSE
    •
    •
```

Part 2 of 2 DL/I Insert Call (ISRT)

Using Multiple SSA'S in an ISRT Call

When using multiple SSA'S in an ISRT call, the last SSA is the target and must be unqualified. When multiple SSA'S are used, higher level SSA'S may be qualified or unqualified. When inserting new segment occurrences, you must be sure that a dependent segment is inserted under the correct parent segment.

DL/I maintains its position in a data base based on the last DL/I call made. This is an important consideration when inserting new segment occurrences. Consider the following example:

Your application program was processing some input from a sequential file. The file contained maintenance data, and some adds, changes, and deletes. Suppose a change was applied to a root segment with a key value of 123456. The next transaction requires adding a new dependent segment for the root segment. The root segment's key value is 123678. If you issue an ISRT call specifying only an SSA for the dependent segment, DL/I will insert that dependent under root 123456 because it is the last known position of a root segment. This is not desirable. The correct way to issue the ISRT call would be with two SSA'S. The first SSA in the list would be qualified for the root whose key value is 123678. This enables DL/I to reposition itself in the data base to the correct root segment. The second, and last SSA would be unqualified as required by DL/I, for the dependent segment.

Adding Data via Load Mode

As mentioned earlier, there are two ways to load data into an IMS data base. The PSB processing options determine the method that will be used.

The preferred method of loading an IMS data base is via the load mode. This requires that the DBA set up a PSB with a processing option of L(LOAD) or LS (LOAD SEQUENTIAL) at the PCB level. When your application program issues an ISRT call and the processing option is L or LS, DL/I performs high speed insertions of the segments. It is essential that the data be presorted in hierarchical sequence to prevent an error occurrence during the load.

Adding Data Via Insert Mode

A less efficient method of loading data into an IMS data base is via the insert mode. This requires that the DBA create a PSB that has a processing option of I (INSERT) at either the PCB or SENSEG level for the segment types being inserted. The data does not need to be sorted prior to loading, but the rules of hierarchically inserting still apply. This means that you cannot insert a child before its parent has been inserted and you are positioned below it. However, you do not need to insert segment occurrences in key sequence, as you do in the load mode.

Common Status Codes

The following are tables of common status codes found in both the load mode and the insert mode.

Load Mode

Status	Description
(blank)	Call was successful.
LB	Duplicate Key found for segment.
LC	Key field not in sequence.
LD	Parent segment does not exist.
LE	Sibling segments not in sequence.

Insert Mode

Status	Description
(blank)	Call was successful.
II	Duplicate Key found for segment.
GE	Parent segment does not exist.
V1	Length invalid for a variable length segment.

Data Base Load Lab Exercise

If you have prepared the DBD'S, PSB'S and PLIPGM5 described in earlier chapters, you have all the tools needed to load in turn, your HISAM, HDAM and HIDAM data bases. Before you do this ensure you have completed the following activities:-

1. Understood the DBD source for each of the three sample data bases above.

2. Prepared the DBD'S

3. Understood the PSB source for each of the three sample data bases above. Two PSB'S need to be created for each data base. One for loading and the other for ongoing data base access

4. Prepared the PSB'S

5. Read and understood PLIPGM5 which is the load program

6. Lets take the HISAM data base as an example. You should have the following artifacts and created the following data sets. Note that USER01 is assumed as the userid:-

   a) The DBD library USER01.DBDLIB

   b) The PSB Library USER01.PSBLIB

   c) The Load Library USER01.LOADLIB

   d) The DBD PNTDBHIS in the library USER01.DBDLIB.

   e) The load PSB PNTPHISL in USER01.PSBLIB

   f) The normal PSB PNTPHISG in USER01.PSBLIB

   g) The program PLIPGM5 in USER01.LOADLIB

   h) The VSAM data set USER01.PNTOVFLW (HISAM OVFLW)

   i) The VSAM data set USER01.PNTDBHIS (DD1 Data set)

   j) The PDS USER01.PROCLIB with the member DLIBATCH as below. Note that the original version in IMS.PROCLIB has been tailored to the version below:-

```
//        PROC MBR=TEMPNAME,PSB=,BUF=7,
//             SPIE=0,TEST=0,EXCPVR=0,RST=0,PRLD=,
//             SRCH=0,CKPTID=,MON=N,LOGA=0,FMTO=T,
//             IMSID=,SWAP=,DBRC=,IRLM=,IRLMNM=,
//              BKO=N,IOB=,SSM=,APARM=,
//             RGN=2048K,
//*            SOUT=A,LOGT=2400,SYS2=,
//             SOUT=A,SYS2=,
//             LOCKMAX=,GSGNAME=,TMINAME=
//G       EXEC PGM=DFSRRC00,REGION=&RGN,
//             PARM=(DLI,&MBR,&PSB,&BUF,
//             &SPIE&TEST&EXCPVR&RST,&PRLD,
//             &SRCH,&CKPTID,&MON,&LOGA,&FMTO,
//             &IMSID,&SWAP,&DBRC,&IRLM,&IRLMNM,
//             &BKO,&IOB,&SSM,'&APARM',
//             &LOCKMAX,&GSGNAME,&TMINAME)
//STEPLIB  DD DSN=IMS.&SYS2.RESLIB,DISP=SHR
//         DD DSN=IMS.&SYS2.PGMLIB,DISP=SHR
//         DD DSN=USER01.LOADLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.&SYS2.RESLIB,DISP=SHR
//IMS      DD DSN=USER01.PSBLIB,DISP=SHR
//         DD DSN=USER01.DBDLIB,DISP=SHR
//PROCLIB  DD DSN=IMS.&SYS2.PROCLIB,DISP=SHR
//IEFRDER  DD DSN=USER01.IMSLOG,DISP=(NEW,KEEP),
//         DCB=(RECFM=VB,BLKSIZE=1920,
//         LRECL=1916,BUFNO=2),SPACE=(TRK,(1,1))
//SYSUDUMP DD SYSOUT=&SOUT,
//         DCB=(RECFM=FBA,LRECL=121,BLKSIZE=605),
//         SPACE=(605,(500,500),RLSE,,ROUND)
//IMSMON   DD DUMMY
```

7. The QSAM PDS USER01.LOAD.DATA with load data in the member DATA. The load data is created using the ISPF editor and is given below. Make sure that the data set characteristics match SYSIN. Note also that the segments are in hierarchical sequence. The first eight characters of the data is the segment type.

```
PATIENT   00001ABCDEF1   18,CHN 600023-1
ILLNESS   01012000MALARIA
TREATMNT  01012000QUININE   0004DR.DOBBS
BILLING   000600
PAYMENT   000600
HOUSHLD   MOHAN     FATHER
PATIENT   00002ABCDEF2   18,CHN 600023-2
ILLNESS   01012000JAUNDICE
TREATMNT  01012000AYURVEDIC 0004DR.JAMES
BILLING   000500
PAYMENT   000400
PAYMENT   000100
HOUSHLD   MEERA     MOTHER
PATIENT   00003ABCDEF3   18,CHN 600023-3
ILLNESS   01012000FLU
TREATMNT  01012000CROCIN    0004DR.PILOO
BILLING   000400
PAYMENT   000400
HOUSHLD   JAYA      SISTER
PATIENT   00004ABCDEF4   18,CHN 600023-4
ILLNESS   01012000MEASLES
TREATMNT  01012000NEEMLEAVES0004DR.TOM
BILLING   000300
PAYMENT   000200
PAYMENT   000100
HOUSHLD   MAYA      SISTER
PATIENT   00005ABCDEF5   18,CHN 600023-5
ILLNESS   01012000TYPHOID
TREATMNT  01012000ANTIBIOTIC0004DR.YOUNG
BILLING   000200
PAYMENT   000200
HOUSHLD   LATA      SISTER
```

8. Here is the load JCL

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC DLIBATCH,MBR=PLIPGM5,PSB= PNTPHISL,DBRC=N
//G.SYSPRINT DD SYSOUT=*
//*
//*  THIS IS FOR THE HISAM PATIENT DATABASE
//G.PNTDBHIS DD DSN=USER01.PNTDBHIS,DISP=SHR
//G.PNTOVFLW DD DSN=USER01.PNTOVFLW,DISP=SHR
//*
//* THIS IS THE LOAD DATA FOR PATIENT DATABASE
//* OR YOU CAN PRESENT THE LOAD DATA INLINE
//G.SYSIN DD DSN=USER01.LOAD.DATA(DATA),DISP=SHR
//*
//G.DFSVSAMP DD *
VSRBF=2048,4
/*
```

```
//
```

9. You can test the load by running the program PLIPGM4 which reads all the segments from the data base in hierarchical sequence and outputs it on SYSPRINT. However delete the IEFRDER log file, else you will get a JCL error. The log file created is USER01.IMSLOG. The JCL is given below:-

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC DLIBATCH,MBR=PLIPGM4,PSB= PNTPHISG,DBRC=N
//G.SYSPRINT DD SYSOUT=*
//*
//*  THIS IS FOR THE HISAM PATIENT DATABASE
//G.PNTDBHIS DD DSN=USER01.PNTDBHIS,DISP=SHR
//G.PNTOVFLW DD DSN=USER01.PNTOVFLW,DISP=SHR
//*
//G.DFSVSAMP DD *
VSRBF=2048,4
/*
//
```

10. You must not get any errors from either the LOAD or the DUMP run

11. If you get any load errors etc, simply rerun the job which created the VSAM clusters. It will delete the existing ones and recreate new ones.

A COBOL SAMPLE PROGRAM AND JCL

Index

IGYWCL PROC (modified to suit IMS)

```
//IGYWCL PROC  LNGPRFX='IGY.V2R1M0',SYSLBLK=3200,
//             LIBPRFX='CEE',
//             PGMLIB='&&GOSET',GOPGM=GO
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//RESLIB   DD  DSNAME=IMS.RESLIB,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DSNAME=USER01.PROCLIB(CBLTDLI),DISP=SHR
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&PGMLIB(&GOPGM),
//             SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
```

Compilation JCL for COBOL

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC IGYWCL,REGION=0M
//COBOL.SYSIN  DD DSN=USER01.COBOL.SOURCE(LOADPGM),DISP=SHR
//LKED.SYSLMOD DD DSN=USER01.LOADLIB(LOADPGM),DISP=SHR
//
```

CBLTDLI

```
LIBRARY RESLIB(CBLTDLI)     DL/I LANG INTF
LIBRARY RESLIB(DFHEI01)     HLPI LANG INTF
LIBRARY RESLIB(DFHEI1)      HLPI LANG INTF
ENTRY DLITCBL
```

PNTPHISG

```
        PCB  TYPE=DB,NAME=PNTDBHIS,PROCOPT=A,KEYLEN=21
        SENSEG NAME=PATIENT,PARENT=0
        SENSEG NAME=ILLNESS,PARENT=PATIENT
        SENSEG NAME=TREATMNT,PARENT=ILLNESS
        SENSEG NAME=BILLING,PARENT=PATIENT
        SENSEG NAME=PAYMENT,PARENT=BILLING
        SENSEG NAME=HOUSHLD,PARENT=PATIENT
        PSBGEN PSBNAME=PSB2,LANG=COBOL
        END
```

READPGM

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    READPGM.
      *****************************************************************
      *          E N V I R O N M E N T    D I V I S I O N             *
      *****************************************************************
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
      *****************************************************************
      *                  D A T A     D I V I S I O N                  *
      *****************************************************************
       DATA DIVISION.
       FILE SECTION.
      *****************************************************************
      *                W O R K I N G    S T O R A G E                 *
      *****************************************************************
       WORKING-STORAGE SECTION.
       01  WORK-AREAS.
           05 PARM-COUNT           PIC S9(09) COMP VALUE 3.
           05 FUNCN                PIC X(4)   VALUE 'GN  '.
           05 SEG-IO-AREA          PIC X(45).
           05 SSA                  PIC X(09) VALUE 'PATIENT'.
       01 CONSTANTS.
           05 C-GB                 PIC X(2)  VALUE 'GB'.
       01 SWITCHES.
           05 S-FLAG-BIT            PIC X(01) VALUE LOW-VALUES.
             88 S-FLAG                        VALUE HIGH-VALUES.
       LINKAGE SECTION.
       01 PCBMASK.
           05 DBD-NAME               PIC X(08).
           05 SEG-ID                 PIC X(02).
           05 STATS                  PIC X(02).
           05 PROCOPT                PIC X(04).
           05 FILLER                 PIC X(04).
           05 SEGMENT-NAME           PIC X(08).
           05 LENGTH-FDBK            PIC S9(05) COMP.
           05 NUMBER-SENSEGS         PIC S9(05) COMP.
           05 KEY-FDBK-AREA          PIC X(21).
      *****************************************************************
      *                 PROCEDURE DIVISION                            *
      *****************************************************************
       PROCEDURE DIVISION.
          ENTRY 'DLITCBL' USING PCBMASK
          PERFORM MAIN-PARA
          PERFORM FINAL-PARA.
       MAIN-PARA.
           CALL 'CBLTDLI' USING FUNCN,
                     PCBMASK,
                     SEG-IO-AREA.
           DISPLAY 'SEGMENT NAME '  SEGMENT-NAME
           DISPLAY 'SEGMENT AREA '  SEG-IO-AREA
           PERFORM UNTIL S-FLAG
           MOVE SPACES TO SEG-IO-AREA
           CALL 'CBLTDLI' USING FUNCN,
```

```
                        PCBMASK,
                        SEG-IO-AREA
          IF STATS NOT = C-GB
              DISPLAY 'SEGMENT NAME '  SEGMENT-NAME
              DISPLAY 'SEGMENT AREA '  SEG-IO-AREA
          END-IF
          IF STATS = C-GB
              DISPLAY 'NON BLANK FROM GN CALL ' STATS
              MOVE HIGH-VALUES TO S-FLAG-BIT
          END-IF
          END-PERFORM.
       FINAL-PARA.
            DISPLAY  'END OF PROGRAM'
            GOBACK.
       A100-EXIT.
            EXIT.
```

A program to load the Data Base

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    LOADPGM.
      ****************************************************************
      *          E N V I R O N M E N T    D I V I S I O N            *
      ****************************************************************
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INFILE ASSIGN TO INFILE
           ORGANIZATION IS SEQUENTIAL
           ACCESS IS SEQUENTIAL
           FILE STATUS IS FILE-STAT.
      ****************************************************************
      *                  D A T A     D I V I S I O N                 *
      ****************************************************************
       DATA DIVISION.
      FILE SECTION.
      FD INFILE
          RECORD CONTAINS 80 CHARACTERS.
      01 IN-REC                    PIC X(80).
     ****************************************************************
     *                W O R K I N G    S T O R A G E                 *
     ****************************************************************
      WORKING-STORAGE SECTION.
      01  WORK-AREAS.
          05 PARM-COUNT            PIC S9(09) COMP VALUE 3.
          05 FUNCN                 PIC X(4)   VALUE 'ISRT'.
          05 FILE-STAT             PIC X(2)   VALUE ' '.
      01  IO-BUFFER.
          05 SSA                   PIC X(10).
          05 SEG-IO-AREA           PIC X(70).
      01 SWITCHES.
          05 S-FLAG-BIT             PIC X(01) VALUE LOW-VALUES.
            88 S-FLAG                         VALUE HIGH-VALUES.
      LINKAGE SECTION.
      01 PCBMASK.
          05 DBD-NAME               PIC X(08).
          05 SEG-ID                 PIC X(02).
```

```
           05 STATS                  PIC X(02).
           05 PROCOPT                PIC X(04).
           05 FILLER                 PIC X(04).
           05 SEGMENT-NAME           PIC X(08).
           05 LENGTH-FDBK            PIC S9(05) COMP.
           05 NUMBER-SENSEGS         PIC S9(05) COMP.
           05 KEY-FDBK-AREA          PIC X(21).
      ****************************************************************
      *                  PROCEDURE DIVISION                          *
      ****************************************************************
       PROCEDURE DIVISION.
           ENTRY 'DLITCBL' USING PCBMASK
           PERFORM MAIN-PARA
           PERFORM FINAL-PARA.
       MAIN-PARA.
           OPEN INPUT INFILE
           PERFORM UNTIL S-FLAG
           READ INFILE RECORD INTO IO-BUFFER
                AT END MOVE HIGH-VALUES TO S-FLAG-BIT
           END-READ
           IF NOT S-FLAG
           CALL 'CBLTDLI' USING FUNCN,
                     PCBMASK,
                     SEG-IO-AREA,
                     SSA
           DISPLAY SEG-IO-AREA
           END-IF
           END-PERFORM.
       FINAL-PARA.
            DISPLAY  'END OF PROGRAM'
            GOBACK.
       A100-EXIT.
             EXIT.
```

Retrieving Data from an IMS Data Base

Index

When you complete this section, you will understand how to retrieve data from an IMS data base. The function codes for all DL/I retrieval calls start with G, Standing for Get. The retrieval calls can only retrieve data from a data base. They cannot in any way alter the contents of a data base. The types of retrieval calls are discussed on the following pages.

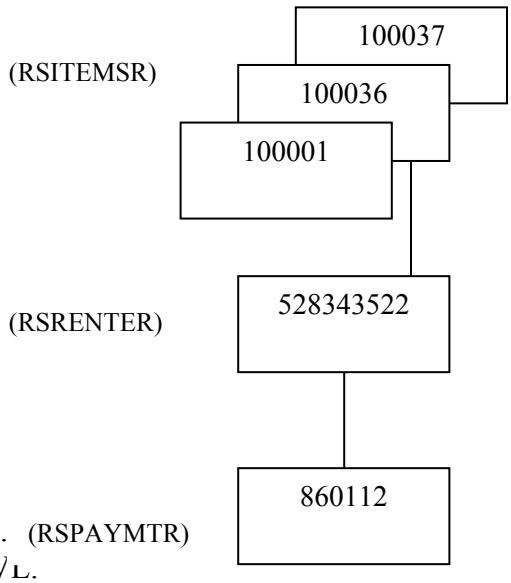
Get Unique (GU)

DL/I provides direct access to not only a specific segment type, but also a specific occurrence of that segment type. Direct access is accomplished through the use of the Get Unique call (function code GU). The GU call causes DL/I to begin its search from the beginning of the data base, regardless of where it is currently positioned in the data base, and retrieve the target segment's data.

Suppose you are looking for payment information on Robert Smith when he rented a color TV on January 12, 1986 from ABC Rental Company. Robert's Social Security number is 528-34-3522. The color TV was item number 100036. It was rented on 01-12-86. Since the information provided is specific, you can use a GU call to accomplish the request. The following example shows how this is done.

PL/I

```
ITEM_FLDVL        - '100036'.
RENTER_FLDVL      - '528343522'.
PAYMENT_FLDVL     - '860112'.

CALL PLITDLI (C_6,
              C_GU,
              CT7DPITM_PTR,
              PAY_SEGMENT_IO,
              RSITEMSR_SSA_Q,
              RSRENTER_SSA_Q,
              RSPAYMTR_SSA_Q);
```



COBOL

```
MOVE  '100036'    TO   ITEM_FLDVL.
MOVE  '528343522' TO RENTER-FLDVL.  (RSPAYMTR)
MOVE  '860112'    TO   PAYMENT-FLDVL.

CALL  'CBLTDLI'   USING C-6,
                        C-GU,
                        L-CT7DPITM,
                        PAY-SEGMENT-IO,
                        RSITEMSR- SSA –Q,
                        RSRENTER-SSA –Q,
```

RSPAYMTR-SSA-Q.

Even though there may be hundreds of items in the rental data base DL/I starts its search from the beginning looking for the first segment, RSITEMSR, with the item number, 100036. Then it searches for the second segment, RSRENTER, with a social security number of 528343522. Finally, DL/I finds the target segment, RSPAYMTR, with a payment date of 860112. All these criteria must be satisfied before DL/I will retrieve the requested data.

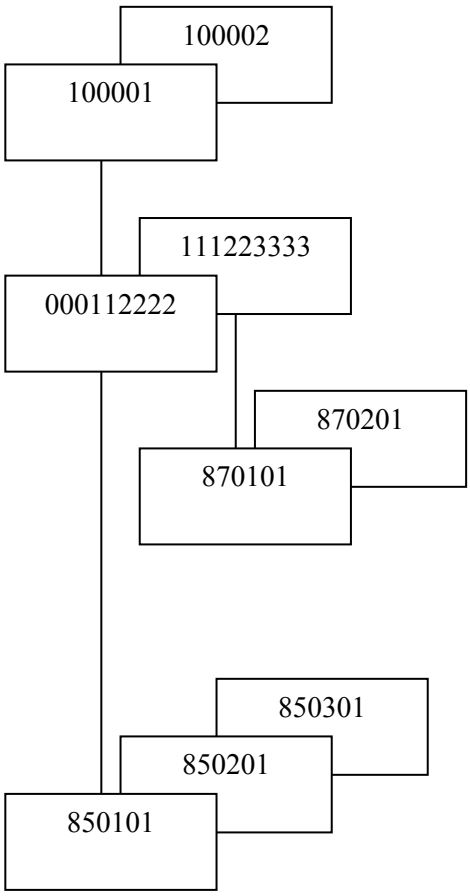
Get Next (GN)

If you do not have specific information or the customer request is general in nature, DL/I provides sequential retrieval of data from a data base via the GN call. The GN call is used to process segments in hierarchical sequence. This allows you to retrieve each segment occurrence in the data base without knowing the key or search field values.

The following is an example using Get Next to process all the segments in the data base.

PL/I

```
CALL PLITDLI   (  C_3,
                  C_GN,
                  DB_PTR,
                  SEG_IO_GENERIC);

DO WHILE (STATUS Ø =’GB’):

CALL PLITDLI (  C_3,
                C_GN,
                DB_PTR,
                SEG_IO_GENERIC);

END: /* DO WHILE */
```



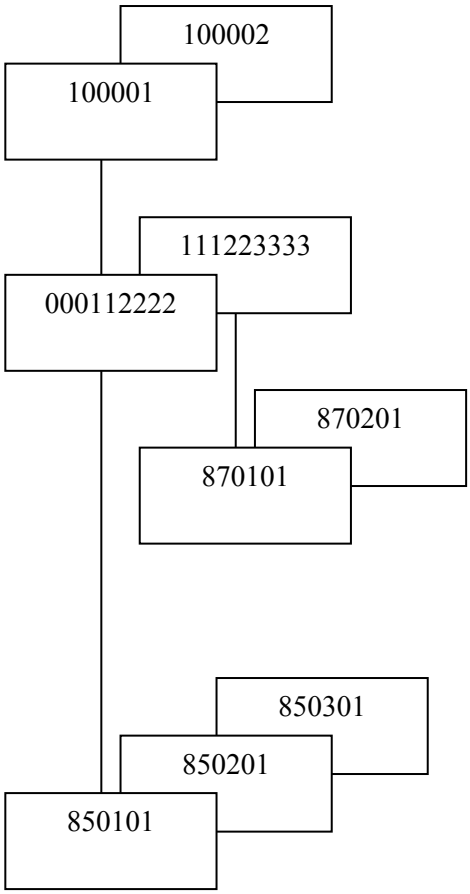
COBOL

```
CALL `CBLTDLI' USING C-GN,
                     L-DB-PCB,
                     SEG_IO_GENERIC.

PERFORM  A100-RENTER UNTIL
         STATUS=’GB’.
GOBACK.

A100-RENTER SECTION.

CALL ‘CBLTDLI’   USING  C-GN,
                        L-DB-PCB,
                        SEG-IO-GENERIC.

A100-EXIT.
EXIT.
```

The following is an example using Get Next to process all the renters in the data base.

PL/I

```
CALL PLITDLI    (   C_5,
                    C_GN,
                    DB_PTR,
                    RENTER_SEG_IO,
                    ITEM_SSA_U,
                    RENTER_SSA_U);

DO WHILE (STATUS Ø =’GB’):

CALL PLITDLI (  C_5,
                C_GN,
                DB_PTR,
                RENTER_SEG_IO,
                ITEM_SSA_U,
                RENTER_SSA_U);

END: /* DO WHILE */
```



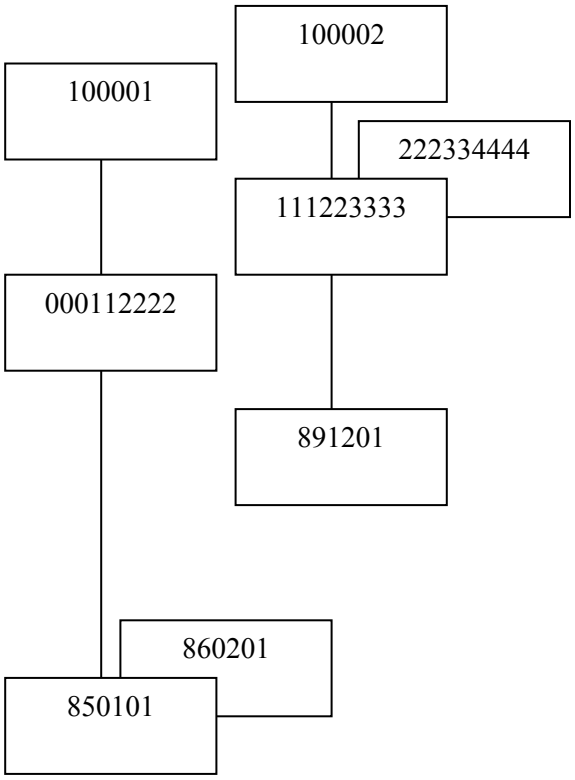
COBOL

```
CALL `CBLTDLI' USING C-GN,
                     L-DB-PCB,
                     RENTER_SEG_IO,
                     ITEM_SSA_U,
                     RENTER_SSA_U.

PERFORM   A100-RENTER UNTIL
          STATUS=’GB’.
GOBACK.

A100-RENTER SECTION.

CALL ‘CBLTDLI’   USING  C-GN,
                        L-DB-PCB,
                        RENTER-SEG-IO,
                        ITEM-SSA-U,
                        RENTER-SSA-U.

A100-EXIT.
EXIT.
```

The following is an example of processing of all the renters under item number,100002.

PL/I

```
ITEM_FLDVALUE = '100002';
CALL   PLITDLI (  C_5,
                  C_GU,
                  DB_PTR,
                  RENTER_SEG_IO,
                  ITEM_SSA_Q,
                  RENTER_SSA_U);
DO WHILE (STATUSØ='GE'):
     •
     •
     CALL PLITDLI (C_5,
                   C_GN,
                   DB_PTR,
                   RENTER_SEG_IO,
                   ITEM_SSA_Q,
                   RENTER_SSA_U);

END: /* DO WHILE */
```



COBOL

```
MOVE  '100002'   TO   ITEM-FLDVALUE.
CALL   'CBLTDLI'   USING  C-GN,
                          L-DB-PCB,
                          RENTER-SEG-IO,
                          ITEM-SSA-Q,
                          RENTER-SSA-U.

PERFORM     A100-RENTER UNTIL
              STATUS='GE'.

GOBACK.

A100-RENTER SECTION.
     •
     •
     CALL 'CBLTDLI'   USING  C-GN,
                             L-DB-PCB,
                             RENTER-SEG-IO,
                             ITEM-SSA-Q,
                             RENTER-SSA-U.
A100-EXIT.
EXIT.
```

In the above example, the first GN call retrieved renter 111223333. The second call retrieved renter 222334444. Finally, the third call raised a GE status code, indicating that there are no more renters under item 100002. A GE status code means not found.

Get Next Within Parent (GNP)

DL/I provides another sequential call, the Get Next Within Parent call (function code (GNP). This type of call retrieves the next occurrence of the target segment type based on the established parent of the segment. This type of call limits DL/I'S retrieval of segments based on the segment's parent. This means that DL/I will only process segments that are directly dependent on the parent established by a previous GU or GN call.

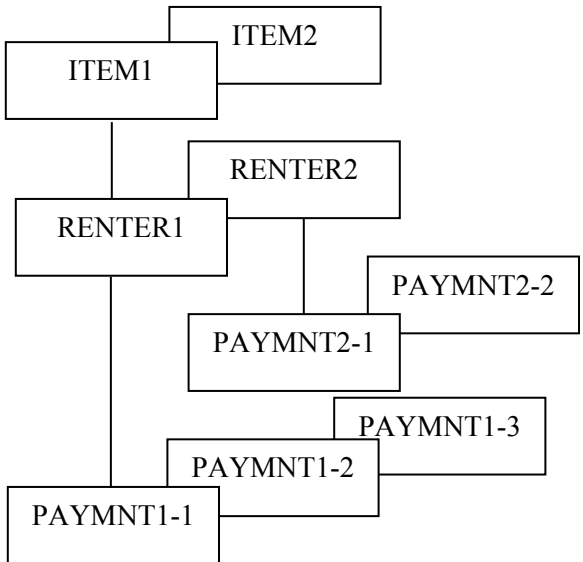
Parentage

An important part of positioning in a data base is parentage. When processing data in lower level segments, ensuring that only segments for a given parent are processed might improve accuracy and efficiency. DL/I provides a facility that allows the application program to establish a parent, so that DL/I will process all of its dependents, but preclude the next occurrence (a twin) of the established parent.

Setting Parentage

Setting parentage means that you are telling DL/I that a specific segment might be Used as a parent. This does not mean you want DL/I to use the segment as a parent, it only indicates your possible intentions. Setting parentage is accomplished by issuing any one of the following DL/I calls, GU, GN, GHU, or GHN. (These calls will be explained later). The target segment in the call has the potential to be a parent from which DL/I will not move position in subsequent calls.

Once parentage is set, you can use it by issuing a GNP or GHNP call. Setting parentage tells DL/I that subsequent GNP calls will only process those dependent segments under the established parent. Once all the dependents have been processed, the next GNP will cause DL/I to return a GE, status code.



Setting Parentage

Let's look at two different situations using the data base in above Figure to see how DL/I responds to the call sequences. First, retrieve all the payment segments under the first renter for item number ITEM1

Example 1

Call Sequence	Stat	Segment	Result
GU ITEM-Q (ITEM1)	bb	ITEM1	PARENTAGE SET
GN ITEM-Q, RENTER -U	bb	RENTER1	PARENTAGE SET
GNP ITEM-U, RENTER –U,PAYMNT-U	bb	PAYMNT-1	PARENTAGE SET-RENTER1
GNP ITEM-U, RENTER –U,PAYMNT-U	bb	PAYMNT-2	PARENTAGE SET-RENTER1
GNP ITEM-U, RENTER –U,PAYMNT-U	bb	PAYMNT-3	PARENTAGE SET-RENTER1
GNP ITEM-U, RENTER –U,PAYMNT-U	GE		PARENTAGE SET-RENTER1

Let's review the sequence of calls above.

1. A qualified GU was issued to retrieve item segment ITEM1. This causes DL/I to set parentage and to retrieve the data for ITEM1.

2. A GN call is issued using a qualified item SSA and an unqualified renter SSA. Since the renter SSA is unqualified, the first occurrence of a renter segment, RENTER1, is retrieved by DL/I for ITEM1. DL/I now sets parentage at RENTER1 because it was the target of the call.

3. A GNP call is issued using a qualified item SSA, an unqualified renter SSA, and an unqualified payment SSA. Since the payment SSA is unqualified, DL/I will retrieve the first occurrence of a payment segment, PAYMNT-1, under the set parent, RENTER1.

4. The next GNP issued is identical to the first. DL/I responds by retrieving payment segment PAYMNT-2.

5. The third GNP produces the same DL/I response. PAYMNTI-3 is retrieved.

6. The last GNP causes DL/I to produce a different result even though the call content is identical to the first three. Instead of retrieving the next payment segment in the data base, DL/I returned a GE, status code. This is because parentage is set at RENTERI, and DL/I will not move from that parent. Since all the other payment segments appear under a different renter segment, DL/I will not process them. This series of DL/I calls produces the same results as in the problem example.

Let's look at another series of calls to solve the sample problem and see how DL/I responds.

Example 2

Call Sequence	Stat	Target Segment	Result
GN ITEM-Q(ITEM1)	bb	ITEM1	PERENTAGE
GNP ITEM-U, RENTER-U, PAYMNT – U	bb	PAYMNT1-1	PARENTAGE SET-ITEM1
GNP ITEM-U, RENTER-U, PAYMNT – U	bb	PAYMNT1-2	PARENTAGE SET-ITEM1
GNP ITEM-U, RENTER-U, PAYMNT – U	bb	PAYMNT1-3	PARENTAGE SET-ITEM1
GNP ITEM-U, RENTER-U, PAYMNT – U	bb	PAYMNT2-1	PARENTAGE SET-ITEM1
GNP ITEM-U, RENTER-U, PAYMNT - U	bb	PAYMNT2-2	PARENTAGE SET-ITEM1
GNP ITEM-U, RENTER-U, PAYMNT - U	GE		PARENTAGE SET-ITEM1

DL/I responds to the sequence of calls above in the following way.

1. A qualified GN was issued to retrieve item segment ITEM1. This causes DL/I to set parentage at ITEM1 and the data to be retrieved for ITEM1.

2. A GNP call is issued using a qualified item SSA, an unqualified renter SSA, and an unqualified payment SSA. Since both the renter and payment SSA'S are unqualified, DL/I will process RENTER1 because it is the first occurrence of the renter segments. DL/I will then retrieve the target segment, PAYMNT1-1, because it is the first occurrence of the payment segments.

3. The next GNP issued is identical to the first. DL/I responds by retrieving payment segment, PAYMNT1-2.

4. The third GNP produces the same DL/I response and PAYMNT1-3 is retrieved.

5. The fourth GNP causes DL/I to retrieve payment segment, PATMNT2-1, located under RENTER2, even though the call content is identical to the first three. DL/I still returns a blank status code even though it had to move up to RENTER2 first and then down to PAYMNT2-1 to satisfy the call. Remember, the parent is set at ITEM1, not RENTER1.

6. DL/I responds to the fifth GNP the same way it did for the first three GNPs, and returns PAYMNT2-2.

7. The last GNP produces a GE status code from DL/I. This is because there are no more payment segments under the set parent, ITEM1.

This series of calls did not solve the sample problem because you retrieved more segments than you needed.

You have seen two different approaches to solve a problem using the parentage facility in DL/I. Each produced a different result because the parentage set by DL/I was different in each case. It is very important to understand not only where DL/I is currently positioned in the data base, but also where the established or set parent exists.

The following example shows how the various DL/I retrieval functions can be combined to solve a problem. Suppose a customer would like to see the payment history for all renters for item number 100021. The following code can be used to satisfy the customer's request.

PL/I

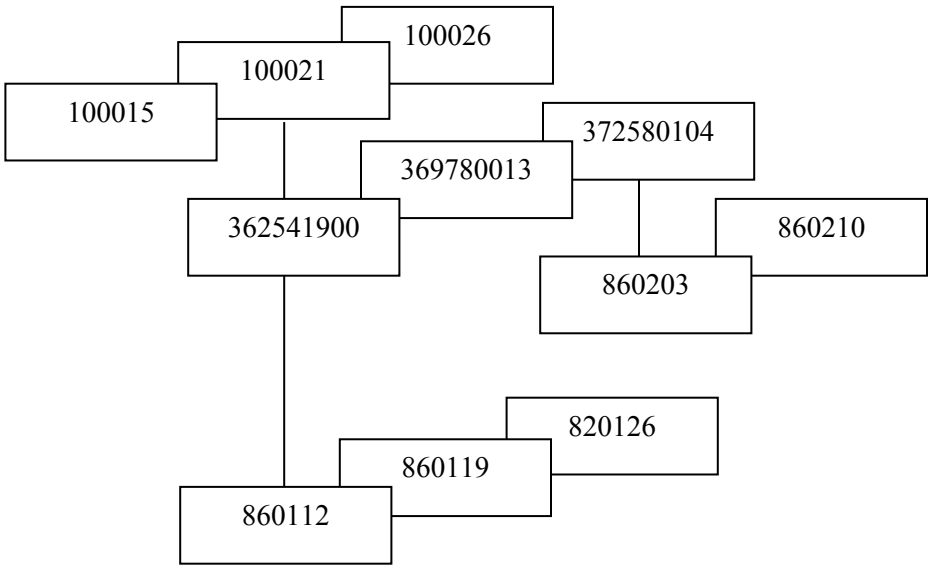
```
ITEM_FLDVL='100021';
/* ITEM_SSA IS QUALIFIED, RENTER_SSA IS UNQUALIFIED.
CALL PLITDLI   (C_5,
                C_GU_FUNC,
                CT7DPITM_PTR,
                RENTER_SEGMENT_IO,
                ITEM_SSA_Q,
                RENTER_SSA_U);

DO WHILE (STATUS_CODE =' ');

    CALL PLITDLI   (C_6,
                    C_GNP_FUNC,
                    CT7DPITM_PTR,
                    PAY_SEGMENT_IO,
                    ITEM_SSA_U,
                    RENTER_SSA_U,
                    PAYMNT_SSA_U);

    DO WHILE (STATUS_CODE = ' ');

    CALL PLITDLI   (C_6,
                    C_GNP_FUNC,
                    CT7DPITM_PTR,
                    PAY_SEGMENT_IO,
                    ITEM_SSA_U,
                    RENTER_SSA_U,
                    PAYMNT_SSA_U);
     END;
     /* GN call issued instead of GU to get next renter. */
     /* ITEM_SSA is qualified. RENTER_SSA is unqualified. */
     CALL PLITDLI  (C_5,
                    C_GN_FUNC,
                    CT7DPITM_PTR,
                    RENTER_SEGMENT_IO,
                    ITEM_SSA_Q,
                    RENTER_SSA_U);
END:
```

COBOL

```
        MOVE   '100021'   TO   ITEM-FLDVL.
*       ITEM_SSA is qualified. RENTER_SSA is unqualified.
        CALL 'CBLTDLI'   USING      C-5,
                                    C-GU-FUNC,
                                    L-CT7DPITM,
                                    RENTER-SEGMENT-IO
                                    ITEM-SSA-Q,
                                    RENTER-SSA-U.

        PERFORM   A100-RENTER
                        UNTIL STATUS-CODE   NOT=' '.
        GOBACK.

        A100-RENTER SECTION.
                  •
        CALL 'CBLTDLI'   USING      C-6,
                                    C-GNP-FUNC,
                                    L-CT7DPITM,
                                    PAY-SEGMENT-IO,
                                    ITEM-SSA-U,
                                    RENTER-SSA-U,
                                    PAYMENT-SSA-U.

        PERFORM   A200-PAYMENT
                              UNTIL  STATUS-CODE NOT =' '.
*       GN call issued instead of GU to get next renter.
*       ITEM-SSA is qualified. RENTER-SSA is unqualified.
        CALL 'CBLTDLI'   USING      C-5,
                                    C-GN-FUNC,
                                    L-CT7DPITM,
                                    RENTER-SEGMENT-IO,
                                    ITEM-SSA-Q,
                                    RENTER-SSA-U.

      A100-EXIT.
            EXIT.

      A200-PAYMENT SECTION.
                  •
                  •
        CALL 'CBLTDLI'   USING  C-6,
                                C-GNP-FUNC,
                                L-CT7DPITM,
                                PAY-SEGMENT-IO,
                                ITEM-SSA-U,
                                RENTER-SSA-U,
```

```
                                        PAYMENT-SSA-U.

A200-EXIT.
     EXIT.
```

Let's examine the skeletal logic used in the previous coding examples using the data base shown earlier and reproduced again here.



Data Base Example

1. The item segment SSA was qualified with a key value of 100021. The SSA'S for both the renter and payment segments were set up to be unqualified.

2. A GU call is issued to retrieve the item segment with an item number equal to 100021. Since the renter SSA is unqualified, DL/I retrieves the first occurrence of the renter segment key, =362541900.

3. Based on the success of the GU call, a loop is set up to process all the payment segments for the renter segment just retrieved. A GNP call is issued to retrieve a payment segment. Since the payment SSA is unqualified, DL/I retrieves the first occurrence of the payment segment, key=860112.

4. The process continues until all the payment segments have been processed for all the renters of item number 100021.

The following table shows all the calls for the sample data base shown in the Figure above and how DL/I responded to each call.

Call Sequence	Segment Returned	Status Code
GU renter	362541900	blank
GNP payment	860112	blank
GNP payment	860119	blank
GNP payment	860126	blank
GNP payment		GE
GN renter	369780013	blank
GNP payment		GE
GN renter	372580104	blank
GNP payment	860203	blank
GNP payment	860210	blank
GNP payment		GE
GN renter		GE

Get Hold Calls

DL/I provides a set of special function retrieval calls, they are Get Hold Calls(GHU). For every retrieval call (for example, GU, GN, and GNP) there is a corresponding hold call (GHU, GHN and GHNP, respectively). The GHU calls cause IMS to enqueue the retrieved segment so that no other application in the system can access it. Your application program has exclusive use of that segment.

For example, if you issue a GHU for segment RSRENTER in data base CT7DPITM, IMS will enqueue that segment occurrence. Now, you have exclusive access to that segment. If you issue a GN call for segment RSPAYMTR in data base CT7DPITM, IMS will free the enqueued (GHU, GHN, GHNP) segment RSRENTER, enabling all other applications to access the segment again.

Common Status Codes

The following table shows some common status codes returned from the various retrieval calls.

Status	Description
(blank)	Call was successful.
GA	DL/I returned a segment at a level higher in the hierarchy.
GB	Logical end of data base.
GE	Segment not found.
GK	DL/I returned a different segment type at the same level.
GP	Parentage never established prior to a GNP call.

Positioning in a Data Base

To this point, standard DL/I calls have been discussed. You have provided DL/I with all of the necessary SSA'S and properly qualified them. What if you forgot to specify an SSA or improperly qualified one or more SSA'S in a call? In that case, DL/I makes assumptions to try to complete the call. Most of these assumptions are based on its current position in the data base.

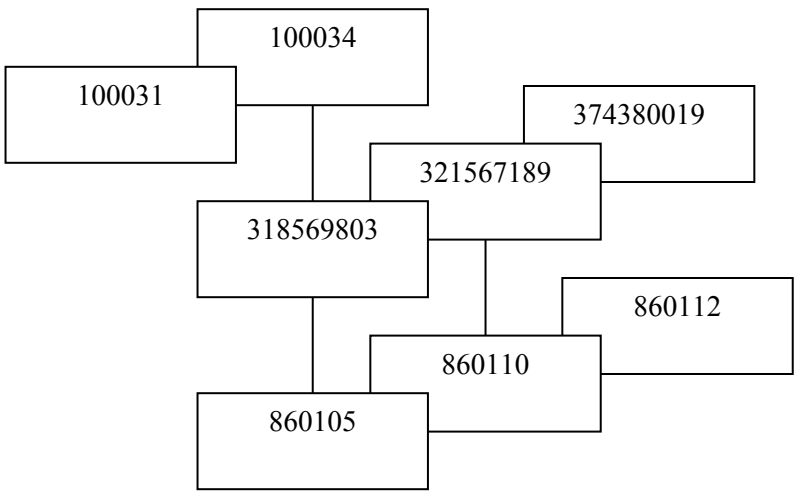
When a data base call is issued to DL/I, the path to the target segment is maintained by DL/I and is referred to as its current position. All subsequent calls made to the data base are based on DL/I'S current position. Many factors go into the formula that DL/I uses to establish its current position in the data base. They include the following:

- Success or failure of the call
- Type of call issued
- Qualified or Unqualified call
- Qualified or Unqualified SSA'S
- Missing SSA'S

Let's examine a typical DL/I call to see how DL/I determines its current position. Figure below shows an example of a DL/I call.

ITEM FIELD VALUE = '100034'
RENTER FIELD VALUE = '321567189'
PAYMENT FIELD VALUE = '860112'

GU CT7DPITM, PAYSEGIO, ITEMSSA_Q, RENTSSA_Q, PAYMTSSA_Q



Example of DL/I Positioning after Successful Call

Let's follow what took place in the call example above.

- Assume that all SSA'S are qualified. First assign values to the keys of the segments to be retrieved.
- Next, issue the DL/I call. In this case, the call is a GU with all three SSA'S specified.
- DL/I searches the data base for an item segment with a key value of 100034. Once the correct item segment is found, DL/I establishes its position at that level. DL/I establishes this position by saving the key value in the PCB's concatenated key area (the key feedback area) of the PCB mask. The concatenated key has the following value:

1	0	0	0	3	4	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- DL/I then proceeds to the renter segment look for a social security number of 321567189. Once this segment occurrence is found, DL/I also establishes its position at this renter segment by adding the key value to the concatenated key area of the PCB mask. This area now contains the following information:

1	0	0	0	3	4	3	2	1	5	6	7	1	8	9	?	?	?	?	?	?

- DL/I now retrieves the target segment, establishes its current position at the payment segment with the date 860112, and updates the concatenated key with the following values:

1	0	0	0	3	4	3	2	1	5	6	7	1	8	9	8	6	0	1	1	2

Finally, DL/I fills in the rest of the PCB mask and returns control to your application program.

The PCB mask was discussed extensively with regard to your example. The following figure shows the contents of the PCB mask after the call.

DBD NAME	C	T	7	D	P	I	T	M
SEGMENT LEVEL	0	3						
STATUS CODE	b	b						
PROC OPTIONS	A	P	b	b				
RESERVED	?	?	?	?				
SEGMENT NAME	R	S	P	A	Y	M	T	R
CONCAT KEYLTH	0	0	1	5				

NUM OF SENSEGS	0	0	0	5															
CONCAT KEY	1	0	0	0	3	4	3	2	5	6	7	1	8	9	8	6	0	1	1	2

Position After Successful DL/I Calls

After Retrievals

When you complete any successful retrieval call, current position is established after the target segment. Remember, the target segment is always the segment specified in the last SSA of your DL/I call.

After Inserts

Immediately after an insert call is issued, position is established just after the segment inserted.

After Deletes

Current position is not affected by a delete call. Position is established with the previous Get Hold call, after the segment retrieved. When the delete is completed, the position remains after the segment deleted, whether the segment had dependents.

After Replaces

Like the delete call, the replace call does not affect the current position in the data base. The Get Hold call prior to the REPL establishes current position after the segment being replaced.

Position When Missing SSA'S are Used in DL/I Calls

When you issue a DL/I call, you usually specify an SSA for each segment type down the path to the target segment. The REPL and DLET calls are exceptions to this rule because they normally do not require SSA'S. If your DL/I call excludes an SSA for any segment down the path to the target segment, you are forcing DL/I to make certain assumptions based on its current position in the data base to satisfy your request. DL/I generally applies the following rules for missing SSA'S:-

- If no position has been previously established for the segment type in the missing SSA, DL/I will substitute an unqualified SSA.

- If position has been established previously on the segment type for which the SSA is missing, DL/I uses the current segment position and may or may not substitute a qualified SSA, depending on the type of call being executed. The IMS Application Programmer's Guide recommends supplying SSA'S at all levels in a DL/I call.

You can determine the current position after a DL/I call based on the contents of the PCB mask.

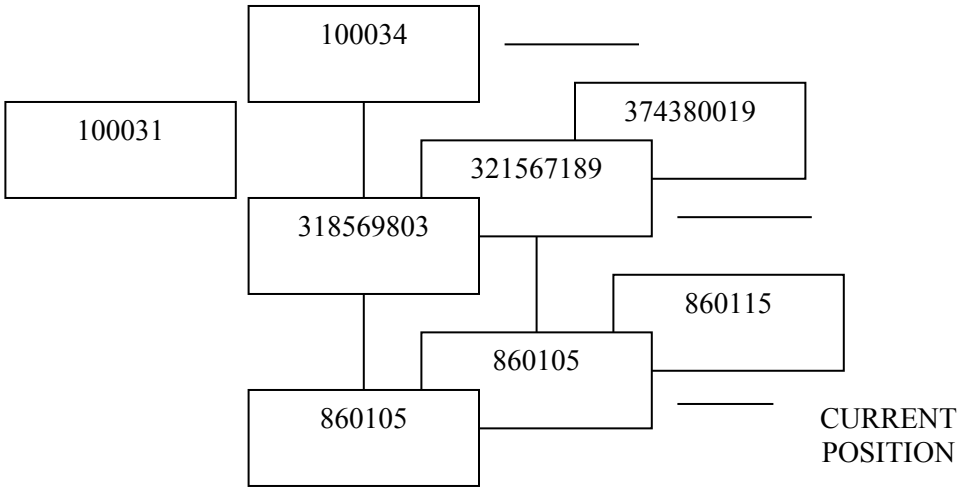
At the beginning of your program, before any DL/I call is made against the data base, the current position is just before the first root segment in the data base.

Position after Unsuccessful DL/I Calls

Dl/I calls often return a non-blank status code, indicating that the call was not successful, or a warning was issued. The extent of the unsuccessful call that DL/I could successfully process determines its current position in the data base. Figure below is an example of an unsuccessful DL/I call, where current position is established.

ITEM FIELD VALUE = '100034'
RENTER FIELD VALUE = '321567189'
PAYMENT FIELD VALUE = '860112'

GU CT7DPITM, PAYSEG10, ITEMSSA_Q, RENTSSA_Q, PAYMTSSA_Q



Example of DL/I positioning after Unsuccessful Call

Let's follow what occurred in the call example in Figure above to see how DL/I determines current position:

- Assume that all SSA'S are qualified. Assign values to the keys of the segments to be retrieved.

- Next, issue the DL/I call. In this case, a GU call is issued in which all three SSA'S are specified.

- DL/I searches the data base looking for an item segment with a key value of 100034. once the segment is found, DL/I establishes its position at that level, as indicated by the bullet. DL/I establishes position by saving the key value in the PCB's concatenated key area of the PCB mask. The concatenated key has the following value:

1	0	0	0	3	4	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- DL/I then proceeds to the renter segment, looking for a social security number of 321567189. Once that segment occurrence is found, DL/I establishes its position at that renter segment also by adding the key value to the concatenated key area of the PCB mask. This area now contains the following information:

1	0	0	0	3	4	3	2	1	5	6	7	8	9	?	?	?	?	?	?	?

- Finally, DL/I attempts to retrieve the target segment, but it cannot find a payment segment with the date, 860112. At this point, DL/I sets the status code in the PCB mask and returns control to your application program.

Examine the contents of the PCB mask in the following example after the unsuccessful call was issued.

DBD NAME	C	T	7	D	P	I	T	M												
SEGMENT LEVEL	0	2																		
STATUS CODE	G	E																		
PROC OPTIONS	A	P	b	b																
RESERVED	?	?	?	?																
SEGMENT NAME	R	S	R	E	N	T	E	R												
CONCAT KEYLTH	0	0	0	F																
NUM OF SENSEGS	0	0	0	5																
CONCAT KEY	1	0	0	0	3	4	3	2	5	6	7	1	8	9	?	?	?	?	?	?

DL/I updated the PCB mask using the information from the last segment it successfully processed. DL/I uses this information to maintain its position in the data base. After this call is completed, the current position will be after the last segment that DL/I examined before determining that the target segment did not exist. In this example, it is after the second payment segment, 860105. If an unqualified GN was issued, the second payment with a key of 860115 would be retrieved.

Notice that the status code in the PCB mask is GE. DL/I is telling you that it could not find the target segment in your request. This does not mean that none of your request was satisfied. DL/I always tries to satisfy the entire request. When it cannot do so, DL/I will satisfy as much of the request as possible. If success is not obtained at the root level, the segment level in the PCB mask will contain a 00. In this case, the segment level was 02,

because it found the item, and the renter that was specified in the corresponding SSA'S. However, you cannot determine from the information in the PCB mask Exactly what the current position is.

Changing Data in an IMS Data Base

Index

Once a data base has been loaded, it rarely remains static. Data bases require maintenance on a regular basis. This includes adding new segments, changing the data in existing segments, or possibly deleting an entire segment occurrence.

DL/I has several calls that provide for the maintenance of data within the data base.

Insert

The insert call, ISRT, Was discussed earlier in connection with initially loading a; database. In this case, the PSB contains the processing options L or LS. The ISRT call structure is identical, whether you are loading a data base or adding new segment occurrences. The main difference is the processing options identified in the PSB. When L or LS is specified, you are in the load mode. If one of the processing options specified in the PSB is I (for insert), or A (for all), you are in the insert mode. As a rule, you only load a data base once. Thereafter, new segments will be added to the data base through the insert mode.

The following are rules for inserting segments in the insert mode.

- The PSB must specify the processing option I or A for the segment (s) you wish to insert.
- You must specify the SSA for the segment type you are inserting. It must be unqualified.
- The place where the segment is inserted depends on the insert rules established in the DBD.

  - If segment is Keyed unique, insertion is in Key sequence.

  - If segment is keyed nonunique, the following rules apply

  The FIRST, LAST or HERE is driven by a command code or the DBD RULES for the SEGM macro.

  First –If the segment does not have a key field, it is inserted as the first occurrence of the segment type.

  If it has a key, and the key is not unique, the segment is inserted as the first occurrence of the segment type with the same key value. If no other segment exists with that key value, the segment is inserted in key sequence.

  Last – If not Keyed, the segment is inserted as the last occurrence of the segment type.

If non unique keyed the segment is inserted as the last occurrence of the segment type with the same key value. If no other segment exists with that key value, the segment is inserted in key sequence.

Here – if non keyed, the segment is inserted before the current position of that segment type. If no position has been established within the segment type. DL/I inserts according to the first rules. This means that it will be inserted as the first occurrence of the segment type.

If nonunique keyed, the segment is inserted before the current position of the occurrence of the segment type with the same key value. If no position has been established within the segment type, DL/I inserts according to the first rules. This means it will be inserted as the first occurrence of the segment type with the same key value. If no other segment exists with that key value. The segment is inserted in key sequence.

Replace

During the life of a data base, segment data is usually changed on a regular basis. The DL/I replace call (REPL), was designed to perform this maintenance chore.

The rules for issuing a REPL call are as follows:

1. The PSB must specify the processing option R or A for segment or segments you wish to replace.

2. Prior to issuing a REPL Call, you must issue a successful GH call such as GHU, GHN or GHNP.

3. You should not issue any other call to the PCB between the time you issue the GH Call and the REPL call. If you do, you must reissue the GH call before attempting to issue the REPL call.

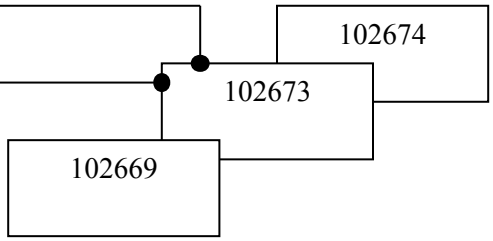
4. You can change any of the data in the I/O area retrieved by the GH call with the following restrictions.
   - You cannot change the key field.
   - You cannot change the lengths of any search fields.

5. Typically, no SSA'S are specified in the REPL call. However, if the SSA'S are specified, they must be unqualified.

The following example shows how a REPL call woks.

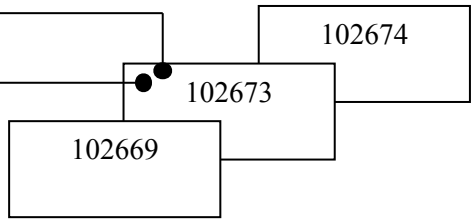
PL/1

```
ITEM_FLDVL  = '102673';
CALL PLITDLI    (C_4,
                 C_GHU,
                 DB_PTR.,
                 ITEM_SEG_IO,
                 ITEM_SSA_Q);

IF STATUS_CODE  =  '  ' THEN
DO:
/* MAKE CHANGES TO SEG-10 */
 CALL PLITDLI    (C_3,
                 C_REPL,
                 DB_PTR,
                 ITEM_SEG_IO);
IF STATUS_CODE=  '  ' THEN
        /* O.K.  */
ELSE
    DO;
        PUT  SKIP  LIST (DB_PCB_MASK);
        PUT  SKIP  LIST ('BAD STATUS CODE');
        CALL  ABND_ROUTINE;
  END;

END:
```



COBOL

```
MOVE  '102673' TO ITEM-FLDVL.
CALL  'CBLTDLI' USING  C-GHU,
                       L-DB-PCB,
                       ITEM-SEG-IO,
                       ITEM-SSA-Q.

IF STATUS-CODE = '  '
 (MAKE CHANGES TO SEG-IO)
CALL 'CBLTDLI' USING  C-REPL,
                       L-DB-PCB,
                       ITEM-SEG-IO

IF STATUS-CODE  =  '    '
   NEXT SENTENCE
ELSE
        DISPLAY    DB-PCB-MASK
        DISPLAY    'BAD STATUS CODE'
        PERFORM  ABND-ROUTINE.
```



In the preceding example, a GHU call was issued to retrieve the exact segment that required modification, item number 102673. The data was changed in the I/O area filled in by the

GHU call. Once the changes were made, the segment was replaced using the REPL call. Notice that no SSA'S are specified. This is because DL/I has all of the necessary information about the segment from the previous GH call.

Common REPL Status codes

Status	Description
(blank)	Call was successful.
DA	Key Field changed.
DJ	No Get Hold call issued.
VI	Invalid length on variable length segment.

Delete

On occasion, it is necessary to remove or delete an occurrence of a segment. DL/I provides this capability through the delete call (DLET).

The rules for issuing a DLET call are as follows:

1. The PSB must specify the processing option of D or A for not only the segment you want to delete, but also for all dependents of that segment. When the DLET call is issued, the segment specified is deleted along with all of its dependents. This rule applies whether or not the PSB is sensitive to the dependent segments.

2. prior to issuing a DLET call, you must issue a successful GH call, such as GHU, GHN or GHNP.

3. You should not issue any other call to the PCB between the time you issue the GH call and the DLET call. If you do, you must reissue the GH call before attempting to issue the DLET Call.

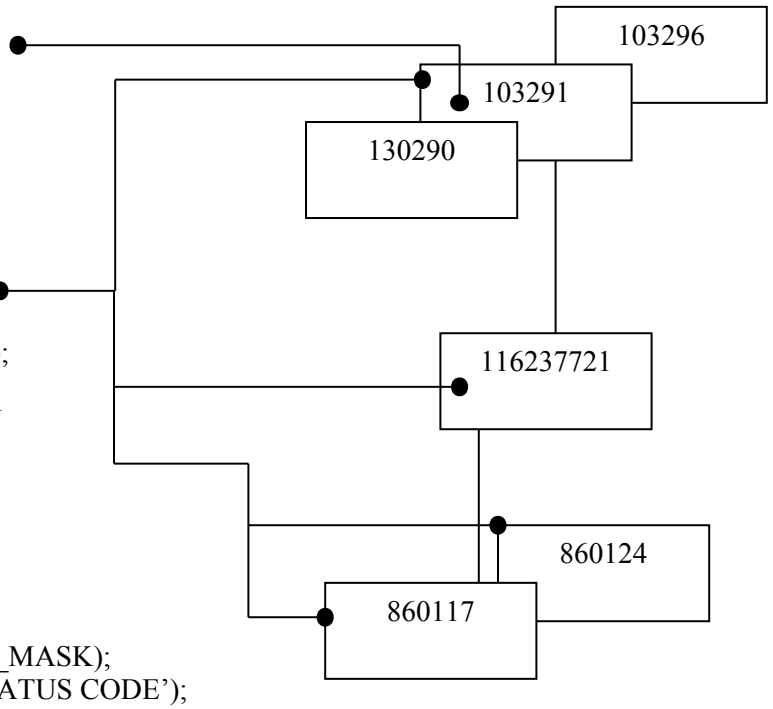
4. you cannot change the key field.

5. As a rule, no SSA'S are specified in the DLET call.

The following example shows how a DLET call works.

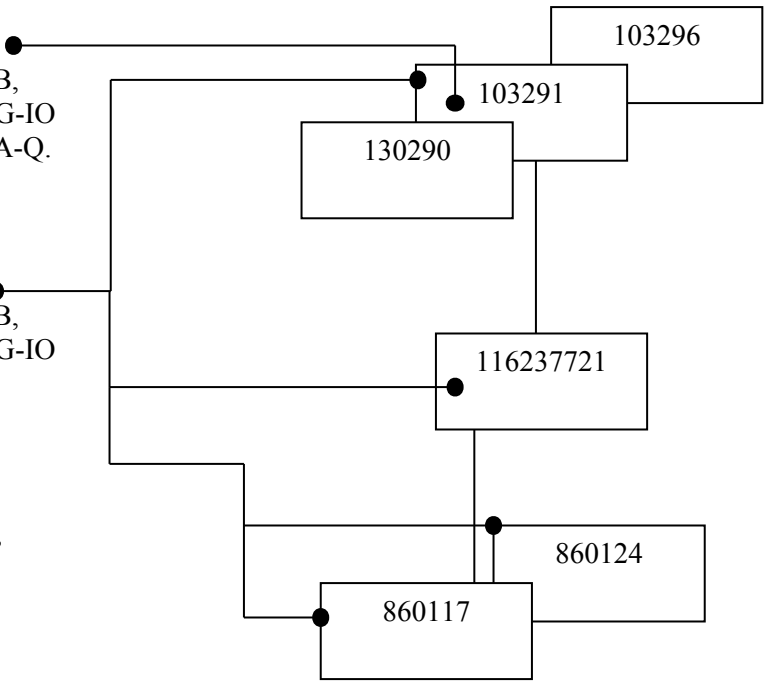
PL/I

```
ITEM_FLDVL  =  '103291'
CALL PLITDLI    (C_4,
                 C_GHU,
                 DB_PTR,
                 ITEM_SEG_IO,
                 ITEM_SSA_Q);
IF STATUS_CODE  =  '  ' THEN
DO;
 CALL PLITDLI    (C_3.
                 C_DLET,
                 DB_PTR,
                 ITEM_SEG_IO);

IF STATUS_CODE  = '      '  THEN
  DO;
   /* O.K.  */

  END;
ELSE
    DO;
       PUT   SKIP  LIST (DB_PCB_MASK);
       PUT   SKIP  LIST ('BAD STATUS CODE');
       CALL     ABND_ROUTINE;
    END;
END;
```



COBOL

```
MOVE  '103291'  TO  ITEM-FLDVL.
CALL 'CBLTDLI'  USING  C-GHU,
                       L-DB-PCB,
                       ITEM-SEG-IO
                       ITEM-SSA-Q.


IF STATUS-CODE  = '  '
CALL ' CBLTDLI' USING C-DLET,
                       L-DB-PCB,
                       ITEM-SEG-IO

IF STATUSC-CODE  = '      '
    NEXT SENTENCE
ELSE
    DISPLAY  L-DB-PCB-MASK
    DISPLAY  'BAD  STATUS CODE'
    PERFORM  ABND-ROUTINE.
```



In the previous example, a GHU call was issued to retrieve the segment to be deleted, item number 103291. The DLET call deletes the segment retrieved through the GHU and its dependents, 116237721, 011786, and 012486. Notice that as in the REPL call, no SSA'S are specified for DLET call, This is because DL/I has all of the necessary information about the segment from the previous Get Hold call.

Common DLET Status Codes

Status	Description
(blank)	Call was successful.
DA	Key Field changed.
DJ	No Get Hold call issued.
DX	Delete rule in DBD was violated.

Advanced DL/I call concepts

The following topics will enhance your ability to use DL/I to process IMS data bases.

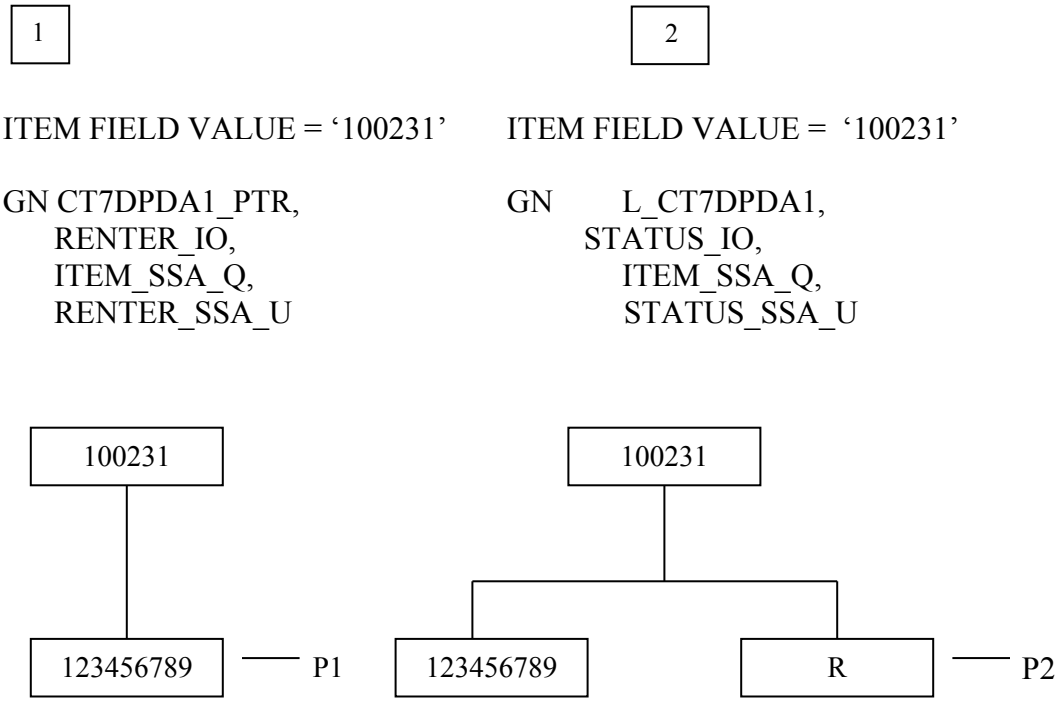
Multiple positioning

DL/I maintains its position down the path to the target segment. There are times when an application program requires the ability to process segment data down two or more hierarchical paths at a time. DL/I provides two methods that allow an application program to simultaneously maintain position down more than one hierarchical path.

POS =M in a PCB

The more unusual and least recommended method is through the POS= keyword in the PCB macro of a PSB. The POS= keyword stands for positioning. The default is POS=S, meaning the position is single. This tells DL/I to maintain its position down the PCB in only a single path. Our previous positioning examples used this default, The following example shows a PSB using multiple positioning.

```
PCB       TYPE=DB, NAME=CT7DPDA1, PROCOPT=A,KEYLEN=21,POS=M
SENSEG NAME=RSITEMSR,PARENT=0
SENSEG NAME=RSRENTER,PARENT=RSITEMSR
SENSEG NAME=RSPAYMTR,  PARENT=RSRENTER
SENSEG NAME=RSSTATSR,PARENT=RSITEMSR
SENSEG NAME=RSFINANR,PARENT=RSITEMSR
PSBGEN PSBNAME=CT7BCDL1,LANG=COBOL
END
```



Example of multiple positioning using POS =M

Figure above shows two DL/I calls and how DL/I maintains its position using the POS=M option.

Examine what DL/I does if POS=S, or if POS=M in the two calls shown in the above figure.

POS=S :- The positioning is single. In call 1, DL/I establishes position after the call in the renter segment with value 123456789, as indicated by P1.

When call two is issued, DL/I removes any established positions for segments at the same level. It also removes all positions of lower level segments. Call two reestablishes position at the status segment. The status, R, indicated by P2.

POS=M:- with multiple positioning, Call one causes DL/I to establish position at renter segment, 123456789, as shown with P1.

When call two is issued, DL/I maintains all positions at the same level and removes only lower level positions. The result of call two is that DL/I keeps the position established at P1 while maintaining an additional position, established at the status segment indicated by P2.

With Multiple PCBs

Using two or more PCBs in the same data base is recommended, and widely practiced. This method requires that you be creative when defining the PCB pointers in your application program. Figure below shows a PSB using multiple PCBs in the same data base and an example of how the pointers might be specified in an application program.

```
PCB       TYPE=DB,NAME=CT7DPDA1,PROCOPT=...
SENSEG NAME=
SENSEG NAME=
•
•
PCB       TYPE=DB,NAME=CT7DPDA1,PROCOPT=...
SENSEG NAME=...
SENSEG NAME=...
•
•
PSBGEN PSBNAME=...
END
```

PL/I

```
DLITPLI :  PROC (CT7DP_PTR1,  CT7DP_PTR2)...
•
•
```

COBOL

```
•
•
ENTRY  'DLITCBL'  USING L – CT7DPITL, L –CT7DPIT2.
•
•
```

Example of Multiple positioning Using Multiple PCBs

When your program uses multiple PCBs for multiple positioning, DL/I assumes that it is using multiple data bases. In reality, there is only one. DL/I maintains position by the PCB. It is not significant to DL/I if two or more PCBs access the same data base.

Before issuing each call, you must ensure the following.

- Each DL/I call points to the correct PCB.
- You have determined the current position in each PCB.

When you simultaneously compare segment data between two or more segments that are located in different hierarchical paths, multiple positioning is a valuable resource.

Advanced SSA concepts

The basic format of an SSA was discussed in Segment Search Argument. But, DL/I provides for more flexible variations of the SSA through the use of Boolean operators and command codes.

Boolean operators

Only one key or search field has been identified in our qualified SSA'S. Also, only equal signs were used as relational operators. Through the use of the Boolean operators, *AND* and OR, you will enable DL/I to search on multiple fields, These searches include the same field of different fields. It is also possible to use comparative qualifiers in addition to **equal to**. These would include qualifiers such as **greater than** or **less than**. The following tables show the various symbols used, along with their respective meanings.

*	Logical "AND"
&	Logical "AND"
+	Logical "OR"
|	Logical "OR"

b= =b EQ	Equal To
b> >b GT	Greater Than
b< >b LT	Less Than
>= => GE	Greater Than or Equal To
<= =< LE	Less Than or Equal To
Ø= =Ø NE	Not Equal To

The SSA format discussed does not change with the expanded set of relational operators. However, it does change using Boolean operators. Figure below shows how an SSA might look using of Boolean operators.

PL/I

```
DCL  01    BOOLEAN_SSA_EXAMPLE,
       05  SEGMENT_NAME                CHAR (08)        INIT (‘RSITEMSR),
       05  QUALIFIER                   CHAR (01)        INIT (‘(‘),
       05  FIELD_NAME1                 CHAR (08)        INIT (‘ITEMNUMB),
       05  RELATIONAL_OPERATOR1        CHAR (02)        INIT (‘>’),
       05  FIELD_VALUE1                CHAR (06)        INIT (‘  ’),
       05  BOOLEAN_OPERATOR1           CHAR (01)        INIT (‘|’),
       05  FIELD_NAME2                 CHAR (08)        INIT (‘COLOR’),
       05  RELATIONAL_OPERATOR2        CHAR (02)        INIT (‘= ‘),
       05  FIELD_VALUE2                CHAR (08)        INIT (‘  ‘),
       05  BOOLEAN_OPERATOR2           CHAR (01)        INIT (‘& ’),
       05  FIELD_NAME3                 CHAR (08)        INIT (‘MANFNAME’),
       05  RELATIONAL_OPERATOR3        CHAR (02)        INIT (‘  =’),
       05  FIELD _VALUE3               CHAR (15)        INIT (‘  ‘),
                    •
                    •
       05  RIGHT_PAREN                 CHAR (01)        INIT (‘)’);
```

COBOL

```
01     BOOLEAN-SSA-EXAMPLE.
       05  BS-SEGMENT-NAME                 PIC   X (08)      VALUE  ‘RSITEMSR’.
       05  BS-QUALIFIER                    PIC   X (01)      VALUE  ‘(‘.
       05  BS-FIELD-NAME1                  PIC   X (08)      VALUE  ‘ITEMNUMB’.
       05  BS-RELATIONAL-OPERATOR1         PIC   X (02)      VALUE  ‘  >‘.
       05  BS-FIELD-VALUE1                 PIC   X (06)      VALUE  ‘   ‘.
       05  BS-BOOLEAN-OPERATOR1            PIC   X (01)      VALUE  ‘|’.
       05  BS-FIELD-NAME 2                 PIC   X (08)      VALUE  ‘COLOR’.
       05  BS-RELATIONAL-OPERATOR2         PIC   X (02)      VALUE  ‘ =’.
       05  BS-FIELD-VALUE2                 PIC   X (08)      VALUE  ‘   ‘.
       05  BS-BOOLEAN-OPERATOR2            PIC   X (01)      VALUE  ‘&’.
       05  BS-FIELD-NAME3                  PIC   X (08)      VALUE  ‘MANFNAME’.
       05  BS-RELATIONAL-OPERATOR3         PIC   X (02)      VALUE  ‘  =’.
       05  BS-FIELD-VALUE3                 PIC   X (15)      VALUE  ‘   ’.
                          •
                          •
       05  BS-RIGHT-PAREN                  PIC    X (01)     VALUE  ‘)’.
```

structure of an SSA Using Boolean and relational Operators

When an SSA contains Boolean operators, the SSA is processed left-to right. The segment data must meet the conditions established by the SSA for DL/I to process the segment requested. A condition is a set of all field values up to a logical OR or the right parenthesis. If a logical OR is part of the SSA, a new condition is begun. The following example illustrates how DL/I establishes conditions to satisfy the call.

```
RSITEMSR (MANFNAME  = WESTINGHOUSE   &    COLOR   =GREEN)
          ------------------------------------------------------------------------
                                condition 1

RSITEMSR (MANFNAME  =RCA             |      MANFNAME  = ZENITH)
          -------------------------          --------------------------------
                  condition 1                        condition 2

RSITEMSR (MANFNAME  = TOSHIBA    |
          ---------------------------------
                   condition1

                  MANFNAME =   CURTIS MATHES    &    SIZE =25IN  )
                 ---------------------------------------------------------------------
                                condition 2
```

The first example has a single condition set that contains two requirements. The manufacturer must be Westinghouse and the color must be green. Both requirements must be met for the segment to be processed by DL/I.

The second example has two condition sets. The first condition set requires that the manufacturer be RCA. The second condition set requires that the manufacturer be Zenith. Since both requirements cannot be met simultaneously, the condition sets are separated by the Boolean operator OR (|). To return the segment to your program I/O Area, the manufacturer must be RCA or Zenith.

The third example also has two condition sets. The first condition set only requires that the manufacturer's name be Toshiba. The second condition set requires that the manufacturer 's name be Curtis Mathes and the size equal 25 inches. If either condition set is met, DL/I will process the segment.

You see that an SSA can be powerful and specific in helping DL/I process the segments you need. The SSA can contain as many qualifications as you need to satisfy your specific requirements. It is possible, though not often practiced, for the DBA to restrict the size of an SSA through the SSASIZE= keyword in the PSB.

Command codes

Index

DL/I offers another facility to enhance the power of the SSA. This facility is called command codes. Command codes are a special group of operators that allow you to direct how DL/I processed the segments specified in your call. The Null (-) command code tells DL/I that the SSA can use command codes in the future. You can find a list of command codes and their meanings in the IBM IMS Applications programming manual.

To take advantage of command codes, you must know how they are coded in as SSA. Figure below shows the common SSA format for using command codes.

PL/I

```
DCL   01  COMMAND_CODE_SSA_EXAMPLE,
            05    SEGMENT_NAME              CHAR (08)       INIT ('RSITEMSR'),
            05    ASTERISK                  CHAR (01)       INIT ('*'),
            05    COMMAND_CODE1             CHAR (01)       INIT ('-'),
            05    COMMAND_CODE2             CHAR (01)       INIT ('-'),
            05    LEFT _PAREN               CHAR (01)       INIT ('('),
            05    FIELD_NAME                CHAR (08)       INIT  ('ITEMNUMB'),
            05    RELATIONAL_OPERATOR       CHAR (02)       INIT ('  ='),
            05    FIELD_VALUE               CHAR (06)       INIT ('  '),
            05    RIGHT_PAREN               CHAR (01)       INIT (')');
```

COBOL

```
01  COMMAND-CODE-SSA-EXAMPLE.
   05  CC-SEGMENT-NAME                PIC  X (08)      VALUE  'RSITEMSR'.
   05  CC-ASTERISK                    PIC  X (01)      VALUE  '*'.
   05  CC-COMMAND-CODE1               PIC  X (01)      VALUE  '-'.
   05  CC-COMMAND-CODE2               PIC  X (01)      VALUE  '-'.
   05  CC-QUALIFIER                   PIC  X (01)      VALUE  '('.
   05  CC-FIELD-NAME                  PIC  X (08)      VALUE  'ITEMNUMB'.
   05  CC-RELATIONAL-OPERATOR         PIC  X (02)      VALUE  '= '.
  05  CC-FIELD-VALUE                  PIC  X (06)      VALUE  '  '.
   05  CC-RIGHT-PAREN                 PIC  X (01)      VALUE  ')'.
```

structure of an SSA Using command codes

Examine the SSA format in more detail. To this point, you have learned that position 9 of an SSA determines whether an SSA is qualified or unqualified. If position 9 is blank, the SSA is unqualified. If position 9 contains a left parenthesis '(' the SSA is qualified. Another option is using an asterisk * in position 9. If an asterisk is present in position 9, DL/I expects one or more command codes to follow. When using command codes, the first character position after the last command code determines whether an SSA is qualified or unqualified.

After the *, two Null command codes are specified. You do not need to code two. One is sufficient, However, coding two makes the SSA more flexible. When the command codes are initialized as null '--', they will be ignored by DL/I and the SSA acts just like an SSA

that has been used all along. If you want to use a command code, you need not code another SSA. Just set the value of COMMAND CODE to the one you wish to use Then issue the call.

The following are some examples of I/O areas of various SSAs:

1. RSITEMSR ITEMNUMB = 100123)
2. RSRENTER(SOCSECNM =373549012)
3. RSITEMSR(COLOR =WHITE|COLOR =GREEN)
4. RSPAYMTR(PAYDATE >=860105&PAYDATE <=860119)
5. RSRENTER*D-(SOCSECNM =366215531)

Number 1 shows an unqualified SSA because position nine is blank. It does not matter what follows the blank because everything after the blank is ignored by DL/I.

Number 2 is a qualified SSA because position nine contains a left parenthesis. It requests that DL/I find a renter segment with the Social security number, 373549012.

Number 3 is a qualified SSA using the OR (|) Boolean operator. For DL/I to process this request, the item must be the color white or green.

Number 4 is a qualified SSA requesting that DL/I find a Payment segment with a payment date greater than or equal to 860105, and (&) with a payment date less than or equal to 860119.

Number 5 is a qualified SSA using command code, D. This request tells DL/I to include the renter segment with social security number 366215531, along with the target segment in the I/O area.

Now that you have seen some sample SSA'S and how DL/I reacts to them, let's discuss some of the other common command codes.

F Command Code

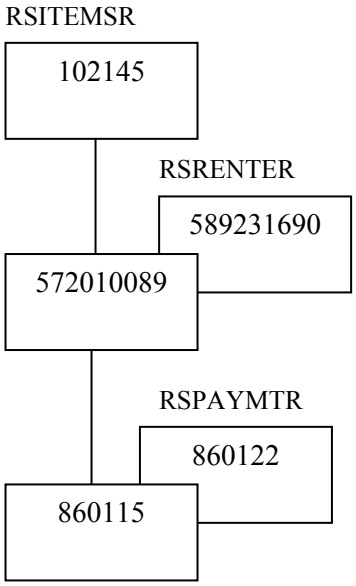
The F directs DL/I to either retrieve the first occurrence of the segment type or to insert the segment type as the first occurrence.

When the F is used retrieval calls, DL/I will retrieve the first occurrence of the segment type. If DL/I is currently positioned beyond the first occurrence, it will back up to the first occurrence, to satisfy the call. This is the only time a sequential call, like a GN or GNP, can cause DL/I to move backward in a data base.

If an insert call uses an F command code, DL/I will insert the segment occurrence as the first occurrence of that segment type based on the following criteria:

- If the segment is not keyed, the segment is inserted as the first segment occurrence of that segment type, whether or not there are existing occurrences of that segment type.
- If the segment is keyed multiple, DL/I will insert the segment as the first occurrence of that segment type with the same key value.

The use of the F overrides any insert rules specified in the DBD. The following examples show the effects of using F in retrieval and insert calls.



CALL EXAMPLE 1 - GU call

RSITEMSR(ITEMNUMB =102145)
RSRENTER*F-

CALL EXAMPLE 2 - ISRT call

RSITEMSR (ITEMNUMB =102145)
RSRENTER (SOCSECNM =572010089)
RSPAYMTR*F-

(NOTE: I/O area for the payment segment has a key of `860122')

The first example (the GN call), causes DL/I to retrieve the a first occurrence of the renter segment, Social Security number 572010089 under item segment 102145. This would be true even if the current position before the call was at the second renter (with the key of 589231690). In this situation, a GN call can actually back up in the data base when using the F command code.

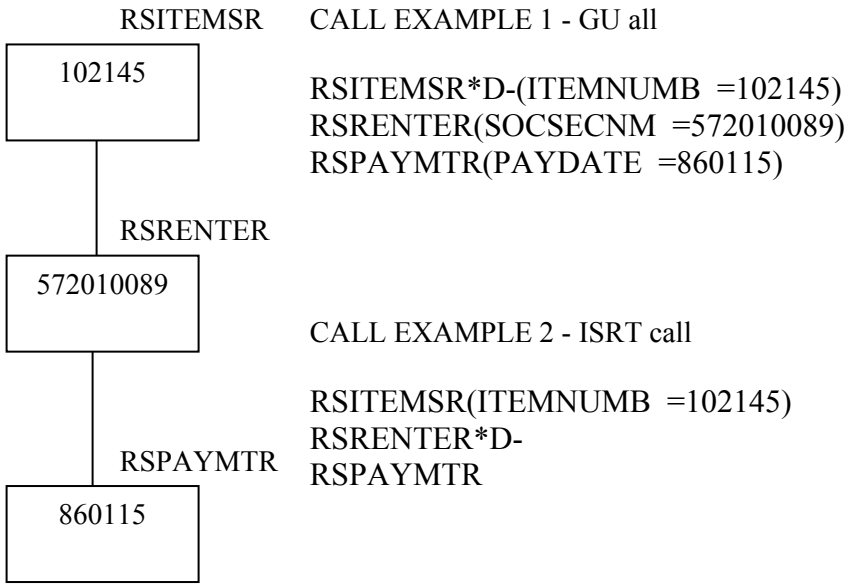
D Command Code

The D command code is used with retrieval and insert calls. It is considered a path command code because it allows you to retrieve or insert a hierarchical path of segments. To use the D command code, the PSB must have a processing option of P for PCB that contains the segment.

When using the D command code in retrieval calls, you must specify a D in the SSA for each segment you want included in the I/O area returned by DL/I.

When using an insert call, specify D in the first SSA for the path of segments to be inserted. All SSA'S starting from the SSA with the D must be unqualified.

The following examples show the effects of a call using the D command code.



Note:-The I/O area for the Payment segment has a key of 860122.

In example one (the GU all), DL/I will process the item segment with the item number, 102145, the renter segment with the Social Security number, 572010089, and the payment segment with a payment date, 860115. The I/O area will contain not only the target segment (as it always does in a retrieval call), but also the segment data for all segments whose SSA specified D. In this case, the SSA for item segment 102145 specified a D and will also be returned. Note the increased I/O area that is required to hold both the RSITEMSR as well as the RSPAYMTR segment.

In example two (the ISRT call), DL/I will not only insert the target segment, it will also insert any segment data for segments with unqualified SSA'S, as long as the SSA specifies the D or the segment is at a lower level in the path than the segment SSA with the D. In example two above, both the renter segment and the payment segment would be added.

The use of the D can reduce the number of calls required by DL/I to satisfy a request.

In the second example (the ISRT call), DL/I will insert the payment segment between dates 860115 and 860122 under the Social Security number, 572010089. This is because a payment segment occurrence already exists for date 860122. DL/I will honor the F and insert the duplicate date occurrence first. This only works if the sequence, or key, field in the DBD has an option M specified.

L Command Code

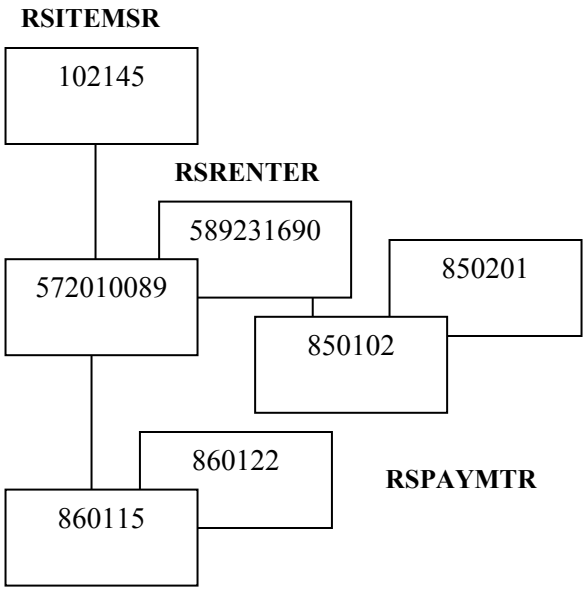
Using the L command code is similar to using the F command code. However, the last occurrence is used instead of the first.

The L command code is only valid on dependent segments.

P Command Code

The P causes DL/I to set parentage at a non-target level. Normally, DL/I sets parentage at the lowest level, or target segment, retrieved. If you do not want to use DL/I'S parentage, you can use the command code P to set parentage where you want it for all subsequent GNP calls. The P can only be specified on one SSA. If you specify it on more than one SSA, DL/I will use the lowest level segment with the P to set parentage.

The following is an example of the P.



GU call issued

RSITEMSR*P-(ITEMNUMB =102145)
RSRENTER(SOCSECNM =572010089)

GNP call issued for RSPAYMTR

RSPAYMTR

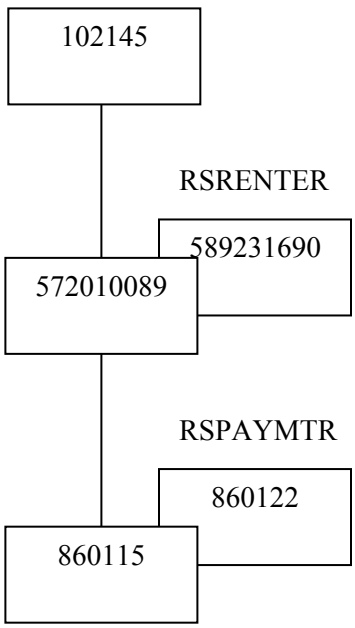
The preceding example shows two DL/I calls. The first call (the GU), causes DL/I to establish parentage at the item segment level on item number 102145. Normally, this call would cause DL/I to establish parentage at the renter segment level since it is the target segment. However, since P was specified in the SSA for the item segment, DL/I established parentage at item 102145. The second call (the GNP), causes DL/I to retrieve the first occurrence of the payment segment under the Social Security, 572010089.

Subsequent GNP calls cause DL/I to retrieve all of the remaining payment segments under item, 102145.

N Command Code.

The command code N is valid only with the REPL call. It is used when you are replacing a path of segments retrieved via a path call using the D command code and you do not want one or more of the segments in the path replaced.

The following example illustrates the use of the command code N.



GHU call issued

RSITEMSR*D-(ITEMNUMB =102145)
RSRENTER*D-(SOCSECNM =572010089)
RSPAYMTR(PAYDATE =860115)

REPL call issued

RSITEMSR
RSRENTER*N-
RSPAYMTR

The preceding GU all caused DL/I to retrieve into the I/O area the segment data for item 102145, renter 572010089, and payment date 860115. The D command codes at the item and renter segment level caused them to be retrieved from the I/O area, along with the target payment segment. The REPL call causes DL/I to replace the data in item segment 102145 and the payment segment 860115. The segment data for renter 572010089 is not replaced, even though it is in the I/O area. This is because the command code N is specified in the SSA for the renter segment.

If you wish to learn more about additional command codes, refer to your *IMS/VS Application Programming* manual in the section "Qualifying Your Call with Command Codes".

Program Assignment

Write a program to exercise the command codes on the data base you have created earlier.

Summary

DL/I is a data access language used to manipulate data in IMS data bases. It is not a programming language, rather an extension of PL/I, COBOL, and 370 Assembler languages. DL/I can be used in a CICS environment, although it is most often used in an IMS environment.

You must establish two-way communication between your application program and DL/I through internal entry points and external linkage. This communication allows your application program to request DL/I services. It also enables DL/I to return the requested data to your program along with information on the success or failure of your request.

DL/I provides a full array of functions to completely access and maintain IMS data bases. Additional functions are also provided to take advantage of other IMS facilities such as Checkpoint and Restart.

DL/I processes data based on the call. The call can be very specific with regard to data values, or it can be very general. Specific requests can be base on field value ranges, specific field values, or multiple field values or ranges.

IMS Batch Message Processing

Index

Upon completion of this section, you will understand both Batch-Oriented and Transaction-Oriented Batch Message Processing programs.

IMS supports both standard batch processing and a special type of batch processing called Batch Message Processing (BMP). Basically, a BMP allows an application program to share the resources of the IMS online environment. This is a very important consideration in today's IMS environment.

When an IMS batch application is executed, it requires a separate IMS control region. Depending on how the JCL is specified, an IMS batch application execution can have exclusive use of the IMS data bases allocated to the job. This means that if ten IMS batch jobs are running simultaneously, there are ten batch IMS control regions running, each independent of the other.

In the IMS online environment, only one IMS control region needs be started, and it controls the entire online environment. The online environment has exclusive use of all IMS data bases allocated to it. Many online application programs run simultaneously. They all share the resources available to the online system. The online system is increasingly required in today's IMS environment. In several installations, IMS online is available almost 24 hours a day. Many more installations are approaching a similar level of online availability. Because 24 hour a day online availability is becoming so widespread, the following questions are arising in more information processing installations:

- When do we run IMS batch jobs?
- When do we back up the IMS data bases?
- When do we perform maintenance on IMS?
- When do we perform maintenance on the IMS data bases?

Only a few of the questions must be answered to effectively support customers in an IMS environment. IMS can address these issues in several ways. One is through the support of the Batch Message Processing program. Two types of BMP programs are available to the application programmer, Batch-Oriented BMP and Transaction- Oriented BMP. The Batch-Oriented BMP is the most widely used.

Batch-Oriented BMP

Like the batch program, a BMP is initiated using JCL. A BMP can use OS/VS files, GSAM data bases, IMS data bases, and IMS online message queues (for output only). The IMS checkpointing facilities are available with BMP programs and is a major advantage to using a BMP. There are some differences one can think of Batch Oriented BMP's as online programs that run in batch, initiated via JCL. They share resources that are normally

available only to online programs. The data bases used in the online are available to the job, as is the online log file, that provides for automatic back out in the event of a program abend.

Transaction-Oriented BMP

The main differences between Batch-Oriented BMPs and transaction-Oriented BMPs are the following:

- Commit points are automatic in the Transaction-Oriented BMP.
  Note: A commit point (sometimes called a Synchronization or SYNC point) is one at which a unit of work is complete and IMS makes all data base changes permanent.

- The online message queues can be used for input or output in Transaction-Oriented BMPs.

Transaction-Oriented BMPs are typically used for processing output from an online or Message Processing Program (MPP). For example, you might use the BMP to produce reports based on a request from a Message Processing Program (online program). This frees the MPP from producing the report, improving the online response. Transaction-Oriented BMPs, like Batch-Oriented BMPs, are initiated through JCL, and scheduled on an as needed basis.

In this course, only the Batch-Oriented BMP will be addressed since it is the most widely used.

Creating BMP Applications

There are several considerations when creating a new application program to run as a BMP or when converting a Standard batch program to run as a BMP.

First, notify the group responsible for the IMS system generation. This group maintains the IMS environment and defines such variables as which application programs can run in the online system. Because a BMP shares online resources, it must be defined to the online system as a valid program. You must supply the PSB name and the fact that it is a BMP to this group. The next time the IMS online system is configured, your PSB will be added. You will then be able to run your program as a BMP.

Second, contact your Data Base Administrator and have your PSB changed to support the BMP operation. Only one task can be performed, as shown in Figure below. Add an additional keyword to the PSBGEN macro, CMPAT=YES. This is a requirement if you are going to share the resources of the IMS online system. But more importantly, it is a requirement if you intend to use the IMS Checkpoint facility.

```
PCB       TYPE=DB,NAME=DATABAS1....
            •
            •
PCB       TYPE=DB,NAME=DATABASn,....
```

```
PSBGEN    LANG=COBOL,PSBNAME=. . .,CMPAT=YES
END
```

PSB Using CMPAT Option for BMPs

Third, add an additional PCB pointer to your application program, as shown below. This pointer must be the first PCB in the list. It represents the I/O PCB or online message PCB assumed by the PSB when CMPAT=YES is specified.

PL/I

```
PLIPGM: PROC (IO_PCB_PTR,
              DB1_PCB_PTR,
              •
              •
              DBn_PCB_PTR) OPTIONS (MAIN):
```

COBOL

```
ENTRY 'DLITCBL'   USING  L-IO-PCB,
                         L-DB1-PCB,
                          •
                          •
                          L-DBn-PCB.
```

I/O PCB Pointer Specification in Program Code

Once you modify your application program as indicated, you will be ready to execute it. Running your program as a BMP requires no change in the program logic. Therefore, your program should operate as it did before it was converted into a BMP. The JCL needed is shown below.

```
//USER011     JOB     . . . .
//STEPnnn     EXEC    PGM=DFSRRC00,PARM=`BMP,pgmname,psbname'
//STEPLIB     DD      DSN=IMS.RESLIB,DISP=SHR
//              DD      DSN=ims.authoriz.lib,DISP=SHR
//osfile        DD      DSN=os.file.name,DISP=OLD
//SYSPRINT    DD      SYSOUT=class
```

JCL Sample for BMP Execution with Online System

The JCL is very terse because all required files for IMS, as well as IMS data bases, are shared from the online system and do not need to be specified in your JCL. Only O/S files must be specified.

What if there comes a time when the online system is not up and you need to run your program? Will it work? The answer is yes. Because your PSB is specified CMPAT=YES

and the first PCB pointer in your program is the I/O PCB, you should encounter no problem with program compatibility or execution. You only need to change your JCL, as shown below.

```
//USER011     JOB     ...
// STEPnnn    EXEC    PGM=DFSRRC00,  PARM='DLI, pgmname, psbname'
//STEPLIB     DD      DSN=IMSVSM1.RESLIB,DISP=SHR
//              DD DSN=ims.authoriz.lib,DISP=SHR
// IMS          DD DSN=your.dbdlib,DISP=SHR
//              DD DSN=your.psblib,DISP=SHR
//DFSRESLB  DD DSN=is.authoriz.lib,DISP=SHR
//DFSVSAMP DD DSN=buffer.info,DISP=SHR
//dbddd        DD DSN=ims.data.base,DISP=SHR
//osfile       DD DSN=os.file.name,DISP=OLD
//IEFRDER     DD DSN=log.file,DISP=(NEW,KEEP),
//                  UNIT=dasd,
//                  DCB=(RECFM=VB,LRECL=4148,BLKSIZE=4152)
//SYSPRINT    DD SYSOUT=class
```

JCL Sample for BMP Execution with Online System Down

This JCL is more complex. It is the same JCL you use to run a non-BMP program. Use the batch JCL to run your program as either a standard batch program or as a BMP. The only required change is the PARM = parameter on the EXEC card. Use PARM=DLI to run your program as a standard batch program, or PARM=BMP to run your program as a BMP. The additional DD cards in the JCL are ignored by IMS when PARM = 'BMP' is specified.

It is wise to practice setting up your IMS application program to run as a BMP, even though it may not initially be run as such. This offers greater flexibility if the need to run your program as a BMP arises. There will be no need to involve the Systems Engineer in running the program, nor will the Systems Engineer need to change it in any way.

There are several reasons why you may want to make your program a BMP. Each should be carefully considered because an application program tends to run more slowly as a BMP than as a standard batch program. The following are some considerations:

- Need for access to online message queues.
- Need for use of the checkpoint facility.
- Need for data base sharing
- Need for the ability to run as both a batch or a BMP.

Summary

Batch message processing permits application programs to use the IMS online environment resources on a shared bases. While there are two kinds of Batch message processing programs, Batch-Oriented and Transaction-Oriented, Batch-Oriented BMPs are more prevalent in today's IMS online environment.

Though less commonly used, the transaction-oriented BMPs offer advantages not available through Batch-Oriented BMPs, such as automatic commit points and use of online message queues for both input and output.

To create a BMP program or convert a standard program to run as a BMP, you need to carry out the following steps:

1. Notify the IMS system generation group that your program should be defined to the online system as a valid one.

2. Request that your DBA authorizes adding the necessary keyword to the PSBGEN macro, CMPAT=YES.

3. Include an additional PCB to your program. It must be the first PCB in the list because it indicates which I/O PCB, or online message PCB, is assumed by the PSB when CMPAT=YES is specified.

4. Request that an ACBGEN be performed by the DBA.

However, transaction-oriented BMPs provide automatic commit points and use of online message queues for both input and output. Running batch-oriented BMPs is similar to running online programs.

If the online system is up, you need only write JCL to specify O/S files when using batch message processing. If you need to run your program when the online system is down, JCL will resemble a standard batch program, but the PARM = parameter on the EXEC card must be changed from PARM=BMP to PARM=DLI.

Because BMP application programs involve more overhead than standard batch programs, only special requirements such as the need for checkpoint facilities, and DL/I online message queue access, should exist to justify their use.

IMS Checkpoint and Restart

Index

When you complete This section, you will be able to do the following:

- Plan the use of checkpoints
- Understand the importance of the Log File
- Issue a checkpoint call
- Restart an IMS batch job that uses checkpoints

When IMS is executing via online or batch, it provides a number of features that ensure data integrity or aid in the restart ability, or recoverability, of IMS data bases and application programs. These features include the following:

- Commit or synchronization points
- Logging
- Back out
- Restore/recovery
- Generalized Sequential Access Method
- Checkpoints
- Extended restart

Each of these features contributes to the integrity of IMS as an excellent data base management system (DBMS). You might decide to run IMS in a batch environment for any of several reasons. However, high input/output volume is likely to be your major consideration. Online programs are designed to handle a small amount of input and to produce a small amount of output. This allows a high degree of speed and efficiency associated with online programs to be maintained. However, high volumes of data must be moved in to and out of data bases and application programs. The batch environment is used for such high volume requirements. When an application program manipulates large quantities of data in IMS, the ability to restart a program at a point after the beginning can be an important option, if a system of program failure occurs. IMS provides facilities for checkpointing to keep track of program processing to restart the program at a point after the beginning. To effectively restart an application program, IMS must provide integrity for both your program and the data is uses.

IMS Integrity

IMS performs two functions to provide data base integrity:

- It allows only one program to process a given segment and its dependent segments in a data base record at a time. This is called program isolation.

- It prevents your program from accessing segments that have been deleted, replaced, or inserted by other programs until a commit point has been reached.

Commit or Synchronization Points

A commit point, also called a synchronization point, is a point at which a unit of work is complete, and IMS makes all data base changes permanent. The data is then made available to other programs for processing. All of this work is performed in buffers. When a program updates a data base, the physical update is not performed until the contents of the buffers are applied to the physical data base. These decisions are made by the operating system. However, when data is physically written to a data base, IMS considers these changes committed. The term, commit point, is derived from this IMS characteristic.

When a commit point is taken, the resources held by the application are freed. Thereafter, all segments held for updating are released to allow other programs to use them. Any messages generated by an online program or BMP are then sent to their final destinations.

Commit points can occur at the following times:

If the program terminates normally.

If the program issues a checkpoint call (this will be discussed later).

If the program completes a unit of work, such as the retrieval of a message from the message queue. This is performed automatically in most online programs (MPP) and transaction-oriented BMPs.

Logging

An important feature of an effective data base management system is its ability to log activity for use in restart and recovery procedures. IMS logs such activity during the execution of batch jobs. This includes the following information:

- IMS start up and shut down
- Program start and termination
- Data base changes

- Message Processing (BMP and online programs)
- Program checkpoints
- Commit points

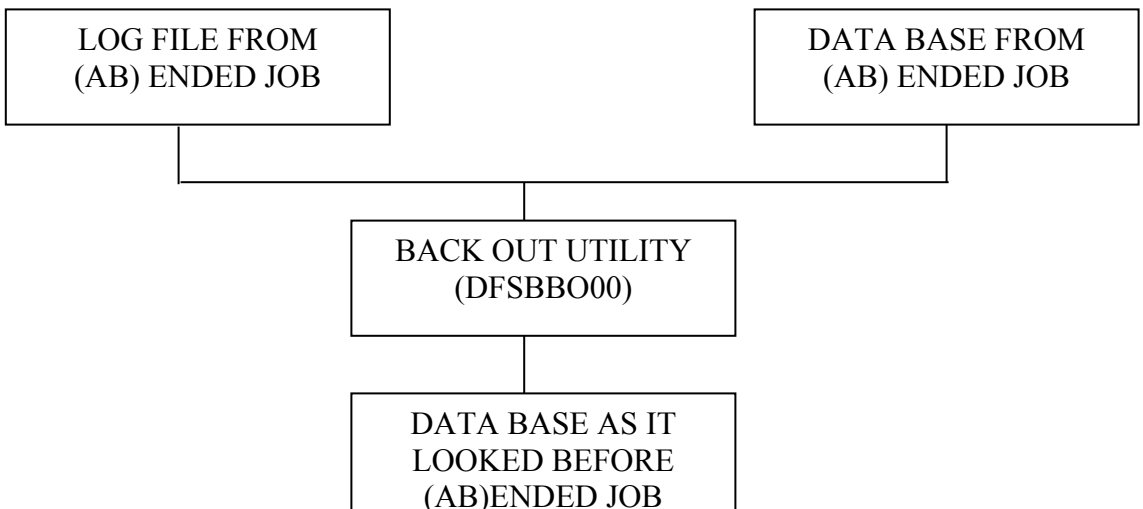
The commit point both creates and saves information. The information is saved on the IMS log file. The above list describes some instances in which information is stored in the IMS log file. Log records are always created in an IMS application. However, in the batch environment assigning a data set to the log file is optional. The DD name is IEFRDER and may be dummied our in batch. The log file is part of the JCL online control region and your program will access it automatically.

Logging is essential to the integrity, restartability, and recoverability of data bases. Therefore, if your application program performs any updating to IMS data bases, you must assign a log data set. You do not need to specify DCB. However, you must ensure that the DSN and DISP specify permanent data sets, not temporary ones. The type of information written to the log tape varies according to the action completed during the program processing. Examples are

Application program scheduled.
symbolic checkpoint requested.
IMS data base opened.
IMS data base closed.
Batch or BMP issued a Basic checkpoint,
IMS data base updated (after image).
IMS data base updated (before image).

Back Out Facility

Another important facility of IMS is the back out facility. This facility is used in conjunction with the logging facility. To better understand the back out facility, suppose that a batch job performing updates abends during execution. Because the job performs updates, the integrity of the data base is now in question, To restore the integrity of the data base prior to the execution of the abended job the back out facility can be used. The following shows the procedure used to back out of changes made to a data base.



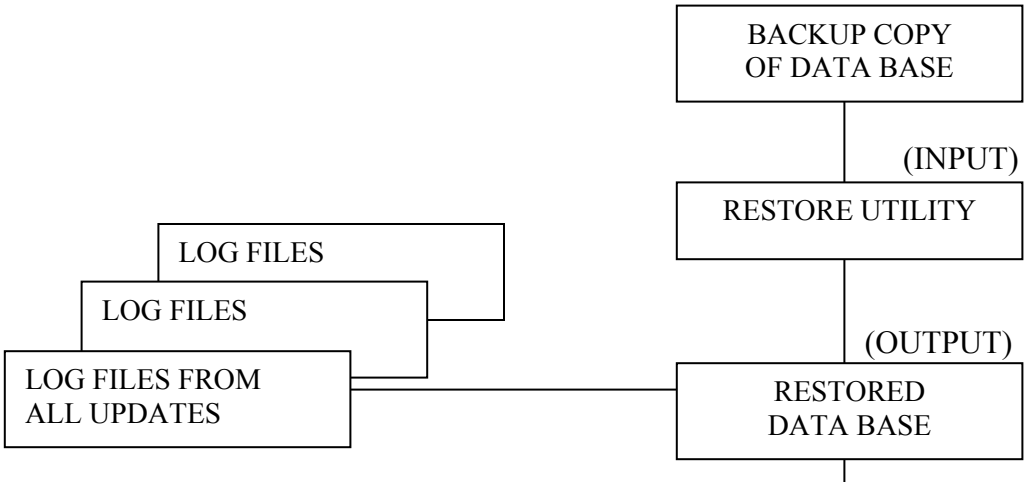
As you can see, the input to the back out utility is the log file that was active at the time the batch job abended in the data base being updated. This log contains all the changes made to the data base during the execution of the job. The output of the back out is the data base, minus all changes made during execution.

The first thing the back out utility does is go to the end of the log file. It does this because the log file is a sequential data set. Therefore the log records are in chronological order, The job of the back out utility is to undo all the changes made to the data base during the execution of the abended job. To do this, the back out proceeds in a backward manner starting from the end of the log file, working its way to the front of the log file. During this process, the back out utility looks for 50 (after image) and 51 (before image) log records for the data base. The 50 record is compared to the data base to ensure that this is what the database segment currently looks like. Then, IMS applies the corresponding 51 record to the data base to effectively undo the change. This process continues until all 50 and 51 records in the data base have been exhausted. When the back out is complete, the data base will contain exactly what it did before the abended job was run, and the integrity of the data base is maintained.

The process of doing a back out is called a *backward recovery* of a data base. The SE and DBA usually performs the back out.

Restore /Recovery Facilities

The back out facility is useful to maintain the integrity of a data base. However, in certain instances, the back out would not help restore data base integrity. For example, if an update batch job is running at the time the system encounters a head crash on the DASD device containing the data base being updated. Furthermore, say that the defective spot on the disk is right in the middle of the data base being updated. In this case, running a back out would not be useful, since I/O errors would be encountered during processing because of the damaged DASD device. To restore the integrity of the data base, the SE and DBA, would confer and perform the following steps, as shown in the diagram below.



(INPUT)

(INPUT)

(OUTPUT)

The first step in the recovery process is to use the latest back – up file, which may have been made hours, days, or even weeks prior to the point of the head crash, to restore the data base. The last thing you want to do is to inform the customer that they must reenter all online changes made since the latest back – up , as well as rerun all batch jobs to restore the data base. To avoid this, IBM has provided another utility – the recovery utility.

After restoring the data base run the recovery utility. The input to this utility is the restored data base and all log files from all online and batch jobs that have run since the time of the back up. These log files are input in chronological sequence. The recovery utility does not start at the end of the log file, as the back out did. It starts from the beginning of first log file. The recovery process locates the first 51 for the data base and makes sure the data base segment looks like that. Next, the corresponding 50 log record is applied to the data base. This same process continues until all log files have been processed At the end of the recovery process, the data base looks the way it did just before running the batch job during which the head crash occurred. All that is needed now is to rerun the batch job and hope for the best that another error does not occur!!.

The process of doing a restore and recovery is called forward recovery of a data base. Like the back out process, the IPC, along with SE and DBA assistance, performs the restore and recovery.

IMS Checkpoint Facilities

In addition to the log file, back up and recover utilities, another important IMS tool providing restart ability to IMS programs is the checkpointing facility.

The IMS checkpointing facilities are available to all programs whose PSB specifies CMPAT = YES. This is because any IMS program issuing checkpoint calls must do so through the input/output program communication block (I/O PCB).

Input/output Program Communication Block

In the IMS online environment, the I/O PCB is used to pass messages back and forth between online programs and terminal users. When used in an online program, the I/O PCB must be the first PCB because IMS automatically assumes that it is first in an online environment. You can, however, use the I/O PCB in a batch program. In fact, it is a requirement when using the IMS checkpoint facility. As in an online program, you must code a PCB pointer and it must be first in the PCB pointer list. Shown below is an example of how to code the I/O PCB in your program.

PL/1

```
CHKPPGM:   PROC   (  IO_PCB_PTR,
                     DB1_PCB_PTR,
                          •
                          •
                     DBN_PCB_PTR)    OPTIONS (MAIN);
```

COBOL

```
ENTRY  'DLITCBL'   USING  L –IO – PCB,
                          L – DB1 –PCB,
                               •
                               •
                          L – DBN – PCB.
```

I/O PCB pointer specification in program code

Additionally, you must code an I/O area used for the I/O PCB. Shown below is the definition of the I/O area for I/O PCB mask.

PL/1

```
DCL  IO_PCB_PTR                  POINTER;
DCL  01  LTPCB                   BASED (IO_PCB_PTR),
         05    LTERM_NAME        CHAR (08),
         05    FILLER            FIXED BIN (15),
         05    STATUS_CODE       CHAR (02),
         05    PREFIX,
               10  PCB_DATE      FIXED  DEC (7,0),
               10  PCB_TIME      FIXED  DEC (7,1),
               10  MSG_NUMBER    FIXED  BIN (31),
         05    MOD_NAME          CHAR (08),
         05    USER_ID           CHAR (08);
```

COBOL

```
LINKAGE  SECTION.
01  L-IO-PCB.
    05  L-LTERM-NAME          PIC  X (08).
    05  FILLER                PIC  S9(4)        COMP.
    05  L-STATUS-CODE         PIC  X  (02).
    05  L-PREFIX.
        10  L-PCB-DATE        PIC  S9(07)       COMP –3.
        10  L-PCB-TIME        PIC  S9(06)V9     COMP –3.
        10  L-MSG-NUMBER      PIC  S9(09)       COMP.
    05  L-MOD-NAME            PIC  X(08).
    05  L-USER-ID             PIC  X(08).
```

I/O PCB mask

Unlike online processing, batch programs must include the CMPAT =YES parameter in their PSBs, if they require checkpointing. The specification of CMPAT = YES in a batch PSB offers an advantage in addition to the ability to use the I/O PCB, it can run the batch job in a native IMS batch environment, or as a batch message processing program. In the BMP environment, IMS uses the I/O PCB to process messages. It also uses the IMS data bases allocated to the online control region. In the batch environment, the CMPAT = YES parameter tells IMS that the first PCB is the I/O PCB. But , the IMS data bases used in the program must be allocated in the JCL. This flexibility becomes more important as requirements for a 24 hour a day IMS online system increase, and IMS data bases are not available for native batch processing.

The checkpoint facility provides the ability to record all data base updates and the data base position at a given instant during IMS batch processing. If your program abends during execution, you have the ability to correct it and restart processing after the beginning of the program.

There are two kinds of checkpoint methods, basic and symbolic. Each will be explained in more detail later. Before choosing a checkpointing method, you must determine if checkpointing is appropriate. Typically, an online program does not require checkpointing. In the case of a batch program, you usually need checkpoints only if the program executes for several CPU minutes. Each installation provides guidelines to help you determine the desirability and recommended frequency for taking checkpoints.

Regardless of the method, checkpointing entails additional overhead. IMS performs checkpoints (commit points) automatically in online and transaction – oriented BMP programs. The frequency of these system checkpoints is determined when IMS is installed. Neither method of checkpointing supports the checkpointing of OS/VS files. If your program uses these files, you must choose one of the following options for checkpointing.

- Provide your own method of checkpointing and restart.
- Convert all OS/VS files into GSAM files as discussed in the “Generalized sequential Access Method ‘ section.

Another factor you must consider before checkpointing is that position within all IMS data bases (except GSAM) is destroyed when you take a checkpoint. You should carefully select where and when you take a checkpoint. The place at which you read a root segment is suitable for checkpointing because this is the easiest point at which you can reestablish position and parentage.

Taking a checkpoint provides you with the ability to restart processing after the beginning of your program. However, you must perform the checkpoint at the correct place and save the appropriate data to ensure the checkpoint works properly. Remember, IMS performs checkpointing according to your specifications. Consequently. You must plan and test the use of checkpoints to ensure that your program has the restart ability you expect.

Note: Although checkpoint and restart are available for your use, they are optional. This means that you must carefully assess the processing requirements of your application, and plan how best to use these facilities. Though the checkpoint facility requires overhead, a program that takes eight hours to process would certainty justify such overhead, should it abend in the seventh hour of processing.

In addition to the standard procedures carried out during a checkpoint, the following tasks are also performed.

- All modified data base buffers are written.
- The log record is written with the checkpoint ID specified in the call.
- A message is sent to the master console with the checkpoint ID.
- If the program processes messages, the next message is returned to the program's I/O area.

Generalized Sequential Access Method

Generalized sequential Access Method (GSAM) data bases are compatible with OS/VS sequential files created by SAM or VSAM access methods. If tape was used, BSAM is the applicable access method. If the data base is on DASD, BSAM or VSAM/ESDS, access method is used.

GSAM data bases have the following restrictions when used in IMS programs:

- They cannot be used in MPP (online) programs.
- They cannot have a hierarchy.
- They cannot use keys.
- They cannot use the DLET and REPL calls.
- Records can only be added at the end of the file.

The following are advantages of GSAM data bases:

- They can be created and accessed as OS/VS sequential files in non- DL/I programs.
- They can be checkpointed.
- They can be restarted using the XRST call. (XRST will be explained in detail later.)

Because GSAM file participate in the checkpoint and restart facilities, sequential files are converted to GSAM files for BMP processing. The steps required for converting a sequential data set into a GSAM data base are outlined in the following:

1. Create a DBD for the data base, as shown below.

```
DBD       NAME=dbdname,ACCESS = (GSAM, BSAM or VSAM)
DATASET   DD1=ddname,RECFM =  , RECORD=(max, min)
DBDGEN
FINISH
END
```

GSAM DBD Specifications

2.Include the GSAM PCB in the program's PSB as shown below.

```
PCB       TYPE =DB.,……..
          •
          •
PCB       TYPE = GSAM,NAME = dbdname,PROCOPT=G(S) or L (S)
          •
          •
PSBGEN    PSBNAME =
END
```

Note : GSAM PCBs must appear last in the PSB.

GSAM PSB Specification

After converting sequential files into GSAM data bases, you must change the application program to replace all OS/VS reads and writes to GSAM DL/I calls. You can use one of five basic calls against a GSAM data base:

OPEN – OPEN permits the explicit opening of the data base. It is not required because IMS automatically opens the data base the first time a get or insert call is issued.

CLSE – CLSE allows the explicit closing of the data base. It is not required because IMS automatically closes the data base upon termination of the program.

GU – GU permits the retrieval of a specific record in the database. Because GSAM does not allow segments or keys, you retrieve a record by using record search argument (RSA), which will be discussed later.

GN – GN allows the retrieval of the next sequential record in the data base.

ISRT –ISRT permits the insertion of a new record in the data base.

However, you can only add to the end of the data base. Accordingly, your JCL should specify DISP =MOD.

It is important to remember that, although you are dealing with GSAM data bases, you are using standard DL/I call formats, as shown below.

PL/1

```
CALL PLITDLI   (C_4,
                FUNC_CODE,                    (OPEN,  CLSE, ISRT, GU, GN)
                GSAM_PCB_PTR,
                IO_AREA,
                RSA_AREA);
```

COBOL

```
CALL  'CBLTDLI' USING  C-4,
                       FUNC-CODE,             (OPEN, CLSE, ISRT, GU, GN)
                       L-GSAM-PCB,
                       IO-AREA,
                       RSA-AREA.
```

GSAM DL/I call format

As with all data base calls, you will need the data base PCB mask I/O area, as shown below.

PL/I

```
DCL GSAM_PCB_PTR                      POINTER;
```

```
DCL 01 GSAM_PCB_MASK                BASED (GSAM_PCB_PTR),
       05 DBD_NAME                  CHAR(08),
       05 SEG_LEVEL                 CHAR(02),          <---(1)
       05 STATUS_CODE               CHAR(02),
       05 PROC_OPT                  CHAR(04),
       05 RSVRD                     FIXED BIN(31),
       05 SEG_NAME                  CHAR(08),          <---(2)
       05 KFB_LEN                   FIXED BIN(31),     <---(3)
       05 NUM_SENSEG                FIXED BIN(31),     <---(4)
       05 KFB_AREA.                                    <---(5)
          10 RSA                    CHAR(08),          <---(6)
          10 UNDEF_AREA             FIXED BIN(31);     <---(7)
```

COBOL

```
       LINKAGE SECTION.
       01 L-GSAM-PCB-MASK.
          05 L-DBD-NAME             PIC X(08).
          05 L-SEG-LEVEL            PIC XX.            <---(1)
          05 L-STATUS-CODE          PIC XX.
          05 L-PROC-OPT             PIC X(04).
          05 L-RSVRD                PIC S9(09) COMP.
          05 L-SEG-NAME             PIC X(08).         <---(2)
          05 L-KFB-LEN              PIC S9(09) COMP.   <---(3)
          05 L-NUM-SENSEG           PIC S9(09) COMP.   <---(4)
          05 L-KFB-AREA                                <---(5)
             10 L-RSA               PIC X(08).         <---(6)
             10 L-UNDEF-AREA        PIC S9(09) COMP.   <---(7)
```

Data Base GSAM PCB Mask

The following points explain the fields above:

1. Not used by GSAM

2. Not used by GSAM

3. always 12 for GSAM

4. not used by GSAM

5. this 12-byte area contains the RSA

6. only used by GSAM if RECFM=U

Data base call formats are determined by the type of call you are making. Each format is discussed in the following:

PARM CNT- Required for PL/I and optional for COBOL. This specifies the number of arguments in the call (not including PARM CNT).

FUNC CODE- Required argument specifying the function code for the type of DL/I call. For GSAM, it will be one of the five function codes described earlier (OPEN, CLSE, ISRT, GU, GN).

GSAM PCB- Required argument specifying the address of the PCB pointer for the GSAM data base.

IO AREA - Required argument for the GN, GU, and ISRT calls. When used with the GN or GU calls, this argument pertains to the I/O area of the record being retrieved. For the ISRT call, it pertains to the I/O area of the record being added.

RSA AREA-Required argument for the GU call and optional for the GN and ISRT calls. This specifies the RSA, a value that corresponds to a specific record in the data base. If BSAM tape or VSAM was used this value represents the Relative Byte Address (RBA). If BSAM DASD was used, the value represents the Track Record (TTR) address. Figure below shows the format of the RSA area.

PL/I

```
DCL 01  GSAM_RSA,
          05     BLOCK_IO     FIXED BIN(31),
          05     VOL_SEQ#     FIXED BIN(15),
          05     REC-DISP     FIXED BIN(15);
```

COBOL

```
01     GSAM_RSA,
          05     GS-BLOCK-IO    PIC S9(09) COMP.
          05     GS-VOL-SEQ#    PIC S9(04) COMP.
          05     GS-REC-DISP    PIC S9(04) COMP.
```

GSAM Record Search Argument I/O Area

The RSA is similar to the SSA in a non-GSAM call.

The content of the RSA is supplied by DL/I when an ISRT call is issued, as long as the I/O area and RSA area are specified in the call. If you save this value, you can use it in a GN or

GU call to retrieve a specific record, or to position yourself at a certain yourself at a certain spot within the data base.

Basic Checkpoint Method

The Basic method of checkpointing invokes IMS'S standard checkpoint functions. While all programs can use the Basic method, it is the only checkpoint method available to MPP'S, or online programs. The application program must provide its own method of restarting, because the extended restart call is not supported (see IMS Extended Restart). Neither OS/VS nor GSAM files are supported with the Basic checkpoint method. Figure below illustrates the necessary code invoking the Basic checkpoint in a program.

PL/I

```
DCL  CHKP_PTR    POINTER:
DCL  01     CHKP_ID,
          05      FILL              CHAR (04)    INIT(`CT7P'),
          05      CHKP_NUM      PIC `9999'   INIT(0);

CHKP_PTR = ADDR(CHKP_ID);
CHKP_NUM = CHKP_NUM+1;

CALL PLITDLI  (C_3,
                    C_CHKP_FUNC,
                    IO_PCB-PTR,
                    CHKP_PTR);

IF STATUS_CODE = ‘  ‘ THEN;
    ELSE    CALL    DLI_ERROR;
```

COBOL

```
01    CHKP-ID.
      05    FILLER      PIC  X(4)  VALUE    'CT7P'.
      05    CHKP-NUM    PIC  9(4)  VALUE    0.

ADD 1 TO CHKP-NUM.

CALL   'CBLTDLI'   USING  C-3,
                          C-CHKP-FUNC,
                          L-IO-PCB,
                          CHKP-ID.

IF     STATUS-CODE     NOT = BB-STATUS
         PERFORM     DLI-ERROR.
```

Basic Checkpoint Call

It is your responsibility to provide an 8-character checkpoint ID and to increment it as processing dictates. Your installation provides guidelines specifying checkpoint IDs. Once you issue the checkpoint call, the only valid status code is blank. A message is sent to the master console, as well as to the JES log, with the checkpoint ID and a date and time stamp. You will need this information, if a checkpoint restart becomes necessary.

A major disadvantage of the Basic checkpoint method is that you cannot change anything in your program prior to restarting.

Basic Checkpoint Summary

The following is true when using the Basic checkpoint method:

- Used in DL/I batch, BMP, and MPP.
- Releases all held IMS resources.
- Makes all data base changes (ISRT'S, REPL'S, and DLET'S) permanent.
- Destroys position in all non-GSAM data bases.
- Programmer must provide restart logic to restore variables and data base position.

Symbolic Checkpoint Method

The Symbolic checkpoint method also invokes the standard checkpoint functions of IMS. Only BMP and batch programs can use this method. Up to seven I/O areas can be saved with Symbolic checkpointing. This is a feature not available when you use the Basic method. Accordingly, you can alter your program code, other than the specified save areas, to either

correct a bug or to enhance processing. This is why the Symbolic method is preferable to the Basic checkpoint method. Symbolic checkpointing uses the extended restart call described in "IMS Extended Restart". An example of the Symbolic checkpoint method is shown below.

PL/I

```
DCL  01     IO_AREA_LTH,                          Longest segment or segment path.
            05     FILL     FIXED BIN(31) INIT(n); MUST be structure.

DCL 01 CHKP_ID,
          05 FILL     CHAR(06)     INIT(`CT7???'),
          05 CHKP_NUM  PIC `99'    INIT(0);

DCL CHKP_PTR                    POINTER;

DCL 01 CHKP_LTH1,
          05 FILL     FIXED BIN(31) INIT(n);

DCL 01 CHKP_AREA1,
          05 FIELD1....

CHKP_PTR      = ADDR(CHKP_ID);
CHKP_NUM      = CHKP_NUM+1;
PARM_CNT      = n;

CALL    PLITDLI   (  PARM_CNT,
                     CHKP_FUNC,
                     IO_PCB_PTR,
                     IO_AREA_LTH,
                     CHKP_PTR,
                     CHKP_LTH1,
                     CHKP_AREA1,
                     •
                     •
                     CHKP_LTH7,
                     CHKP_AREA7);

IF STATUS CODE= ‘  ‘ THEN;
       ELSE    CALL    DLI_ERROR;
```

Up to 7 Areas
Both Length and
I/O areas must
Be structures

Symbolic Checkpoint Call-PL/I

COBOL

```
01 IO_AREA_LTH        PIC S9(9) COMP VALUE +nnn.        Longest segment
                                                        or Segment path
```

```
01 CHKP-ID.
    05 FILLER            PIC    X(06) VALUE `CT7???'.
    05 CHKP-NUM          PIC    9(02) VALUE 0.

01 CHKP-LTH1             PIC S9 (09) COMP VALUE +nnn.

01 CHKP-AREA1.
   05 FIELDI....                                        Up to 7 areas
                    •
                    •

ADD 1 TO CHKP-NUM

CALL   'CBLTDLI'   USING  CHKP-FUNC,
                          IO-PCB,
                          IO-AREA-LTH,
                          CHKP-ID,
                          CHKP-LTH1,
                          CHKP-AREA1,
                          •
                          •
                          CHKP-LTH7,
                          CHKP-AREA7.

IF  STATUS-CODE     NOT=' '
          PERFORM        DLI-ERROR.
```

Symbolic Checkpoint Call-COBOL

Like the Basic call, the checkpoint ID and date and time stamp, are sent to the master IMS console and JES log. This information is important if you need to use the extended restart facility.

How does one determine what data needs to be saved in a the checkpoint area?
Imagine the following scenario:

- A program starts executing. It reads an input file that has 760 update transactions and starts updating a data base. As it updates, it keeps track of how many Adds, Replaces and Deletes are made to the data base.

- Checkpoints are taken after every 100 updates.

- After 730 updates, the program suffers an abend.

Since there was a checkpoint taken after every 100 updates, the first 700 updates were made permanent (committed) to the data base, but the last 30 were lost.

Suppose now that the program bug that caused the abend was corrected. The program was complied and relinked, and the program has restarted from the last checkpoint. The procedure for restarting a program will be described later. For now, just think of it conceptually.

Whenever a program is restarted, it always starts executing from the top of the program, just as in a normal run. This means that if you initialized your add, change, and delete accumulators to zero in the declarations, they will all be reset to zero. Therefore, if the program is restarted from the seventh checkpoint and it processes 60 more updates before ending normally. The add, replace and delete accumulators will only reflect the last 60 updates, even though 760 updates actually took place.

This is a situation where the checkpoint areas become important. For information to be saved during execution, and then later restored upon restarting, the variables will be saved in the log file every time a checkpoint call is issued. Then, when the program is restarted with the extended restart (XRST) call, the variables will be restored from the log file created during the first execution. In the example, if you had declared the add, replace, and delete accumulators in the checkpoint area, those variables would have been restored to whatever their values were at the seventh checkpoint. Then, when the program finished normally after the restart, those accumulators would accurately reflect that 760 updates were made.

Therefore, when coding programs that are going to take advantage of IMS checkpoint/restart facility, you must decide which variables you do not want to be reset to zero when your program has to be restarted. These variables usually include accumulators such as input/output record counters, page counters (for reports), and other types of counters, like the add, replace, and delete counters described above.

Even though up to seven save areas can be specified, typically only one is used. It is more efficient and maintainable to keep all the save information in a single area.

Symbolic checkpointing is the only checkpoint method that gives your program this restartability. If you use basic checkpointing, you must provide your own method of restoring variables in the program.

IMS Extended Restart

The IMS Checkpoint Facilities part of this section explained the use of checkpoints. This section will elaborate on how to use them to restart a batch job. Remember, this facility is valid only for Symbolic checkpoint method.

If a batch program abends and dynamic back out was not used, you must run the back out utility. This restores your IMS data bases to the last checkpoint, or to a checkpoint specified in the back out utility. Next, you must create restart JCL that specifies the checkpoint from which to restart. This JCL (shown in the following) is basically the same JCL used for batch IMS execution. The differences are the addition of the checkpoint ID in the parm, and the additional DD IMSLOGR.

```
//USER011     JOB     . . .
// STEPnnn    EXEC    PGM=DFSRRC00,
//             PARM='BMP,pgmname,psbname,.....chkptid'
                     or
//             PARM='DLI,pgmname,psbname,......chkptid'
//STEPLIB     DD   DSN=authoriz.ims.library,DISP=SHR
//             DD   DSN=system.loadlib,DISP=SHR
//IMS          DD   DSN=dbdlib,DISP=SHR
//             DD    DSN=psblib,DISP=SHR
//DFSVSAMP DD   DSN=buffer.info,DISP=SHR
//dbddd        DD   DSN=ims.data.base,DISP=OLD
//osfile       DD   DSN=osfile.name,DISP=OLD
//IMSLOGR    DD   DSN=log.file(0),DISP=OLD
//IEFRDER    DD   DSN=log.file(+1),DISP=(NEW,KEEP),
//             UNIT=dasd,DCB=(RECFM=VB,LRECL=4148,BLKSIZE=4152)
//SYSPRINT   DD   SYSOUT=class
```

JCL for Performing an Extended Restart

You can specify an 8-character checkpoint ID or a 12-character date and time stamp in the format of YYDDD/HHMMSS. Figures below show the code necessary for performing an extended checkpoint restart call.

PL/I

```
DCL  01     IO_AREA_LTH.
              05     FILL      FIXED BIN(31) INIT(n);
DCL  XRST_ID       CHAR(12);
DCL XRST_PTR      POINTER;
DCL  01     CHKP_LTH1.
           05  FILL      FIXED BIN(31) INIT(n);

DCL  01     CHKP_AREA1.
           05   FIELD1....

XRST_ID  =    ' ';
XRST_PTR  = ADDR (XRST_ID);
PARM_CNT = n;

CALL    PLITDLI   (  PARM_CNT,
                     XRST_FUNC,        /*  'XRST'  */
                     IO_PCB_PTR,
                     IO_AREA-LTH,
                     XRST_PTR,
                     CHKP_LTH1,
                     CHKP_AREA1,
                     •
```

```
                                    •
                                    CHKP_LTH7,
                                    CHKP_AREA7);

IF STATUS_CODE =' ' THEN
   DO;
      IF    XRST_ID = '  '
            THEN  CALL    NORMAL_START_ROUTINE;
       ELSE
            CALL RESTART_ROUTINE;
  END;
ELSE
  DO;
     CALL DLI_ERROR;
   END;
```

Extended Restart Call-PL/I

COBOL

```
01    IO-AREA-LTH   PIC S9(9)   COMP   VALUE +nnn.
01    XRST-ID          PIC X(12).
01    CHKP-LTH1    PIC S9(9) COMP VALUE +nnn.
01    CHKP-AREA1.
      05    FIELD1.....

MOVE   SPACES    TO    XRST-ID.
CALL   'CBLTDLI'   USING  XRST-FUNC,
                          L-IO-PCB,
                          IO-AREA-LTH,
                          XRST-ID,
                          CHKP-LTH1,
                          CHKP-AREA1,
                          •
                          •
                          CHKP-LTH7,
                          CHKP-AREA7.

 IF    STATUS-CODE     NOT = '  '
          CALL  DLI-ERROR.
 IF    XRST-ID = SPACES
          PERFORM    NORMAL-START-ROUTINE
     ELSE
          PERFORM RESTART-ROUTINE.
```

Extended Restart Call-COBOL

You should have only one XRST call in your program, and it should be the first call. You have the option of specifying the checkpoint ID or the date and time stamp in your program instead of in the JCL. If you specified the checkpoint ID in both the program and in JCL, IMS will use the JCL for the restart. The only valid status code is blank. If you use JCL for specifying the restart, you should check your I/O area for the checkpoint ID to determine whether or not it is blank. If it is blank, no restart was requested and you can proceed as though this were a new execution of the program. If it is not blank, a restart was requested and your program should process accordingly.

If a restart was requested, the XRST call automatically restores all of the variables you declared in the checkpoint area. It also restores the position in all data bases and GSAM files at the current position when that checkpoint was taken. (IMS does this by issuing internal GU calls). However, it is possible that between the time your program abended and restarted, another program executed and deleted that segment. The IBM IMS Application Programmers guide recommends that if this is a possibility in your system, you should structure your code to handle this possibility. For example, in your restart routine (the example above), you can issue a GU with a >= operator in the SSA so that either the segment itself would be retrieved, or the next sequential segment. To do this, the key of the segment has to be one of the variables saved in the checkpoint area.

Again, symbolic checkpointing is the only method that provides restartability with the restoration of variables and repositioning in all data bases. If you use Basic checkpoints, you will have to provide your own method of restoring variables and positioning in the data base.

Symbolic Checkpoint Summary

The following is true of the Symbolic checkpoint method:

- Used in DL/I batch or BMP-cannot be used with MPP.
- Releases all held IMS resources.
- Makes all data base changes (ISRT'S, REPL'S, and DLET'S) permanent.
- Destroys position in all non-GSAM data bases.
- Records programmer specified save areas (up to seven) and data base position in log file.
- Can use the XRST call to restart program.
- Does not checkpoint OS/VS files-must convert them to GSAM.

Extended Restart Summary

The following explains the use of the extended restart call:

- If used, it must be the first DL/I call issued by the program, and can only be issued once (regardless of whether the program is being restarted).

- It can only be used with the symbolic checkpoint method. It causes subsequent checkpoint calls to be recognized as symbolic.

- Restores save area variables and data base position (including GSAM) from checkpoint call.

- Program can be restarted from:

  1. A specific checkpoint ID
  2. A date/time stamp
  3. The last checkpoint taken (if the program is BMP and the log file is on DASD JCL only)

Summary

IMS provides such features as commit points, logging, and checkpoints to aid in the restart or recovery of IMS data base application programs. These features are especially advantageous when an application is processing large volumes of data and a program or system failure occurs.

IMS also provides program isolation and prevents your program from accessing segments that have been deleted, or inserted by other programs executing simultaneously with yours. IMS logs activity such as start up, shut down, data base changes, program checkpoints, and commit points for restart and recovery. All of this information is stored on the IMS log file. Use of the IMS log file is optional in batch, but required with online processing. You must assign a log data set for all data base updating to ensure integrity, restartability, and recoverability of the data base.

The checkpoint facilities provide the ability to take snapshots of program address space and data base position at any given instant during program processing. You select when and where to perform a checkpoint. However, it must be performed at the correct place and you must save the appropriate data to ensure that your program has the necessary restartability. There are two types of checkpointing-Basic and Symbolic.

The Basic checkpoint method can be used in all programs. However, MPP'S or online programs must use the Basic method. The major disadvantage of the Basic checkpoint method is that you cannot change anything in the program prior to restarting it.

The Symbolic checkpoint method is used only by BMP and batch programs. Up to seven I/O areas can be saved with the Symbolic method. Its major advantage is that you can change the program code (except save areas) to debug or enhance your program's processing.

GSAM data bases allow BMP programs to access sequential data files. GSAM data bases can be created and accessed as OS/VS sequential files in non-DL/I programs. GSAM data bases can be checkpointed. They can also be restarted using the XRST call. Sequential data

files are converted to GSAM files for BMP processing to allow the use the checkpoint and restart facilities.

Samples

Two sample programs are provided below. CBLPGM7 illustrates a basic checkpoint usage. The CBLPGM8 illustrates a Symbolic checkpoint and restart.. You can run CBLPGM7 as is and see the log messages whenever check points are taken. To test CBLPGM8 proceed as below:-

Make sure that the old IMSLOG data set is deleted and the DISP of this dataset is (NEW,KEEP) so that a new data set is created.

Run CBLPGM8 once on any of the databases created earlier, for example HISAM.

Rerun CBLPGM8 after supplying a CKPTID parm parameter in the invocation of DLIBATCH procedure.

Set the IMSLOG file created in the earlier run to IMSLOGR DD name which had been set to DUMMY.

Observe how the program proceeds to process the data base from the CKPTID.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    CBLPGM7.
*****************************************************************
*          E N V I R O N M E N T    D I V I S I O N             *
*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
*****************************************************************
*                  D A T A     D I V I S I O N                  *
*****************************************************************
 DATA DIVISION.
 FILE SECTION.
*****************************************************************
*                W O R K I N G    S T O R A G E                 *
*****************************************************************
 WORKING-STORAGE SECTION.
 01  WORK-AREAS.
     05 PARM-COUNT           PIC S9(09) COMP VALUE 3.
     05 FUNCN                PIC X(4)   VALUE 'GN  '.
     05 SEG-IO-AREA          PIC X(45).
     05 SEG-IO-RED REDEFINES SEG-IO-AREA.
        10  PATNO            PIC X(5).
        10  NAME             PIC X(10).
        10  ADDR             PIC X(30).
     05 SSA                  PIC X(09) VALUE 'PATIENT'.
     05 C-3                  PIC S9(08) COMP VALUE 3.
 01  PATIENT-SSA-Q.
     05  SEG                 PIC X(8) VALUE 'PATIENT'.
     05  FILLER              PIC X VALUE '('.
     05  FIELD               PIC X(8) VALUE 'PATNO'.
     05  OP                  PIC X(2) VALUE ' ='.
     05  FLD-VALUE           PIC X(5).
     05  FILLER              PIC X VALUE ')'.
 01   CHKP-ID.
     05  FILLER   PIC  X(4)  VALUE  'CT7P'.
     05  CHKP-NUM  PIC  9(4)  VALUE  0.
 01 CONSTANTS.
     05 C-GB                 PIC X(2)  VALUE 'GB'.
     05 C-GU                 PIC X(4)  VALUE 'GU'.
 01 SWITCHES.
     05 S-FLAG-BIT            PIC X(01) VALUE LOW-VALUES.
       88 S-FLAG                        VALUE HIGH-VALUES.
 01 C-CHKP-FUNC               PIC X(4) VALUE 'CHKP'.
 LINKAGE SECTION.
 01 IOPCB.
     05 FILLER                 PIC X(8).
     05 FILLER                 PIC S9(4) COMP.
     05 IOSTAT                 PIC X(2).
 01 PCBMASK.
     05 DBD-NAME               PIC X(08).
     05 SEG-ID                 PIC X(02).
     05 STATS                  PIC X(02).
     05 PROCOPT                PIC X(04).
     05 FILLER                 PIC X(04).
     05 SEGMENT-NAME           PIC X(08).
     05 LENGTH-FDBK            PIC S9(05) COMP.
```

```
     05 NUMBER-SENSEGS          PIC S9(05) COMP.
     05 KEY-FDBK-AREA           PIC X(21).
*****************************************************************
*                  PROCEDURE DIVISION                           *
*****************************************************************
 PROCEDURE DIVISION.
     ENTRY 'DLITCBL' USING IOPCB PCBMASK
     PERFORM MAIN-PARA
     PERFORM FINAL-PARA.
 MAIN-PARA.
     CALL 'CBLTDLI' USING FUNCN,
               PCBMASK,
               SEG-IO-AREA.
     DISPLAY 'SEGMENT NAME '  SEGMENT-NAME  STATS
     DISPLAY 'SEGMENT AREA '  SEG-IO-AREA
     DISPLAY 'KEY          ' KEY-FDBK-AREA
     PERFORM UNTIL S-FLAG
     MOVE SPACES TO SEG-IO-AREA
     CALL 'CBLTDLI' USING FUNCN,
                     PCBMASK,
                     SEG-IO-AREA
     IF STATS NOT = C-GB
         DISPLAY 'SEGMENT NAME '  SEGMENT-NAME  STATS
         DISPLAY 'SEGMENT AREA '  SEG-IO-AREA
         DISPLAY 'KEY          ' KEY-FDBK-AREA
     END-IF
     IF STATS = C-GB
         DISPLAY 'NON BLANK FROM GN CALL ' STATS
         MOVE HIGH-VALUES TO S-FLAG-BIT
     END-IF
     IF SEGMENT-NAME = 'PATIENT'
       ADD 1 TO CHKP-NUM
       CALL  'CBLTDLI'  USING  C-3,
            C-CHKP-FUNC,
            IOPCB,
            CHKP-ID
       MOVE PATNO TO FLD-VALUE
       DISPLAY 'FLD-VALUE' FLD-VALUE
       CALL 'CBLTDLI' USING C-GU,
               PCBMASK,
               SEG-IO-AREA,
               PATIENT-SSA-Q
       DISPLAY 'STAT' STATS
     END-IF
     END-PERFORM.
 FINAL-PARA.
      DISPLAY  'END OF PROGRAM'
      GOBACK.
 A100-EXIT.
      EXIT.
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    CBLPGM8.
*****************************************************************
*          E N V I R O N M E N T    D I V I S I O N             *
*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
*****************************************************************
*                  D A T A     D I V I S I O N                  *
*****************************************************************
 DATA DIVISION.
 FILE SECTION.
*****************************************************************
*                W O R K I N G    S T O R A G E                 *
*****************************************************************
 WORKING-STORAGE SECTION.
 01  WORK-AREAS.
     05 PARM-COUNT           PIC S9(09) COMP VALUE 3.
     05 FUNCN                PIC X(4)   VALUE 'GN  '.
     05 SEG-IO-AREA          PIC X(45).
     05 SEG-IO-RED REDEFINES SEG-IO-AREA.
        10  PATNO            PIC X(5).
        10  NAME             PIC X(10).
        10  ADDR             PIC X(30).
     05 SSA                  PIC X(09) VALUE 'PATIENT'.
     05 C-6                  PIC S9(08) COMP VALUE 6.
 01  IO-AREA-LTH             PIC S9(9) COMP VALUE 4.
 01  IO-AREA                 PIC S9(9) COMP VALUE 0.
 01  DATA-AREA-LTH           PIC S9(9) COMP VALUE 45.
 01  DATA-AREA               PIC X(45).
 01  PATIENT-SSA-Q.
     05  SEG                 PIC X(8) VALUE 'PATIENT'.
     05  FILLER              PIC X VALUE '('.
     05  FIELD               PIC X(8) VALUE 'PATNO'.
     05  OP                  PIC X(2) VALUE ' ='.
     05  FLD-VALUE           PIC X(5).
     05  FILLER              PIC X VALUE ')'.
 01   CHKP-ID.
     05  FILLER   PIC  X(4)  VALUE  'CT7P'.
     05  CHKP-NUM  PIC  9(4)  VALUE  0.
 01 CHKP-RESTART   PIC X(12) VALUE SPACES.
 01 CONSTANTS.
     05 C-GB                 PIC X(2)  VALUE 'GB'.
     05 C-GU                 PIC X(4)  VALUE 'GU'.
 01 SWITCHES.
     05 S-FLAG-BIT            PIC X(01) VALUE LOW-VALUES.
       88 S-FLAG                        VALUE HIGH-VALUES.
 01 C-CHKP-FUNC               PIC X(4) VALUE 'CHKP'.
 01 C-XRST-FUNC               PIC X(4) VALUE 'XRST'.
 LINKAGE SECTION.
 01 IOPCB.
     05 FILLER                 PIC X(8).
     05 FILLER                 PIC S9(4) COMP.
     05 IOSTAT                 PIC X(2).
 01 PCBMASK.
     05 DBD-NAME               PIC X(08).
```

```
           05 SEG-ID                  PIC X(02).
           05 STATS                   PIC X(02).
           05 PROCOPT                 PIC X(04).
           05 FILLER                  PIC X(04).
           05 SEGMENT-NAME            PIC X(08).
           05 LENGTH-FDBK             PIC S9(05) COMP.
           05 NUMBER-SENSEGS          PIC S9(05) COMP.
           05 KEY-FDBK-AREA           PIC X(21).
      ****************************************************************
      *                  PROCEDURE DIVISION                          *
      ****************************************************************
       PROCEDURE DIVISION.
           ENTRY 'DLITCBL' USING IOPCB PCBMASK
           PERFORM MAIN-PARA
           PERFORM FINAL-PARA.
       MAIN-PARA.
           CALL  'CBLTDLI'  USING  C-6,
                  C-XRST-FUNC,
                  IOPCB,
                  IO-AREA-LTH,
                  CHKP-RESTART,
                  DATA-AREA-LTH,
                  DATA-AREA
           IF CHKP-RESTART NOT= SPACES
               MOVE DATA-AREA TO SEG-IO-AREA
               DISPLAY 'SEGMENT AREA '  SEG-IO-AREA
           END-IF
           CALL 'CBLTDLI' USING FUNCN,
                     PCBMASK,
                     SEG-IO-AREA.
           DISPLAY 'SEGMENT AREA '  SEG-IO-AREA
           PERFORM UNTIL S-FLAG
           MOVE SPACES TO SEG-IO-AREA
           CALL 'CBLTDLI' USING FUNCN,
                            PCBMASK,
                            SEG-IO-AREA
           IF STATS NOT = C-GB
               DISPLAY 'SEGMENT AREA '  SEG-IO-AREA
           END-IF
           IF STATS = C-GB
               DISPLAY 'NON BLANK FROM GN CALL ' STATS
               MOVE HIGH-VALUES TO S-FLAG-BIT
           END-IF
           IF SEGMENT-NAME = 'PATIENT'
             ADD 1 TO CHKP-NUM
      *      ADD 1 TO DATA-AREA
             MOVE SEG-IO-AREA TO DATA-AREA
             CALL  'CBLTDLI'  USING  C-6,
                   C-CHKP-FUNC,
                   IOPCB,
                   IO-AREA-LTH,
                   CHKP-ID,
                   DATA-AREA-LTH,
                   DATA-AREA
             MOVE PATNO TO FLD-VALUE
             DISPLAY 'FLD-VALUE' FLD-VALUE
             CALL 'CBLTDLI' USING C-GU,
                     PCBMASK,
```

```
                SEG-IO-AREA,
                PATIENT-SSA-Q
          DISPLAY 'STAT' STATS
        END-IF
        END-PERFORM.
    FINAL-PARA.
         DISPLAY  'END OF PROGRAM'
         GOBACK.
    A100-EXIT.
         EXIT.
```

Variable Length Segments

Index

Using Variable-Length Segments

Variable-length segments are simply segments whose length can vary in occurrence of some segment types. A database can contain both variable-length segment and fixed-length segment types. Variable-length segments can be used for HISAM, HDAM, and HIDAM databases.

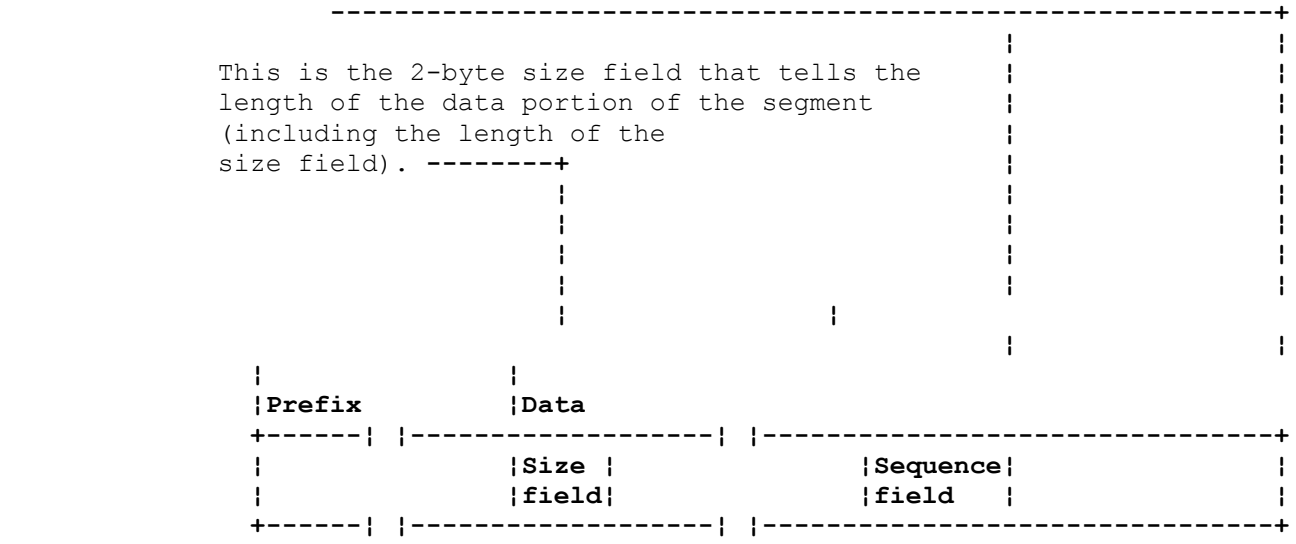
How to Specify Variable-Length Segments

It is the data portion of a variable-length segment whose length varies. The data portion varies between a minimum and a maximum number of bytes. As shown in below, you specify minimum and maximum size in the BYTES= keyword in the SEGM statement in the DBD. Because IMS needs to know the length of the data portion of a variable-length segment, you include a 2-byte size field in each segment when loading it. The size field is in the data portion of the segment. The length of the data portion you specify must include the two bytes used for the size field. If the segment type has a sequence field, the minimum length specified in the size field must equal at least the size field and all data to the end of the sequence field.

This is the DBD statement that defines the size range of the variable-length segment:

```
SEGM  NAME=SEGNAME,BYTES=(MAXBYTES,MINBYTES)
```

This is the minimum (MINBYTES) and maximum (MAXBYTES) size of the segment. Notice that if a sequence field exists for the segment type, MINBYTES must equal at least the length of the size field plus all data to the end of the sequence field.



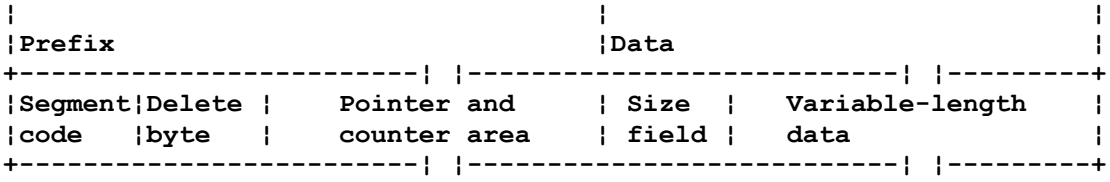
How Variable-Length Segments Are Specified

How Variable-Length Segments Are Stored and Processed

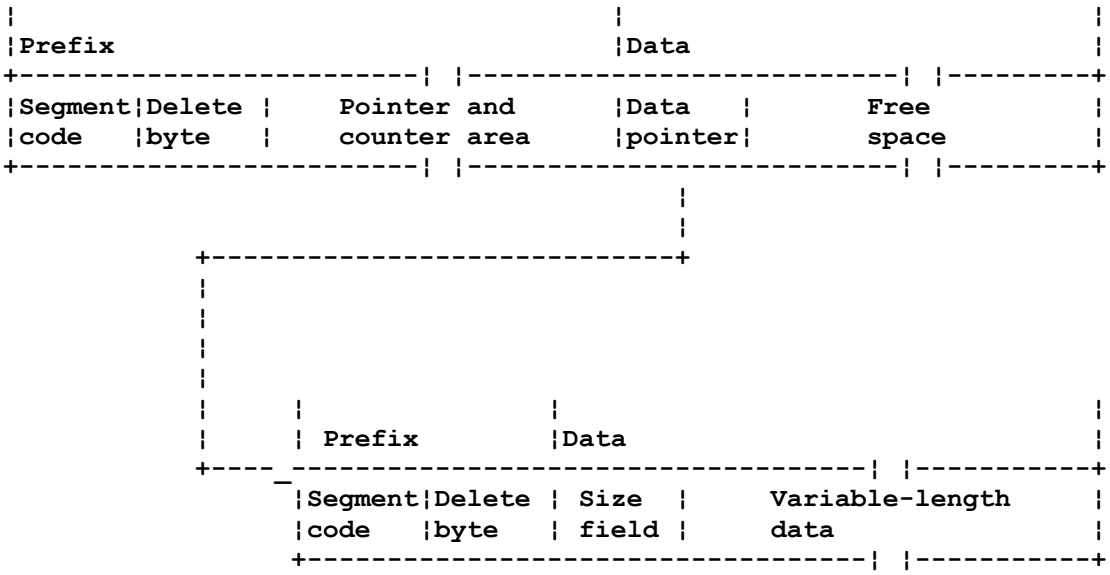
When a variable-length segment is initially loaded, the space used to store its data portion is the length specified in the MINBYTES operand or the length specified in the size field, whichever is larger. If the space in the MINBYTES operand is larger, more space is allocated for the segment than is required. The additional space can be used when existing data in the segment is replaced with data that is longer.

Figure below shows the format of variable-length segments. The prefix and data portion of HDAM and HIDAM variable-length segments can be separated in storage when updates occur. When this happens, the first four bytes following the prefix point to the separated data portion of the segment.

This is the format of a HISAM variable-length segment. It is also the format of an HDAM or HIDAM variable-length segment when the prefix and data portion of the segment have not been separated in storage.



This is the format of an HDAM or HIDAM variable-length segment when the prefix and data portion of the segment have been separated in storage.



Format of Variable-Length Segments

After a variable-length segment is loaded, replace operations can cause the size of data in it to be either increased or decreased. When the length of data in an existing HISAM segment is increased, the logical record containing the segment is rewritten to acquire the additional space. Any segments displaced by the rewrite are put in overflow storage. Displacement of

segments to overflow storage can affect performance. When the length of data in an existing HISAM segment is decreased, the logical record is rewritten so all segments in it are physically adjacent.

When a replace operation causes the length of data in an existing HDAM or HIDAM segment to be increased, one of two things can happen:

- If the space allocated for the existing segment is long enough for the new data, the new data is simply placed in the segment. This is true regardless of whether the prefix and data portions of the segment were previously separated in the data set.

- If the space allocated for the existing segment is not long enough for the new data, the prefix and data portions of the segment are separated in storage. IMS puts the data portion of the segment as close to the prefix as possible. Once the segment is separated, a pointer is placed in the first four bytes following the prefix to point to the data portion of the segment. This separation increases the amount of space needed for the segment, because, in addition to the pointer kept with the prefix, a 1-byte segment code and 1-byte delete code are added to the data portion of the segment (see Figure 80). In addition, if separation of the segment causes its two parts to be stored in different blocks, two read operations will be required to access the segment. When a replace operation causes the length of data in an existing HDAM or HIDAM segment to be decreased, one of three things can happen:

- If prefix and data are not separated, the data in the existing segment is replaced with the new, shorter data followed by free space.

- If prefix and data are separated but sufficient space is not available immediately following the original prefix to recombine the segment, the data in the separated data portion of the segment is replaced with the new, shorter data followed by free space.

- If prefix and data are separated and sufficient space is available immediately following the original prefix to recombine the segment, the new data is placed in the original space, overlaying the data pointer. The old separated data portion of the segment is then available as free space in HD databases.

When to Use Variable-Length Segments

Use variable-length segments when the length of data in your segment varies, for example, with descriptive data. By using variable-length segments, you do not need to make the data portion of your segment type as long as the longest piece of descriptive data you have. This saves storage space. Note, however, that if you are using HDAM or HIDAM databases and your segment data characteristically grows in size over time, segments will split. If a segment split causes the two parts of a segment to be put in different blocks, two read operations will be required to access the segment until the database is reorganized. So variable-length segments work well if segment size varies but is stable (as in an address segment). Variable-length segments might not work well if segment size typically grows (as in a segment type containing a cumulative list of sales commissions).

What Application Programmers Need to Know about Variable-Length Segments

If you are using variable-length segments in your database, you need to let application programmers who will be using the database know this. They need to know which of the segment types they have access to are variable in length and the maximum size of each of these variable-length segment types. In calculating the size of their I/O area, application programmers must use the maximum size of a variable-length segment. In addition, they need to know that the first two bytes of the data portion of a variable-length segment contain the length of the data portion including the size field.

Working with the application programmer, you should devise a scheme for accessing data in variable-length segments. You should devise a scheme because if variable-length fields and fixed-length fields in a segment are mixed, the application program has no way of knowing where specific fields begin. One way to solve this problem is to put the size of a variable-length field at the beginning of the variable-length field. If a segment has only one variable-length field, it can be made the last field in the segment. If it is at all possible, the simplest scheme is to have only one field in a variable-length segment.

Secondary Indexes

Index

Characteristics of Secondary Indexes

Secondary indexing is a solution to the different processing requirements of various applications. It allows you to have an index based on any field in the database, and not just the key field in the root segment.

Secondary indexes can be used with HISAM, HDAM, and HIDAM databases. A secondary index is in its own separate database and must use VSAM as its access method. Because a secondary index is in its own database, it can be processed as a separate database.

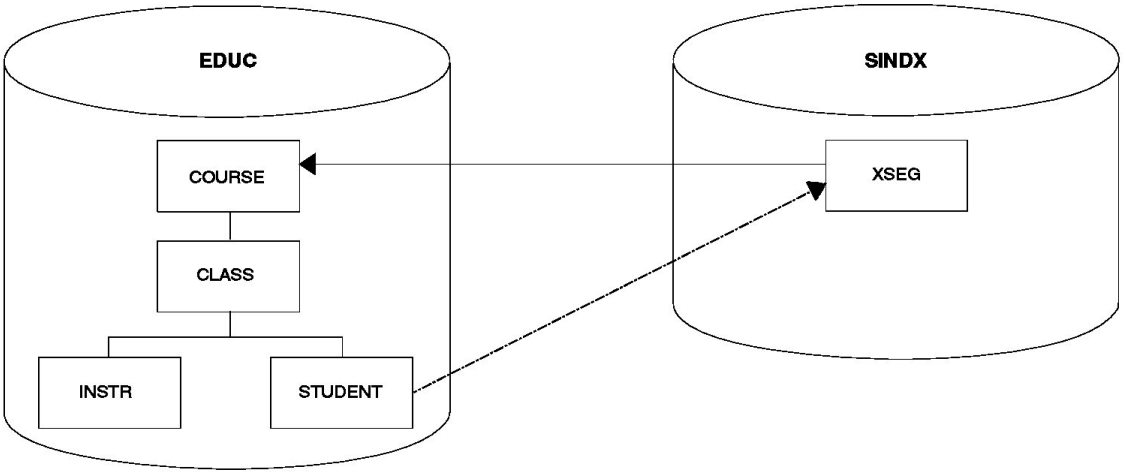
Secondary indexes are invisible to the application program. When an application program needs to do processing using the secondary index, this fact is communicated to IMS by coding the PROCSEQ= parameter in the PCB. If an application program needs to do processing using the regular processing sequence, PROCSEQ= is simply not coded. If the application program needs to do processing using both the regular processing sequence and the secondary index, the application program's PSB must contain two PCBs, one with PROCSEQ= coded and one without.

When two PCBs are used, it enables an application program to use two paths into the database and two sequence fields. One path and sequence field is provided by the regular processing sequence, and one is provided by the secondary index. The secondary index gives an application program both an alternative way to enter the database and an alternative way to sequentially process database records.

A final characteristic of secondary indexes is that there can be 32 secondary indexes for a segment type and a total of 1000 secondary indexes for a single database.

Segments Used for Secondary Indexes

Now that the concept and general characteristics of secondary indexes have been explored, we next look at how a secondary index works. As shown in Figure below, to set up a secondary index, three types of segments must be defined to IMS. The three types of segments are pointer, target, and source segments.



Prefix		Data SRCH	SUBSEQ
DB	RBA	STUDNM	/SX1

This is the content of the pointer segment. Since a student's name may not be unique, a subsequence field is used.

DBD for EDUC Database

```
DBD     NAME=EDUC,ACCESS=HDAM,...
SEGM    NAME=COURSE...
FIELD   NAME=(COURSECD,...
LCHILD  NAME=(XSEG,SINDX),PTR=INDX
XDFLD   NAME=XSTUDNT,SEGMENT=STUDENT,
SRCH=STUDNM,SUBSEQ=/SX1
SEGM    NAME=CLASS...
FIELD   NAME=(EDCTR,...
SEGM  NAME=INSTR...
FIELD   NAME=(INSTNO,...
SEGM  NAME=STUDENT...
FIELD   NAME=SEQ,...
FIELD   NAME=STUDNM,BYTES=20,START=1
FIELD   NAME=/SX1
DBDGEN
FINISH
END
```

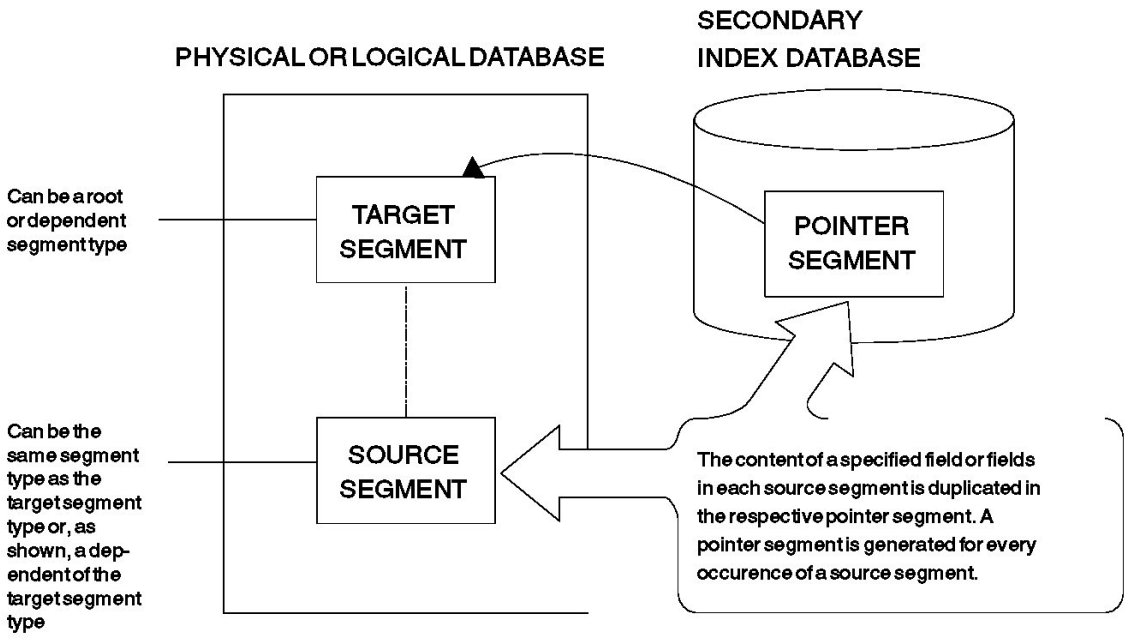
DBD for SINDX Database

```
DBD     NAME=SINDX,ACCESS=INDEX
SEGM    NAME=XSEG,...
FIELD   NAME=(XSEG,SEQ,U),BYTES=24,
START=1
LCHILD  NAME=(COURSE,EDUC),
INDEX=XSTUDNT,PTR=SNGL
DBDGEN
FINISH
END
```

- Pointer Segment. The pointer segment is contained in the secondary index database and is the only type of segment in the secondary index database. Its format looks like this:

```
     ¦                    ¦                                          ¦
     ¦Prefix              ¦Data                                      ¦
     +--------------------------------¦  ¦-----------¦  ¦---------¦
     ¦Delete¦RBA of the   ¦ Key ¦  Optional   ¦Symbolic pointer     ¦
     ¦byte  ¦segment to be¦field¦   fields    ¦to the segment       ¦
     ¦      ¦retrieved    ¦     ¦            ¦to be retrieved      ¦
     +--------------------------------¦  ¦-----------¦  ¦---------+
Bytes    1         4                  Varies            Varies
```

The first field in the prefix is the delete byte. The second field is the address of the segment the application program retrieves from the regular database. (This field is not present if the secondary index uses symbolic pointing. Symbolic pointing is pointing to a segment using its concatenated key. HIDAM and HDAM may use symbolic pointing, however, HISAM must use symbolic pointing.)



Segments Used for Secondary Indexes

The data portion of the segment contains the key field and other optional fields. The key field contains the key. The key gets the application program to the correct pointer segment in the secondary index. The application program uses the address in the prefix of the pointer segment to retrieve the necessary segment from the database. If symbolic pointing is used, the key will get the application program to the correct pointer segment in the secondary index. The application program uses the symbolic pointer in the pointer segment to retrieve the necessary segment from the database.

- Target Segment. The target segment is in the regular database, and it is the segment the application program needs to retrieve. A target segment is the segment to which the pointer segment points. The target segment can be at any one of the 15 levels in the database, and it is accessed directly using the RBA or symbolic pointer stored in the pointer segment. Physical parents of the target segment are not examined to retrieve the target segment.

- Source Segment. The source segment is also in the regular database. The source segment contains the field (or fields) that the pointer segment has as its key field. Data is copied from the source segment and put in the pointer segment's key field. The source and the target segment can be the same segment, or the source segment can be a dependent of the

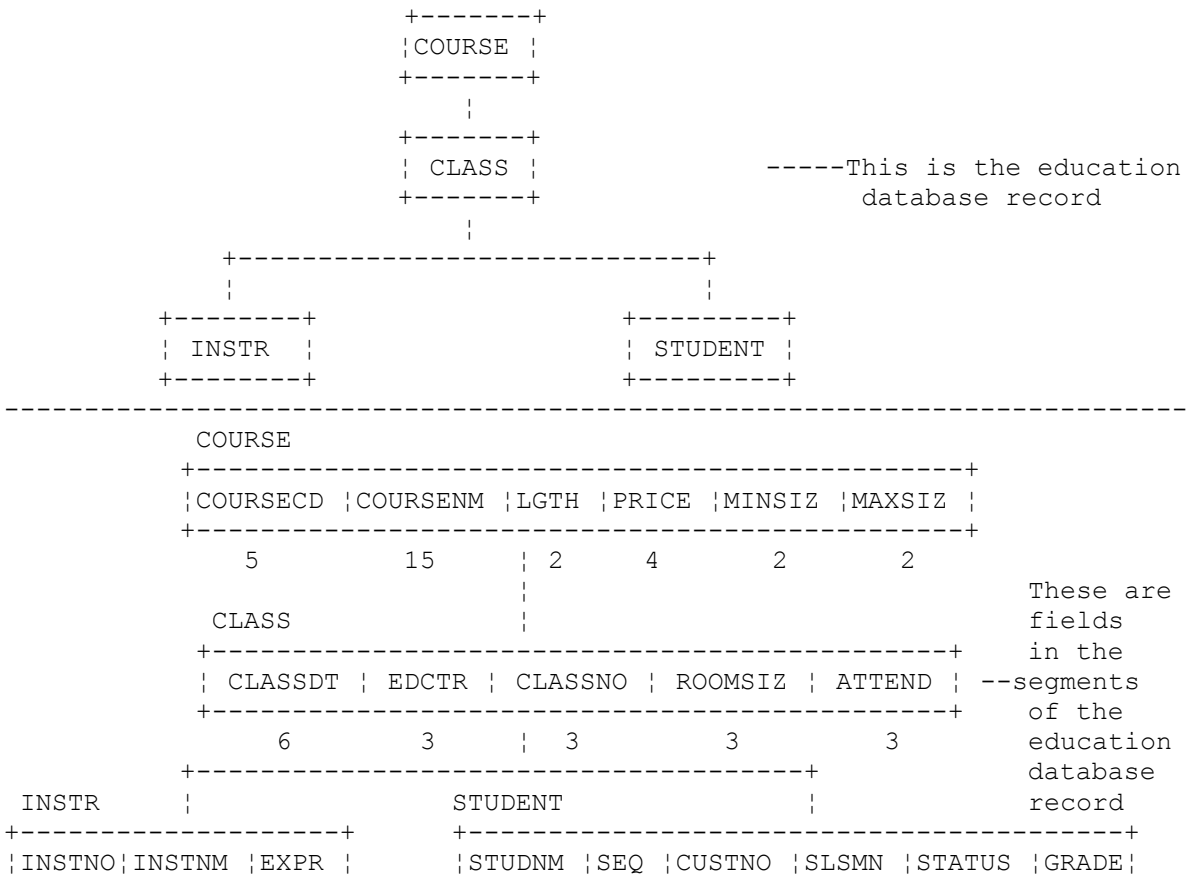
target segment. The optional fields are also copied from the source segment with one exception, which is discussed later.

Using the education database in Figure below, you can see how three segments work together. In this example, the education database is a HIDAM database that uses RBAs rather than symbolic pointers. Suppose an application program needs to access the education database by student name and then list all courses the student is taking:

- The segment the application is trying to retrieve is the COURSE segment, because the segment contains the names of courses (COURSENM field). Therefore, COURSE is the target segment, and needs retrieval.

- In this example, the application program is going to use the student's name in its DL/I call to retrieve the COURSE segment. The DL/I call is qualified using student name as its qualifier. The source segment contains the fields used to sequence the pointer segments in the secondary index. In this example, the pointer segments must be sequenced by student name. The STUDENT segment becomes the source segment. It is the fields in this segment that are copied into the data portion of the pointer segment as the key field.

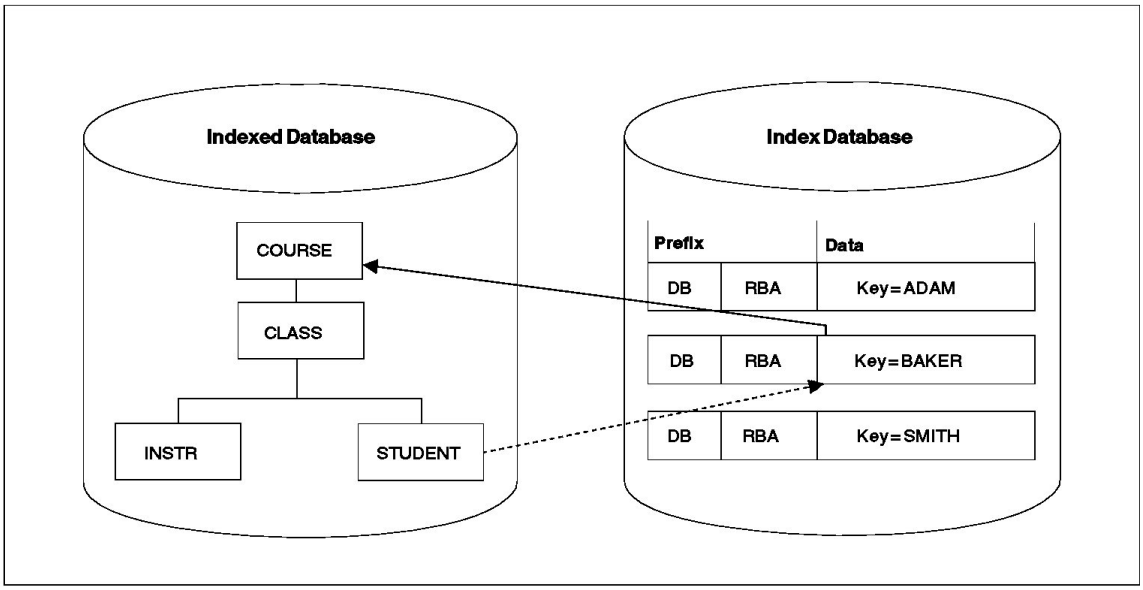
- The call from the application program invokes a search for a pointer segment with a key field that matches the student name. Once the correct pointer segment in the index is found, it contains the address of the COURSE segment the application program is trying to retrieve.



```
+--------------------+          +----------------------------------------+
   3       20      2              20      2      4       3       1      2
```

Education Database Record and the Fields in It

GU COURSE ␣␣ (XNAME␣␣␣ = ␣BAKER ␣ . . . ␣) ← This is the call the application program issues. XNAME is from the NAME= parameter on the XDFLD statement.



COURSE is the target segment the application program is trying to retrieve.

STUDENT is the source segment containing the one or more fields the application program uses as a qualifier in its call and that the data portion of a pointer segment contains as a key.

The BAKER segment in the secondary index is the pointer segment, whose prefix contains the address of the segment to be retrieved and whose data fields contain the key the application program uses as a qualifier in its call.

How a Segment Is Accessed Using a Secondary Index

How the Hierarchy Is Restructured

When the PROCSEQ= parameter in the PCB is coded (specifying that the application program needs to do processing using the secondary index), the way in which the application program perceives the database record changes.

If the target segment is the root segment in the database record, the structure the application program perceives does not differ from the one it can access using the regular processing

sequence. However, if the target segment is not the root segment, the hierarchy in the database record is conceptually restructured. Figure below illustrates this concept.

The target segment (as shown in the figure) is segment G. Target segment G becomes the root segment in the restructured hierarchy. All dependents of the target segment (segments H, J, and I) remain dependents of the target segment. However, all segments on which the target is dependent (segments D and A) and their subordinates become dependents of the target and are put in the leftmost positions of the restructured hierarchy. Their position in the restructured hierarchy is the order of immediate dependency. D becomes an immediate dependent of G, and A becomes an immediate dependent of D.

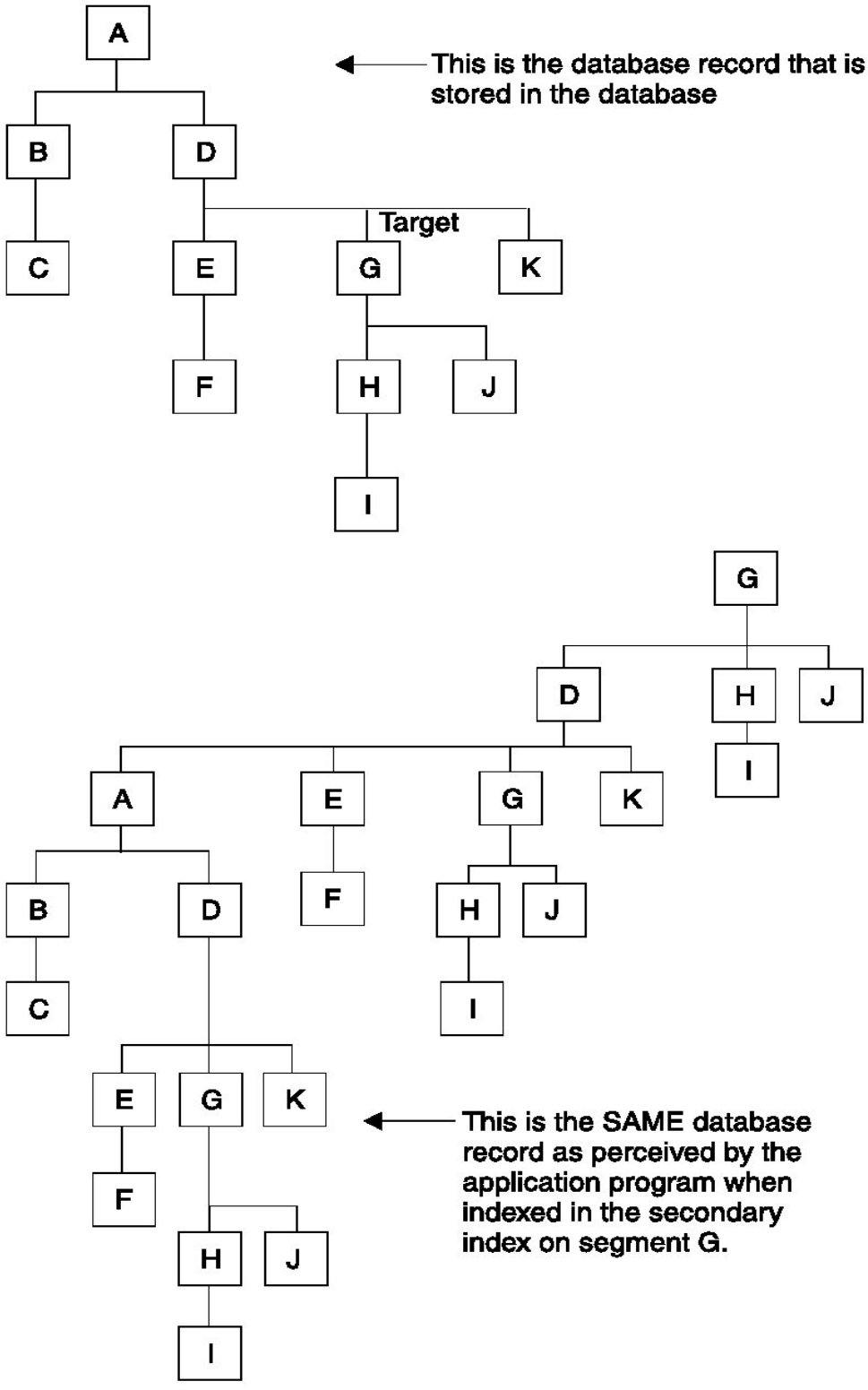
Secondary data structure

This new structure is called a secondary data structure. A processing restriction exists when using a secondary data structure, and the target segment and the segments on which it was dependent (its physical parents, segments D and A) cannot be inserted or deleted.

Secondary processing sequence

The restructuring of the hierarchy in the database record changes the way in which the application program accesses segments. The new sequence in which segments are accessed is called the secondary processing sequence. Figure below shows how the application program perceives the database record.

If the same segment is referenced more than once as shown in the Figure below, you must use the DBDGEN utility to generate a logical DBD that assigns alternate names to the additional segment references. If you do not generate the logical DBD, the PSBGEN utility will issue the message "SEG150" for the duplicate SENSEG names.



How the Hierarchy in a Database Record Is Restructured When Secondary Indexing Is Used

How a Secondary Index Is Stored

Secondary index databases contain root segments only. They are stored in a single VSAM KSDS if the key in the pointer segment is unique. If keys are not unique, an additional data set must be used (an ESDS) to store segments containing duplicate keys. (KSDS data sets do not allow duplicate keys.) Duplicate keys exist when, for example, a secondary index is used to retrieve courses based on student name. As shown in the following figure, several source segments could exist for each student:

```
¦           ¦                ¦
¦Prefix     ¦Data            ¦
+-----------------------------+
¦ DB  ¦ RBA ¦ MATH   ¦ ADAMS  ¦
+-----------------------------+

+-----------------------------+
¦ DB  ¦ RBA ¦ FRENCH ¦ ADAMS  ¦
+-----------------------------+

+-----------------------------+
¦ DB  ¦ RBA ¦ HIST   ¦ ADAMS  ¦
+-----------------------------+
```

```
Each pointer segment in a secondary index is stored in one logical record. A logical
record containing a pointer
segment looks like this:

----- Logical Record -------------_

          ---  Pointer Segment ---_

+--------------¦ ¦----------¦ ¦------+
¦ Pointer ¦   Prefix  ¦    Data      ¦
¦ Field   ¦           ¦              ¦
+--------------¦ ¦----------¦ ¦------+
```

The format of the logical record is the same in both a KSDS and ESDS data set. The pointer field at the beginning of the logical record exists only when the key in the data portion of the segment is not unique. If keys are not unique, some pointer segments will contain duplicate keys. These pointer segments must be chained together, and this is done using the pointer field at the beginning of the logical record.

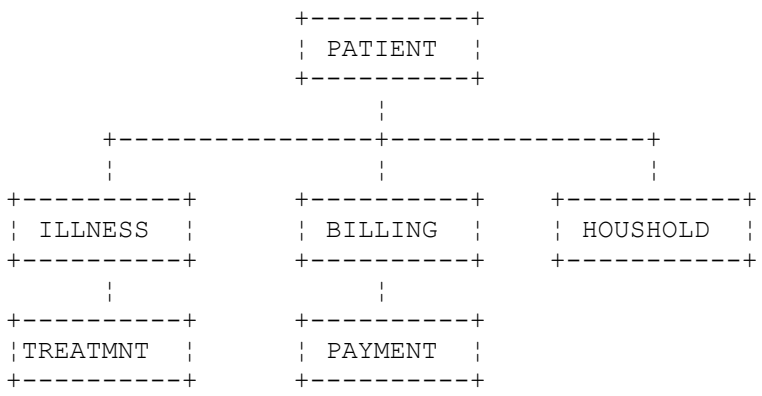
Pointer segments containing duplicate keys are stored in the ESDS in LIFO (last in, first out) sequence. When the first duplicate key segment is inserted, it is written to the ESDS, and the KSDS logical record containing the segment it is a duplicate of points to it. When the second duplicate is inserted, it is inserted into the ESDS in the next available location. The KSDS logical record is updated to point to the second duplicate. The effect of inserting duplicate pointer segments into the ESDS in LIFO sequence is that the original pointer segment (the one in the KSDS) is retrieved last. This retrieval sequence should not be a problem, because duplicates, by definition, have no special sequence.

Lab example Based On the Patient Database

See the Medical Data Base Description earlier in this material. Only the hierarchical structure is shown again for ready reference. The secondary index is built over the DOCTOR field in the TREATMNT segment. This gives us a view of the database from the perspective of a doctor.

Warning!!:-
The various steps in building the database with a secondary index must be followed in the same sequence. It is highly recommended that except for changing globally USER01 to your userid, no other names are changed as this could lead to errors and wastage of time in retracing the steps.

patient database



The DBD's for this database are as below.

PNDBHIIX (HIDAM root Index Database)

```
DBD   NAME=PNDBHIIX,ACCESS=INDEX
DATASET DD1=PNDBHIIX
SEGM NAME=INDXSEG,BYTES=5
LCHILD NAME=(PATIENT,PNDBHIX),INDEX=PATNO
FIELD NAME=(INDXSEQ,SEQ,U),BYTES=5,START=1,TYPE=C
DBDGEN
FINISH
END
```

PNDBHIX (HIDAM Database)

```
DBD   NAME=PNDBHIX,ACCESS=(HIDAM,VSAM)
DATASET DD1=PNDBHIX
SEGM NAME=PATIENT,BYTES=45,PARENT=0
LCHILD NAME=(INDXSEG,PNDBHIIX),PTR=INDX
FIELD NAME=(PATNO,SEQ,U),BYTES=5,START=1,TYPE=C
FIELD NAME=NAME,BYTES=10,START=6,TYPE=C
FIELD NAME=ADDR,BYTES=30,START=16,TYPE=C
SEGM  NAME=ILLNESS,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=(ILLDATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=ILLNAME,BYTES=10,START=9,TYPE=C
```

```
SEGM NAME=TREATMNT,BYTES=32,PTR=T,PARENT=((ILLNESS,SNGL))
LCHILD NAME=(DOCPTR,DOCDBD),POINTER=INDX
FIELD NAME=(DOCTOR,SEQ,M),BYTES=10,START=23,TYPE=C
XDFLD NAME=DOCNAME,SRCH=DOCTOR
FIELD NAME=DATE,BYTES=8,START=1,TYPE=C
FIELD NAME=MEDICINE,BYTES=10,START=9,TYPE=C
FIELD NAME=QUANTITY,BYTES=4,START=19,TYPE=C
SEGM  NAME=BILLING,BYTES=6,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=BILLING,BYTES=6,START=1,TYPE=C
SEGM  NAME=PAYMENT,BYTES=6,PTR=T,PARENT=((BILLING,SNGL))
FIELD NAME=PAYMENT,BYTES=6,START=1,TYPE=C
SEGM  NAME=HOUSHLD,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=RELNAME,BYTES=10,START=1,TYPE=C
FIELD NAME=RELATN,BYTES=8,START=11,TYPE=C
DBDGEN
FINISH
END
```

Notes:-
The LCHILD and XDFLD macros are coded after the SEGM statement for the target Segment.

If the XDFLD does not specify a NAME=source-segment, it is assumed that the source and target segments are the same.

The SRCH= can specify up to 5 fields from the source segment.

DOCBDB Secondary Index Database

```
DBD  NAME=DOCDBD,ACCESS=INDEX
DATASET DD1=DOCDBD1,OVFLW=DOCDBD2
SEGM NAME=DOCPTR,PARENT=0,BYTES=10
LCHILD NAME=(TREATMNT,PNDBHIX),INDEX=DOCNAME,PTR=SNGL
FIELD NAME=(INDXSEQ,SEQ,M),BYTES=10,START=1,TYPE=C
DBDGEN
FINISH
END
```

Notes:-
Code PTR=SNGL if we want a direct pointer. Code PTR=SYMB if we want a symbolic pointer (If the index is being built over a HISAM data base)

The PSB's are PTHISXL for load, PTHISXG for reads, and PTHISXA for Inserts.

PTHISXL

```
PCB  TYPE=DB,NAME=PNDBHIX,PROCOPT=LS,KEYLEN=23
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PTHISXL,LANG=PL/I
END
```

PTHISXA

```
PCB  TYPE=DB,NAME=PNDBHIX,PROCOPT=A,KEYLEN=23
SENSEG NAME=PATIENT,PARENT=0
```

```
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PTHISXA,LANG=PL/I
END
```

PTHISXG

```
PCB  TYPE=DB,NAME=PNDBHIX,PROCOPT=A,KEYLEN=23,PROCSEQ=DOCDBD
SENSEG NAME=TREATMNT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=TREATMNT
SENSEG NAME=PATIENT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PTHISXG,LANG=PL/I
END
```

STEP 1

Prepare the DBD's and PSB's above using the JCL shown in the section on DBD and PSB preparation JCL.

STEP 2

Run the PREREORG utility as shown below. Preserve the output data set USER01.RLCDS.

PREORG

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//     JCLLIB ORDER=(USER01.PROCLIB)
//STEP1   EXEC PREREORG
//SYSPRINT  DD SYSOUT=A,DCB=BLKSIZE=1200
//G.DFSURCDS  DD DSNAME=USER01.RLCDS,DISP=(NEW,KEEP),
//             UNIT=SYSDA,VOL=SER=IMSMSC,DCB=(BLKSIZE=1600),
//             SPACE=(CYL,1)
//G.SYSIN     DD *,DCB=BLKSIZE=80
DBIL=PNDBHIX
/*
//
```

The **PREREORG procedure** is shown below. Be sure to change USER01 to your userid!.

```
//       PROC
//G      EXEC PGM=DFSRRC00,REGION=0M,
//            PARM='ULU,DFSURPR0,,,,,,,,,,,N'
//STEPLIB  DD DSN=IMS.RESLIB,DISP=SHR
//         DD DSN=IMS.PGMLIB,DISP=SHR
//         DD DSN=USER01.LOADLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.RESLIB,DISP=SHR
//IMS      DD DSN=USER01.DBDLIB,DISP=SHR
```

STEP 3

Create the VSAM clusters prior to loading the database.
For the Secondary Index database clusters USER01.PNTDBIX1 and USER01.PNTDBIX2.
For the HIDAM database clusters USER01.PNTDBIX and USER01.PNTDBIIX.

IDCHDIX

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
   DELETE USER01.PNTDBIX CLUSTER
   DELETE USER01.PNTDBIIX CLUSTER
   DELETE USER01.PNTDBIX1 CLUSTER
   DELETE USER01.PNTDBIX2 CLUSTER
   DEFINE CLUSTER (NAME(USER01.PNTDBIX) NONINDEXED -
   RECORDSIZE(2041,2041) CONTROLINTERVALSIZE(2048) -
   TRACKS(2 2))
   DEFINE CLUSTER (NAME(USER01.PNTDBIIX) INDEXED KEYS(5,5) -
   RECORDSIZE(10,10) TRACKS(1,1)) DATA(CONTROLINTERVALSIZE(1024))
   DEFINE CLUSTER (NAME(USER01.PNTDBIX2) NONINDEXED -
   RECORDSIZE(20,20) CONTROLINTERVALSIZE(1024) -
   TRACKS(1 1))
   DEFINE CLUSTER (NAME(USER01.PNTDBIX1) INDEXED KEYS(10,9) -
   RECORDSIZE(20,20) TRACKS(1,1)) DATA(CONTROLINTERVALSIZE(1024))
//
```

STEP 4

Load the databases using PL/I program PLIPGM5 and PSB PTHISXL. A new data set is created against ddname DFSURWF1 with DSN of USER01.DFSURWF. This data set must be preserved for completing further load steps. The load JCL is shown below. The source for PLIPGM5 has been shown earlier.

IMSBATCH

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC DLIBATCH,MBR=PLIPGM5,PSB=PTHISXL,DBRC=N
//G.SYSPRINT DD SYSOUT=*
//*
//* Primary and Secondary Index data base DSN's
//G.PNDBHIX DD DSN=USER01.PNTDBIX,DISP=SHR
//G.PNDBHIIX DD DSN=USER01.PNTDBIIX,DISP=SHR
//G.DOCDBD1 DD DSN=USER01.PNTDBIX1,DISP=SHR
//G.DOCDBD2 DD DSN=USER01.PNTDBIX2,DISP=SHR
//*
//* Load data
//G.SYSIN DD DSN=USER01.LOAD.DATA(DATA1),DISP=SHR
//*
//G.DFSURWF1 DD DSN=USER01.DFSURWF,DISP=(NEW,KEEP),
//    DCB=(RECFM=VB,LRECL=900,BLKSIZE=1208),SPACE=(TRK,(1,1))
//G.DFSVSAMP DD *
VSRBF=2048,4
/*
//
```

DATA1

```
PATIENT   00001ABCDEF1   18,CHN 600023-1
ILLNESS   01012000MALARIA
TREATMNT  01012000QUININE   0004DR.DOBBS
```

```
BILLING   000600
PAYMENT   000600
HOUSHLD   MOHAN     FATHER
PATIENT   00002ABCDEF2   18,CHN 600023-2
ILLNESS   01012000JAUNDICE
TREATMNT  01012000AYURVEDIC 0004DR.JAMES
BILLING   000500
PAYMENT   000400
PAYMENT   000100
HOUSHLD   MEERA     MOTHER
PATIENT   00003ABCDEF3   18,CHN 600023-3
ILLNESS   01012000FLU
TREATMNT  01012000CROCIN    0004DR.PILOO
BILLING   000400
PAYMENT   000400
HOUSHLD   JAYA      SISTER
PATIENT   00004ABCDEF4   18,CHN 600023-4
ILLNESS   01012000MEASLES
TREATMNT  01012000NEEMLEAVES0004DR.TOM
BILLING   000300
PAYMENT   000200
PAYMENT   000100
HOUSHLD   MAYA      SISTER
PATIENT   00005ABCDEF5   18,CHN 600023-5
ILLNESS   01012000TYPHOID
TREATMNT  01012000ANTIBIOTIC0004DR.YOUNG
BILLING   000200
PAYMENT   000200
HOUSHLD   LATA      SISTER
```

STEP 5

Run the PREXFGEN utility.

The outputs of this step are data sets ddname DFSURWF3 and a DSN of USER01.WF3 and ddname DFSURIDX and a DSN of USER01.DFSURIDX which must be kept for the next step.

The input data sets are as below.

```
DFSURCDS   DSN=USER01.RLCDS   (Control data set from PREREORG)
PNDBHIX    DSN=USER01.PNTDBIX
PNDBHIIX   DSN=USER01.PNTDBIIX
DOCDBD1    DSN=USER01.PNTDBIX1
DOCDBD2    DSN=USER01.PNTDBIX2
```

PREFXGEN

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//PREFXRES EXEC PGM=DFSURG10
//STEPLIB  DD DSN=IMS.RESLIB,DISP=SHR
//SYSUDUMP DD SYSOUT=A
//SYSPRINT DD SYSOUT=A,DCB=BLKSIZE=1200
//SYSOUT   DD SYSOUT=A
//SORTLIB  DD DSN=SYS1.SORTLIB,DISP=SHR
//SORTWK01 DD  UNIT=SYSDA,SPACE=(1008,(60),,CONTIG)
//SORTWK02 DD  UNIT=SYSDA,SPACE=(1008,(60),,CONTIG)
```

```
//SORTWK03 DD  UNIT=SYSDA,SPACE=(1008,(60),,CONTIG)
//SORTIN   DD DSN=USER01.DFSURWF,DISP=SHR
//DFSURWF2 DD DSN=USER01.WF2,UNIT=SYSDA,SPACE=(1008,(30),,CONTIG),
//         DISP=(NEW,KEEP),DCB=(RECFM=VB,LRECL=900,BLKSIZE=1008)
//DFSURWF3 DD DSN=USER01.WF3,UNIT=SYSDA,SPACE=(1008,(30),,CONTIG),
//         DISP=(NEW,KEEP),DCB=(RECFM=VB,LRECL=900,BLKSIZE=1008)
//DFSURIDX DD DSN=USER01.DFSURIDX,UNIT=SYSDA,SPACE=(1008,(30),,CONTIG),
//         DISP=(NEW,KEEP),DCB=(RECFM=VB,LRECL=900,BLKSIZE=1008)
//DFSURCDS DD DSN=USER01.RLCDS,DISP=(SHR)
//PNDBHIX DD DSN=USER01.PNTDBIX,DISP=SHR
//PNDBHIIX DD DSN=USER01.PNTDBIIX,DISP=SHR
//DOCDBD1 DD DSN=USER01.PNTDBIX1,DISP=SHR
//DOCDBD2 DD DSN=USER01.PNTDBIX2,DISP=SHR
/*
```

STEP 6

Unload the database using the HISAM unload utility.
Input data set is USER01.DFSURIDX which is generated by the PREFXGEN utility.
Output data set is USER01.DBXOUT1 which is used as input to the HISAM reload utility.

HISUNLD

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//STEP1   EXEC PGM=DFSRRC00,PARM='ULU,DFSURUL0,,,,,,,,,,,N'
//STEPLIB   DD DSNAME=IMS.RESLIB,DISP=SHR
//DFSRESLB  DD DSNAME=IMS.RESLIB,DISP=SHR
//IMS       DD DSNAME=USER01.DBDLIB,DISP=SHR
//SYSPRINT  DD SYSOUT=A,DCB=BLKSIZE=1330
//DOCDBD1 DD DSN=USER01.PNTDBIX1,DISP=SHR
//DOCDBD2 DD DSN=USER01.PNTDBIX2,DISP=SHR
//DBXOUT1   DD DSNAME=USER01.DBXOUT1,DISP=(NEW,KEEP),
//             UNIT=SYSDA,SPACE=(TRK,(1,1))
//NDXWDS1  DD DSN=USER01.DFSURIDX,DISP=SHR
//* +----1----+----2----+----3----+----4----+----5----+----6----+----7
//SYSIN     DD *
OPTIONS=(STATS,ABEND)
X1MDOCDBD   DOCDBD1  DBXOUT1           NDXWDS1
/*
//
```

STEP 7

Reload the database using the HISAM reload utility.
The input data set is USER01.DBXOUT1 from the HISAM unload step.

HISRELOD

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//STEP1   EXEC PGM=DFSRRC00,PARM='ULU,DFSURRL0,,,,,,,,,,,N'
//STEPLIB   DD DSNAME=IMS.RESLIB,DISP=SHR
//DFSRESLB  DD DSNAME=IMS.RESLIB,DISP=SHR
//IMS       DD DSNAME=USER01.DBDLIB,DISP=SHR
//SYSPRINT  DD SYSOUT=A,DCB=BLKSIZE=1330
//DOCDBD1 DD DSN=USER01.PNTDBIX1,DISP=SHR
//DOCDBD2 DD DSN=USER01.PNTDBIX2,DISP=SHR
//DFSUIN01  DD DSNAME=USER01.DBXOUT1,DISP=SHR
//SYSUDUMP DD SYSOUT=*
//DFSVSAMP  DD *
VSRBF=2048,4
```

```
/*
//SYSIN DD *
OPTIONS=ABEND
/*
//
```

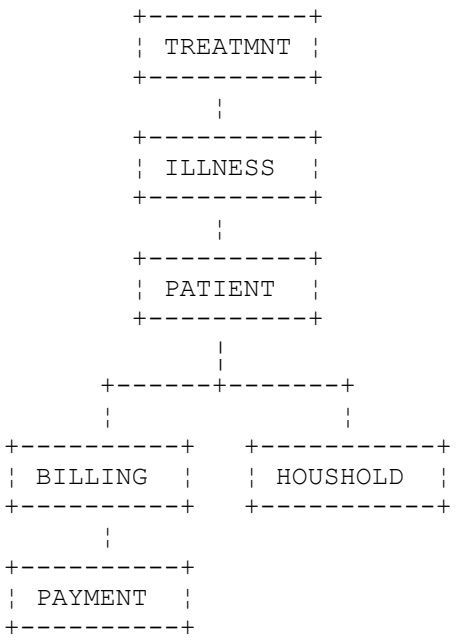
STEP 7

Test the secondary index by accessing the database using the PSB below.

PTHISXG

```
PCB  TYPE=DB,NAME=PNDBHIX,PROCOPT=A,KEYLEN=23,PROCSEQ=DOCDBD
SENSEG NAME=TREATMNT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=TREATMNT
SENSEG NAME=PATIENT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PTHISXG,LANG=PL/I
END
```

You should be able to get segments in hierarchic sequence in the inverted hierarchy below.



Use PLIPGM4 to dump the database. The source for the PL/I program has been provided earlier.

You can also try out inserting new data base records in the data base and check that the secondary index is automatically maintained by IMS. Use PLIPGM5 described earlier and PSB PTHISXA to insert the new data base records. Dump the data base in hierarchic sequence using PLIPGM4 and PTHISXG.

New Data

DATA2

```
PATIENT   00007ABCDEF7   18,CHN 600023-1
```

```
ILLNESS   01012000CANCER
TREATMNT  01012000NOTHING   0004DR.HOPE
BILLING   000600
PAYMENT   000600
HOUSHLD   RAM       FATHER
```

DATA3

```
PATIENT   00006ABCDEF6   18,CHN 600023-1
ILLNESS   01012000CANCER
TREATMNT  01012000NOTHING   0004DR.FLURY
BILLING   000600
PAYMENT   000600
HOUSHLD   RAM       FATHER
```

Logical Relationships

Index

With logical relationships, you can give an application program access to a data structure that contains segments from more than one physical database.

An alternative to using logical relationships to resolve the different needs of applications is to create separate databases or carry duplicate data in a single database. However, in both cases this creates duplicate data. Avoid duplicate data because:

- Extra maintenance is required when duplicate data exists. because both sets of data must be kept up to date. In addition, updates must be done simultaneously to maintain data consistency.

- Extra space is required on DASD to hold duplicate data.

By establishing a path between two segment types, logical relationships eliminate the need to store duplicate data. To establish a logical relationship, three segment types are always defined:

1. A physical parent
2. A logical parent
3. A logical child

For example, suppose two databases exist, one for orders that a customer has placed and one for items that can be ordered. The first database is called the ORDER database; the second is called the ITEM database.

The ORDER database contains data such as:

Order number
Customer's name and address
Type of items ordered
Quantity of each item ordered
Delivery data

The ITEM database contains data such as:

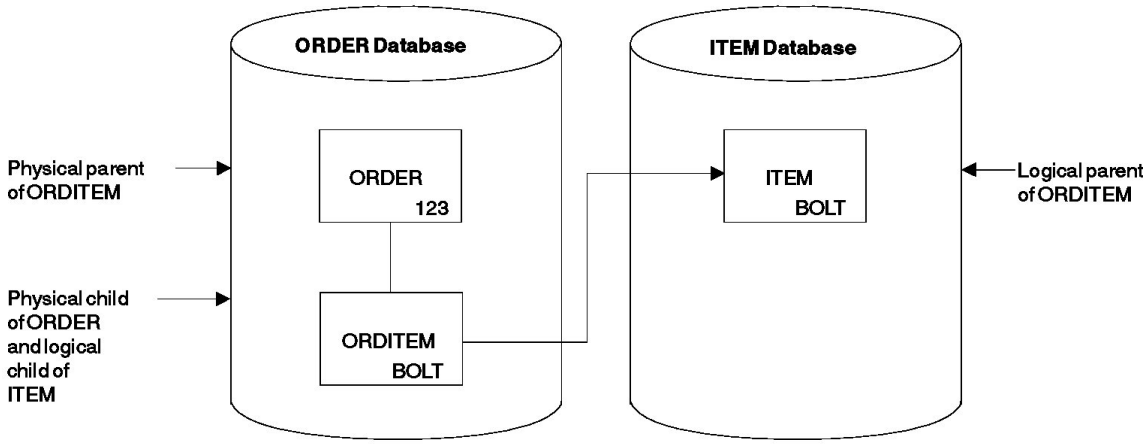
Type of items that can be ordered
Quantity of each item in stock
Quantity of each item in stock that has been ordered but not yet delivered

Defining a logical relationship

If an application program needs data from both databases, this can be done by defining a logical relationship between the two databases. As shown in Figure below, a path can be established

between the ORDER and ITEM databases using a segment type, called a logical child segment, that points into the ITEM database. Figure below is a simple implementation of a logical relationship. In this case, ORDER is the physical parent of ORDITEM. ORDITEM is the physical child of ORDER but the logical child of ITEM.

In a logical relationship, there is a logical parent segment type and it is the segment type pointed to by the logical child. In this example, ITEM is the logical parent of ORDITEM. ORDITEM establishes the path or connection between the two segment types. If an application program now enters the ORDER database, it can access data in the ITEM database by following the pointer in the logical child segment from the ORDER to the ITEM database.



A Simple Logical Relationship

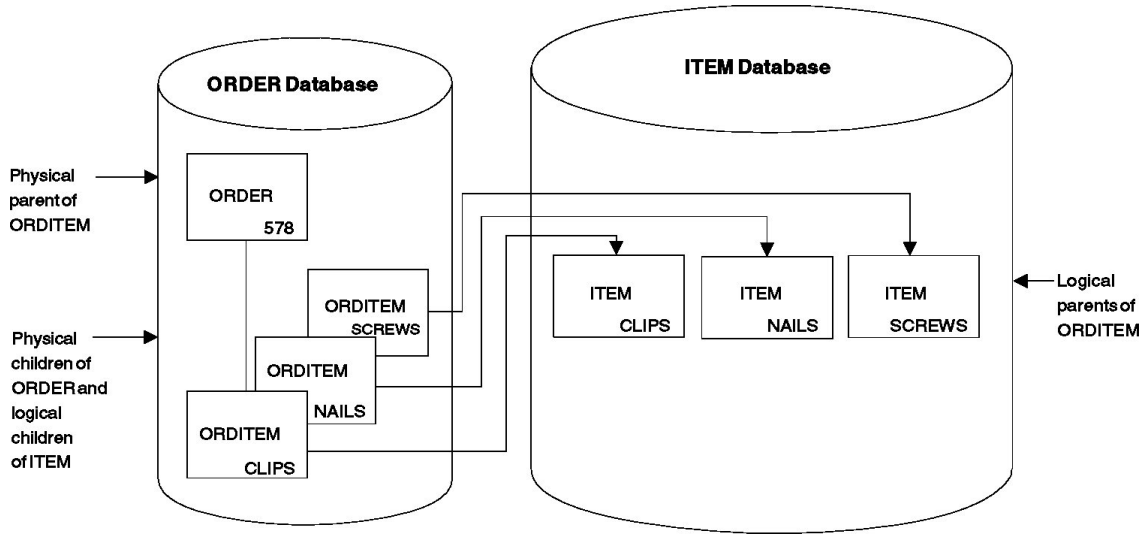
The physical parent and logical parent are the two segment types between which the path is established. The logical child is the segment type that establishes the path. The path established by the logical child is created using pointers.

There are three ways in which a logical relationship can be established or implemented. These methods of implementation are as follows:

Unidirectional logical relationship
Bi-directional physically paired logical relationship
Bi-directional virtually paired logical relationship

Unidirectional Logical Relationships

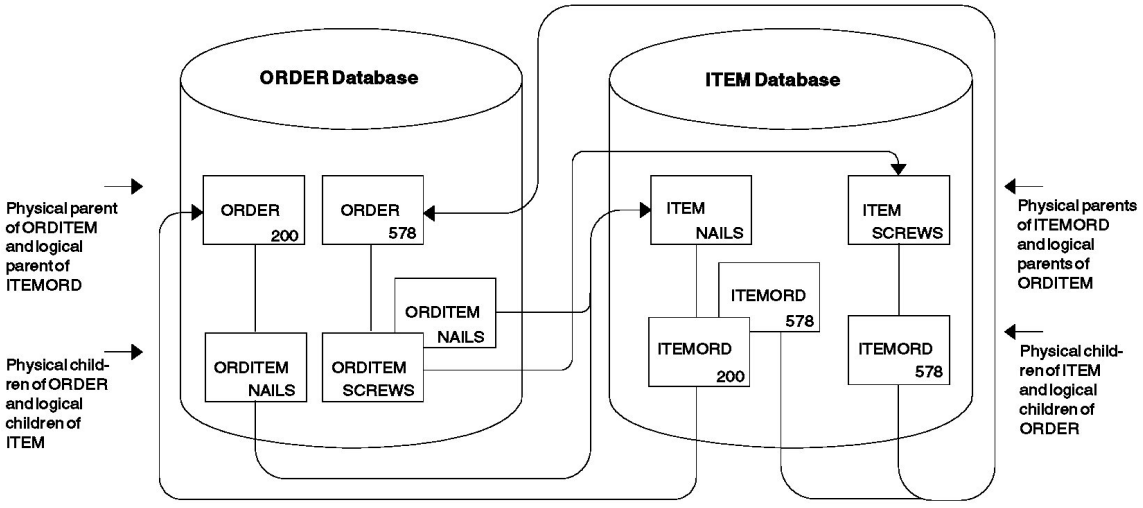
A unidirectional relationship links two segment types, a logical child and its logical parent, in one direction. A one-way path is established using a pointer in the logical child. Figure below shows a unidirectional relationship that has been established between the ORDER and ITEM databases. A unidirectional relationship can be established between two segment types in the same or different databases. Typically, however, a unidirectional relationship is created between two segment types in different databases. In the figure, the logical relationship can be used to cross from the ORDER to the ITEM database. It cannot be used to cross from the ITEM to the ORDER database, because the ITEM segment does not point to the ORDER database.



Unidirectional Logical Relationship

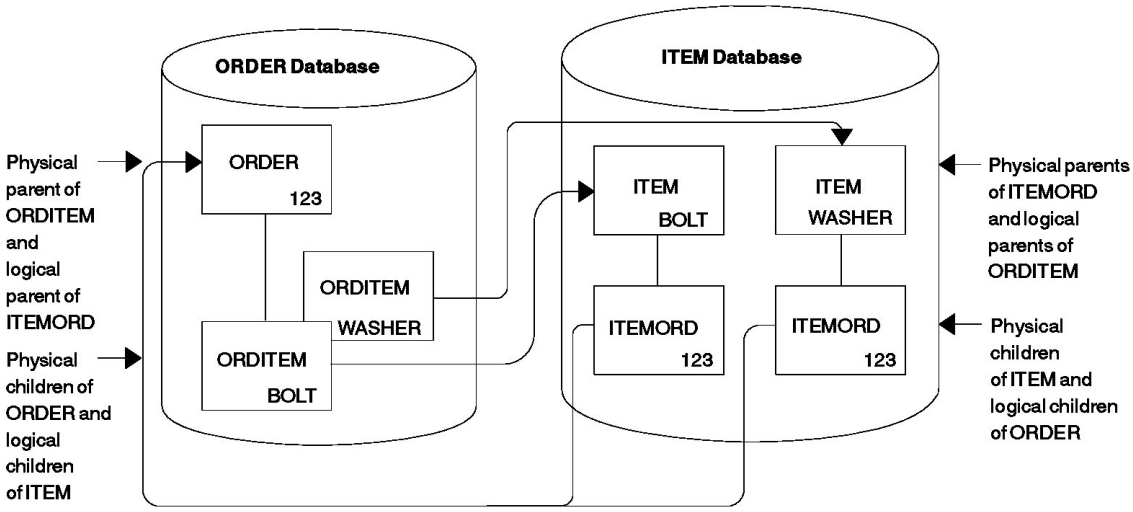
Unidirectional Logical relationship

It is possible to establish two unidirectional relationships, as shown in Figure below. Then either physical database can be entered and the logical child in either can be used to cross to the other physical database. However, IMS treats each unidirectional relationship as a one-way path. It does not maintain data on both paths. If data in one database is inserted, deleted, or replaced, the corresponding data in the other database is not updated. If, for example, DL/I replaces ORDITEM-SCREWS under ORDER-578, ITEMORD-578 under ITEM-SCREWS is not replaced. This maintenance problem does not exist in both bi-directional physically paired-logical and bi-directional virtually paired-logical relationships. Both relationship types are discussed next. IMS allows either physical database to be entered and updated and automatically updates the corresponding data in the other database.



Two Unidirectional relationships

Bidirectional Physically Paired Logical Relationship



Bi-directional Physically Paired Logical Relationship

A bi-directional physically paired relationship links two segment types, a logical child and its logical parent, in two directions. A two-way path is established using pointers in the logical child segments. Figure below shows a bi-directional physically paired logical relationship that has been established between the ORDER and ITEM databases.

Like the other types of logical relationships, a physically paired relationship can be established between two segment types in the same or different databases. The relationship shown in Figure allows either the ORDER or the ITEM database to be entered. When either database is entered, a path exists using the logical child to cross from one database to the other.

In a physically paired relationship, a logical child is stored in both databases. However, if the logical child has dependents, they are only stored in one database. For example, IMS maintains data in both paths in physically paired relationships. In Figure above if ORDER 123 is deleted from the ORDER database, IMS deletes from the ITEM database all ITEMORD segments that point to the ORDER 123 segment. If data is changed in a logical child segment, IMS changes the data in its paired logical child segment. Or if a logical child segment is inserted into one database, IMS inserts a paired logical child segment into the other database.

With physical pairing, the logical child is duplicate data, so there is some increase in storage requirements. In addition, there is some extra maintenance required because IMS maintains

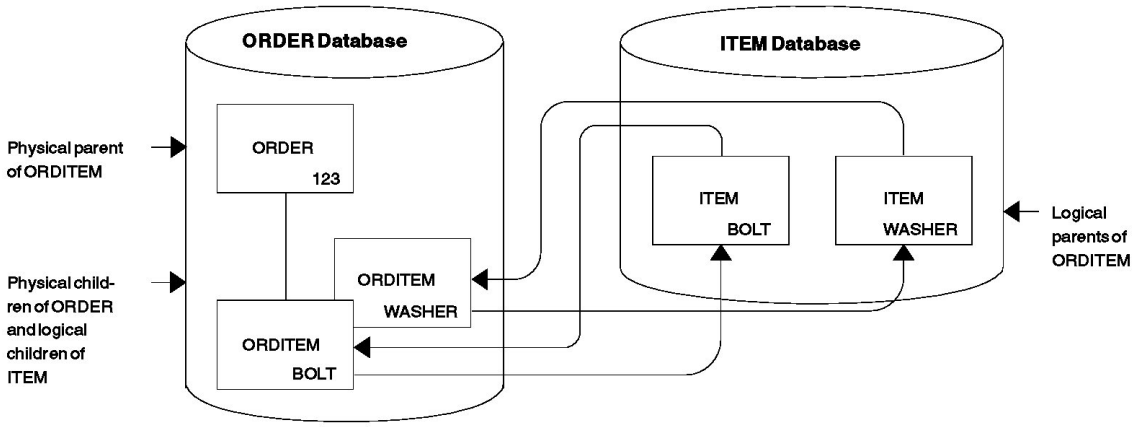
data on two paths. In the next type of logical relationship examined, this extra space and maintenance do not exist, however, IMS still allows you to enter either database. IMS also performs the maintenance for you.

Bi-directional Virtually Paired Logical Relationship

A bi-directional virtually paired relationship is like a bi-directional physically paired relationship in that:

- It links two segment types, a logical child and its logical parent, in two directions, establishing a two-way path.
- It can be established between two segment types in the same or different databases.

Figure below shows a bi-directional virtually paired relationship between the ORDER and ITEM databases. Notice that although there is a two-way path, a logical child segment exists only in the ORDER database. Going from the ORDER to the ITEM database, IMS uses the pointer in the logical child segment. Going from the ITEM to the ORDER database, IMS uses the pointer in the logical parent, as well as the pointer in the logical child segment.



bi-directionally Virtually Paired Logical Relationship

To define a virtually paired relationship, two logical child segment types are defined in the physical databases involved in the logical relationship. Only one logical child is actually placed in storage. The logical child defined and put in storage is called the real logical child. The logical child defined but not put in storage is called the virtual logical child.

Pointing and Pointers in Logical Relationships

In all logical relationships the logical child establishes a path between two segment types. The path is established by use of pointers. The following sections look at pointing in logical relationships and the various types of pointers that can be used. Four types of pointers can be specified for logical relationships:

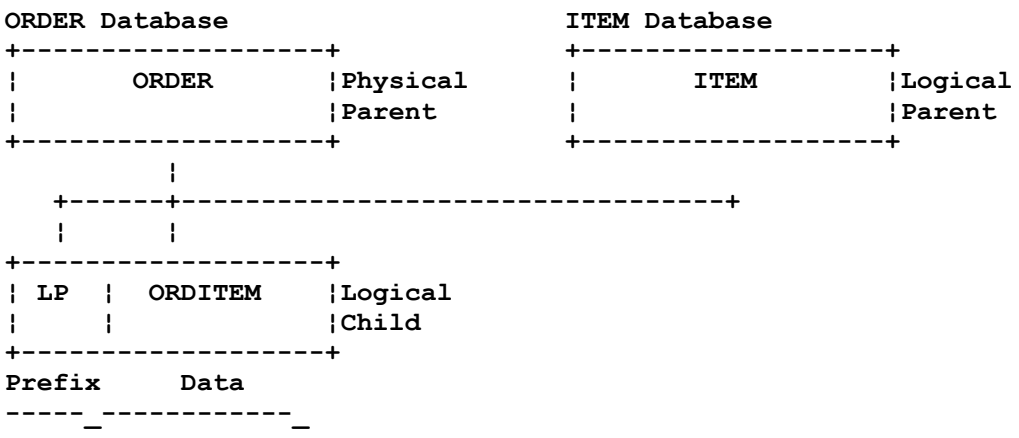
Logical parent pointer
Logical child pointer

Physical parent pointer
Logical twin pointer

Logical Parent Pointer

The pointer from the logical child to its logical parent is called a logical parent (LP) pointer. This pointer must be a symbolic pointer when it is pointing into a HISAM database. It can be either a direct or a symbolic pointer when it is pointing into an HDAM or HIDAM database.

A direct pointer consists of the direct address of the segment being pointed to, and it can only be used to point into a database where a segment, once stored, is not moved. This means the logical parent segment must be in an HD (HDAM and HIDAM) database, since the logical child points to the logical parent segment. The logical child segment, which contains the pointer, can be in a HISAM or an HD database. A direct LP pointer is stored in the logical child's prefix, along with any other pointers, and is four bytes long. Figure below shows the use of a direct LP pointer. In a HISAM database, pointers are not required between segments because they are stored physically adjacent to each other in hierarchic sequence. Therefore, the only time direct pointers will exist in a HISAM database is when there is a logical relationship using direct pointers pointing into an HD database.



Direct Logical Parent (LP) Pointer

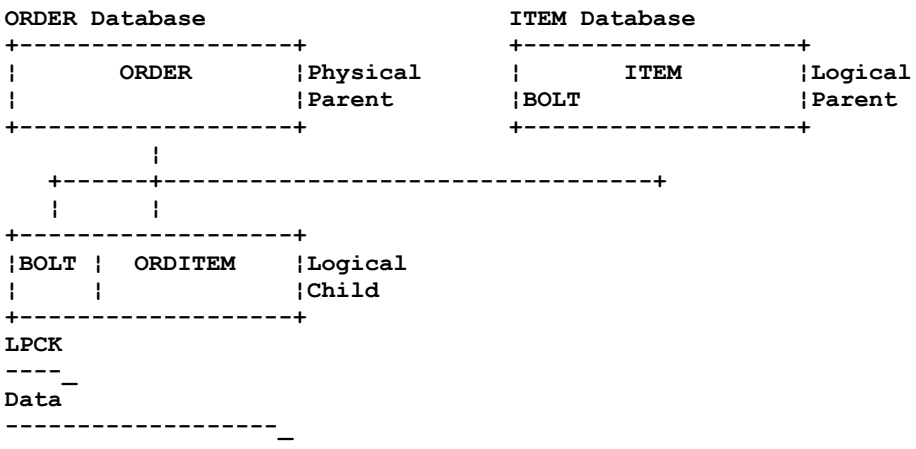
In Figure above, the direct LP pointer points from the logical child ORDITEM to the logical parent ITEM. Because it is direct, the LP pointer can only point to an HD database. However, the LP pointer can "exist" in a HISAM or an HD database. The LP pointer is in the prefix of the logical child and consists of the 4-byte direct address of the logical parent.

A symbolic LP pointer, which consists of the logical parent's concatenated key (LPCK), can be used to point into a HISAM or HD database. The Figure below illustrates how to use a symbolic LP pointer. The logical child ORDITEM points to the ITEM segment for BOLT. BOLT is therefore stored in ORDITEM in the LPCK. A symbolic LP pointer is stored in the first part of the data portion in the logical child segment.

The LPCK part of the logical child segment is considered non replaceable and you should not change it in your program. However IMS does not verify whether you have accidentally changed it, and if you do the REPL call doe not fail.

With symbolic pointers, if the database the logical parent is in is HISAM or HIDAM, IMS uses the symbolic pointer to access the index to find the correct logical parent segment. If the database the logical parent is in is HDAM, the symbolic pointer must be changed by the randomizing module into a block and RAP address to find the logical parent segment. IMS accesses a logical parent faster when direct pointing is used.

Although the figures show the LP pointer in a unidirectional relationship, it works exactly the same way in all three types of logical relationships.



Symbolic Logical Parent (LP) Pointer

In the Figure above, the symbolic LP pointer points from the logical child ORDITEM to the logical parent ITEM. With symbolic pointing, the ORDER and ITEM databases can be either HISAM or HD. The LPCK, which is in the first part of the data portion of the logical child, functions as a pointer from the logical child to the logical parent, and is the pointer used in the logical child.

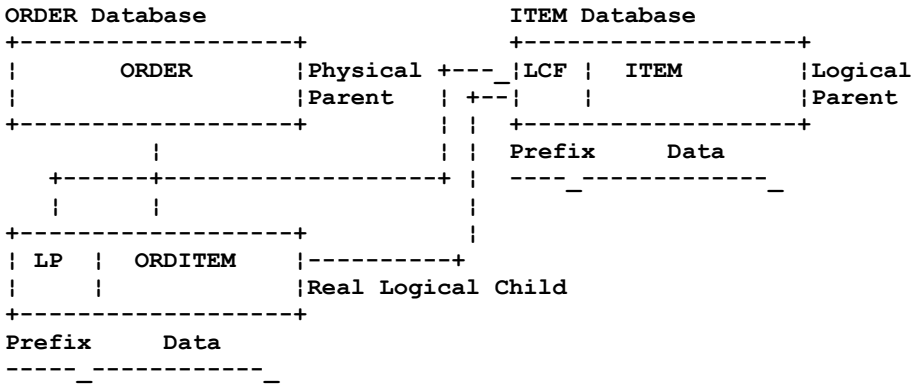
Logical Child Pointer

Logical child pointers are only used in logical relationships with virtual pairing. When virtual pairing is used, there is only one logical child on DASD, called the real logical child. This logical child has an LP pointer. The LP pointer can be symbolic or direct. In the ORDER and ITEM databases you have seen, the LP pointer allows you to go from the database containing the logical child to the database containing the logical parent. To enter either database and cross to the other with virtual pairing, you use a logical child pointer in the logical parent. Two types of logical child pointers can be used:

- Logical child first (LCF) pointers, or
- The combination of logical child first (LCF) and logical child last (LCL) pointers.

The LCF pointer points from a logical parent to the first occurrence of each of its logical child types. The LCL pointer points to the last occurrence of the logical child segment type for which it is specified. A LCL pointer can only be specified in conjunction with a LCF pointer. Figure below shows the use of the LCF pointer. These pointers allow you to cross from the ITEM database to the logical child ORDITEM in the ORDER database. However, although you are able to cross databases using the logical child pointer, you have only gone from ITEM to the logical child ORDITEM. To go to the ORDER segment, use the physical parent pointer explained in the next section.

LCF and LCL pointers are direct pointers. They contain the 4-byte direct address of the segment to which they point. This means the logical child segment, the segment being pointed to, must be in an HD database. The logical parent can be in a HISAM or HD database. If the logical parent is in a HISAM database, the logical child segment must point to it using a symbolic pointer. LCF and LCL pointers are stored in the logical parent's prefix, along with any other pointers. Figure below shows a LCF pointer.



Logical Child First (LCF) Pointer (Used in Virtual Pairing Only)

In the Figure above, the LCF pointer points from the logical parent ITEM to the logical child ORDITEM. Because it is a direct pointer, it can only point to an HD database, although, it can exist in a HISAM or an HD database. The LCF pointer is in the prefix of the logical parent and consists of the 4-byte RBA of the logical child.

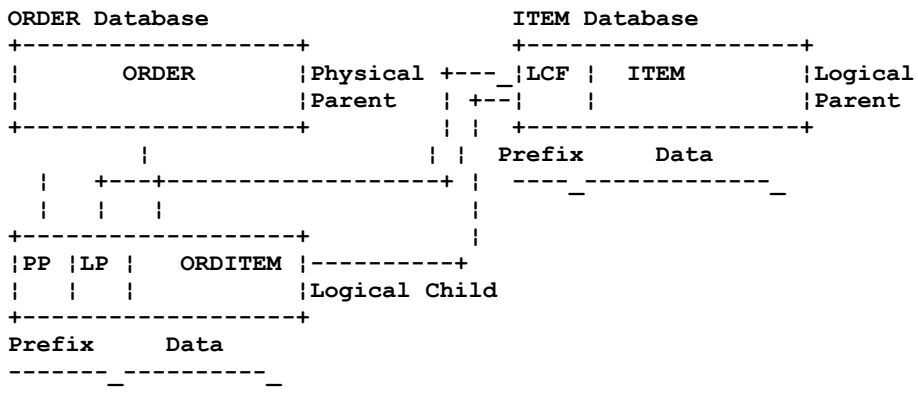
Physical Parent Pointer

Physical parent (PP) pointers point from a segment to its physical parent. They are generated automatically by IMS for all HD databases involved in logical relationships. PP pointers are put in the prefix of all logical child and logical parent segments. They are also put in the prefix of all segments on which a logical child or logical parent segment is dependent in its physical database. This creates a path from a logical child or its logical parent back up to the root segment on which it is dependent. Because all segments on which a logical child or logical parent is dependent are chained together with PP pointers to a root, access to these segments is possible in reverse of the usual order.

In the previous Figure, you saw that you could cross from the ITEM to the ORDER database when virtual pairing was used, and this was done using logical child pointers. However, the logical child pointer only got you from ITEM to the logical child ORDITEM. Figure below

shows how to get to ORDER. The PP pointer in ORDITEM points to its physical parent ORDER. If ORDER and ITEM are in an HD database but are not root segments, they (and all other segments in the path of the root) would also contain PP pointers to their physical parents.

PP pointers are direct pointers. They contain the 4-byte direct address of the segment to which they point. PP pointers are stored in a logical child or logical parent's prefix, along with any other pointers.



Physical Parent (PP) Pointer

In the Figure above, the PP pointer points from the logical child ORDITEM to its physical parent ORDER. It is generated automatically by IMS for all logical child and logical parent segments in HD databases. In addition, it is in the prefix of the segment that contains it and consists of the 4-byte direct address of its physical parent. PP pointers are generated in all segments from the logical child or logical parent back up to the root.

Logical Twin Pointer

Logical twin pointers are used only in logical relationships with virtual pairing. Logical twins are multiple logical child segments that point to the same occurrence of a logical parent. Two types of logical twin pointers can be used:

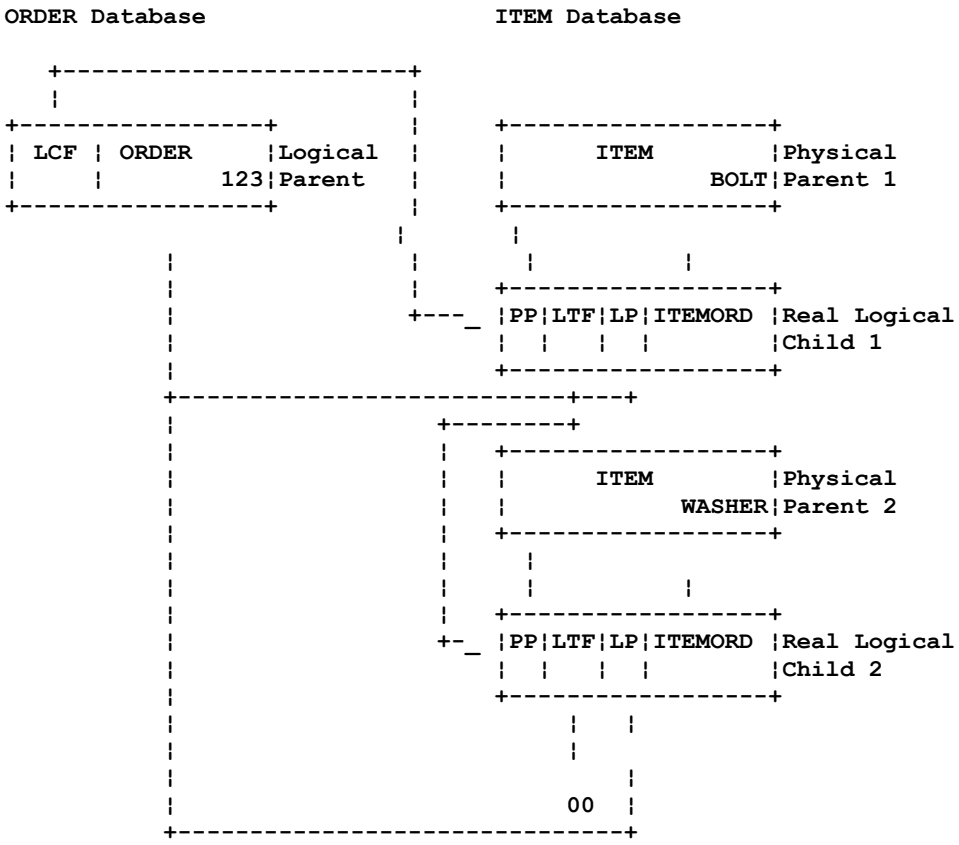
- Logical twin forward (LTF) pointers, or
- The combination of logical twin forward (LTF) and logical twin backward (LTB) pointers.

An LTF pointer points from a specific logical twin to the logical twin stored after it. An LTB pointer can only be specified in conjunction with an LTF pointer. When specified, an LTB points from a given logical twin to the logical twin stored before it. Logical twin pointers work in a similar way to the physical twin pointers used in HD databases. As with physical twin backward pointers, LTB pointers improve performance on delete operations. They do this when the delete that causes DASD space release is a delete from the physical access path. Similarly, PTB pointers improve performance when the delete that causes DASD space release is a delete from the logical access path.

Figure below shows use of the LTF pointer. In this example, ORDER 123 has two items: bolt and washer. The ITEMORD segments beneath the two ITEM segments use LTF pointers. If the ORDER database is entered, it can be crossed to the ITEMORD segment for bolts in the ITEM database. Then, to retrieve all items for ORDER 123, the LTF pointers in the ITEMORD segment can be followed. In the Figure only one other ITEMORD segment exists, and it is for washers. The LTF pointer in this segment, because it is the last twin in the chain, contains zeros.

LTB pointers on dependent segments improve performance when deleting a real logical child in a virtually paired logical relationship. This improvement occurs when the delete is along the physical path.

LTF and LTB pointers are direct pointers. They contain the 4-byte direct address of the segment to which they point. This means LTF and LTB pointers can only exist in HD databases. Figure below shows a LTF pointer.

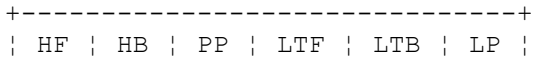


Logical Twin Forward (LTF) Pointer (Used in Virtual Pairing Only)

In the Figure above, the LTF pointer points from a specific logical twin to the logical twin stored after it. In this example, it points from the ITEMORD segment for bolts to the ITEMORD segment for washers. Because it is a direct pointer, the LTF pointer can only point to an HD database. The LTF pointer is in the prefix of a logical child pointer and consists of the 4-byte RBA of the logical twin stored after it.

Sequence of Pointers in a Segment's Prefix

When a segment contains more than one type of pointer and is involved in a logical relationship, pointers are put in the segment's prefix in the following sequence:



```
+------------------------------+
             or
+--------------------------------------------+
| TF | TB | PP | LTF | LTB | LP | PCF | PCL |
+--------------------------------------------+
```

Multiple PCF and PCL pointers can exist in a segment type, however, more than one of the other types of pointers can not.

Counter Used in Logical Relationships

IMS puts a 4-byte counter in all logical parents that do not have logical child pointers. The counter is stored in the logical parent's prefix and contains a count of the number of logical children pointing to this logical parent. The counter is maintained by IMS and is used to handle delete operations properly. If the count is greater than zero, the logical parent cannot be deleted from the database because there are still logical children pointing to it.

Intersection Data

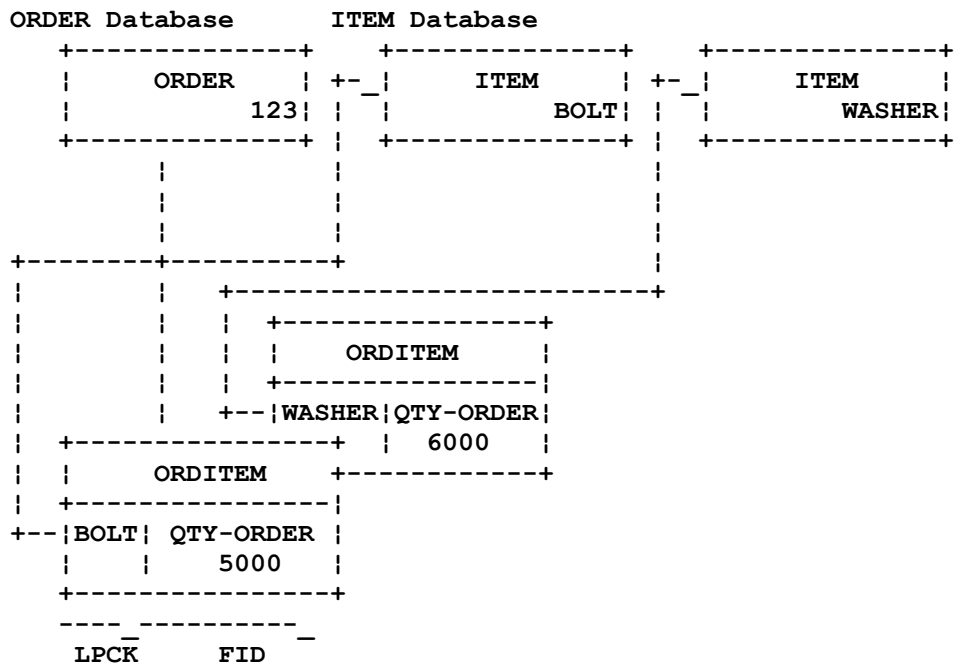
When two segments are logically related, data can exist that is unique to only that relationship. In Figure below, for example, one of the items ordered in ORDER 123 is 5000 bolts. The quantity 5000 is specific to this order (ORDER 123) and this item (bolts). It does not belong to either the order or item on its own. Similarly, in ORDER 123, 6000 washers are ordered. Again, this data is concerned only with that particular order and item combination.

This type of data is called intersection data, since it has meaning only for the specific logical relationship. The quantity of an item could not be stored in the ORDER 123 segment, because different quantities are ordered for each item in ORDER 123. Nor could it be stored in the ITEM segment, because for each item there can be several orders, each requesting a different quantity. Because the logical child segment links the ORDER and ITEM segments together, data that is unique to the relationship between the two segments can be stored in the logical child.

The two types of intersection data are: fixed intersection data (FID) and variable intersection data (VID).

Fixed intersection data

Data stored in the logical child is called fixed intersection data (FID). When symbolic pointing is used, it is stored in the data part of the logical child after the LPCK. When direct pointing is used, it is the only data in the logical child segment. Because symbolic pointing is used in Figure below, BOLT and WASHER are the LPCK, and the 5000 and 6000 are the FID. The FID can consist of several fields, all of them residing in the logical child segment.



Fixed Intersection Data

Variable intersection data

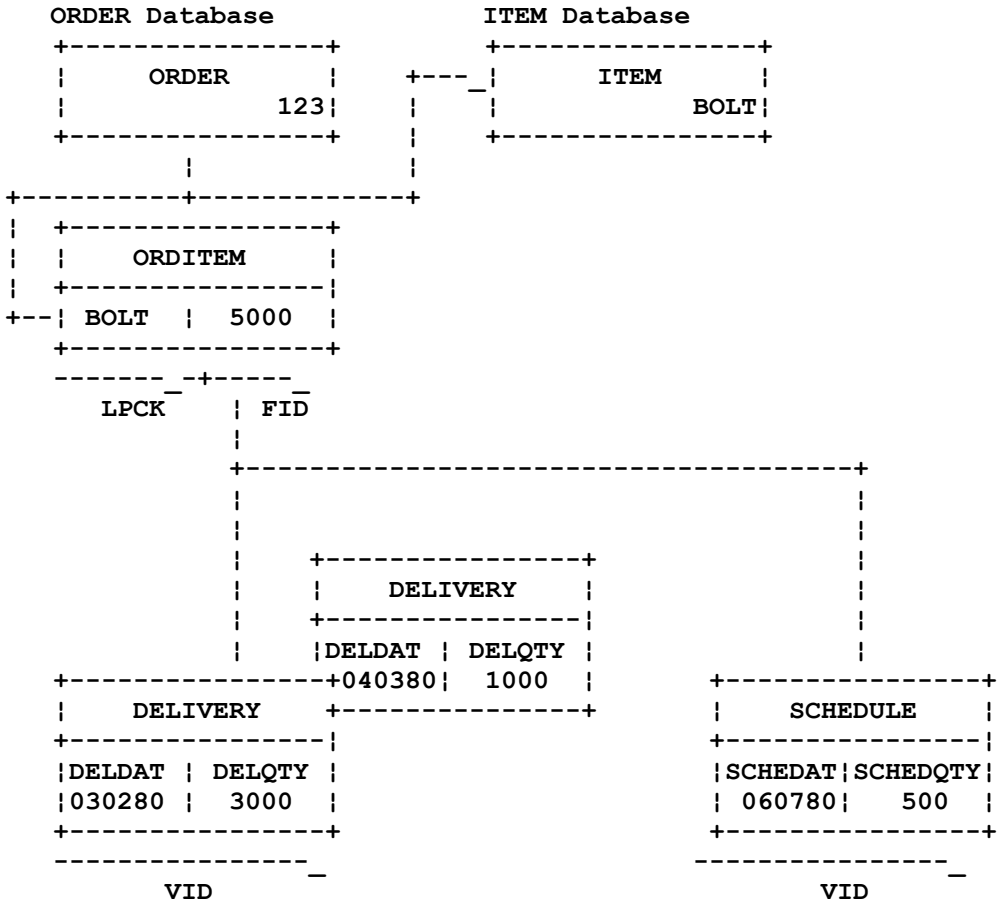
VID is used when you have data that is unique to a relationship, but several occurrences of it exist. For example, suppose you cannot supply in one shipment the total quantity of an item required for an order. You need to store delivery data showing the quantity delivered on a specified date. The delivery date is not dependent on either the order or item alone. It is dependent on a specific order-item combination. Therefore, it is stored as a dependent of the logical child segment. The data in this dependent of the logical child is called variable intersection data. For each logical child occurrence, there can be as many occurrences of dependent segments containing intersection data as you need.

Figure below shows variable intersection data. In the ORDER 123 segment for the item BOLT, 3000 were delivered on March 2 and 1000 were delivered on April 3. Because of this, two occurrences of the DELIVERY segment exist. Multiple segment types can contain intersection data for a single logical child segment. In addition to the DELIVERY segment shown in the figure, notice the SCHEDULE segment type. This segment type shows the planned shipping date and the number of items to be shipped. Segment types containing VID can all exist at the same level in the hierarchy as shown in the figure, or they can be dependents of each other.

FID, VID and physical pairing

In the previous figures, intersection data has been stored in a unidirectional logical relationship. It works exactly the same way in the two bi-directional logical relationships. However, when physical pairing is used, VID can only be stored on one side of the relationship. It does not matter on which side it is stored. An application program can access it using either the ORDER or ITEM database. FID, on the other hand, must be stored on

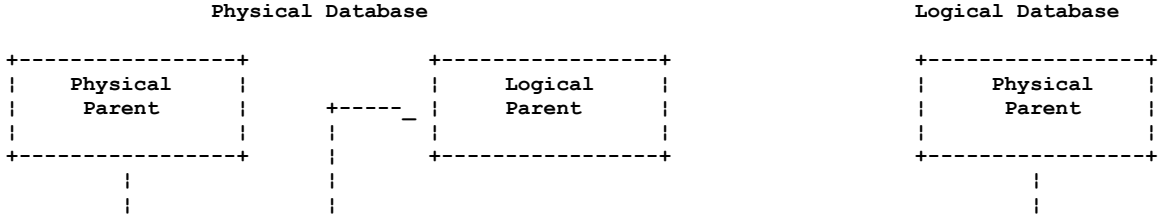
both sides of the relationship when physical pairing is used. IMS automatically maintains the FID on both sides of the relationship when it is changed on one side. However, extra time is required for maintenance, and extra space is required on DASD for FID in a physically paired relationship.



Variable Intersection Data

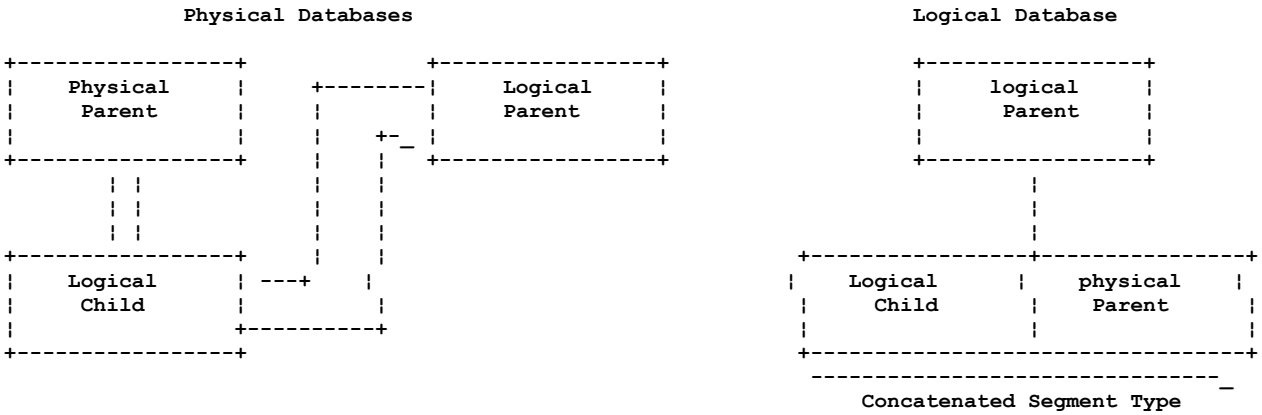
Paths Used in Logical Relationships

The relationship between physical parent and logical child in a physical database and the LP pointer in each logical child creates a physical parent to logical parent path. To define use of the path, the logical child and logical parent are defined as a concatenated segment type, as shown in Figure below. Definition of the path and the concatenated segment type is done in what is called a logical database. How to define a logical database is shown later in this section.



Defining a Physical Parent to Logical Parent Path in a Logical Database

In addition, when LC pointers are used in the logical parent and logical twin and PP pointers are used in the logical child, a logical parent to physical parent path is created. To define use of the path, the logical child and physical parent are defined as one concatenated segment type that is a physical child of the logical parent, as shown in Figure below. Again, definition of the path is done in a logical database.



Defining a Logical Parent to Physical Parent Path in a Logical Database

When use of a physical parent to logical parent path is defined, the physical parent is the parent of the concatenated segment type. When an application program retrieves an occurrence of the concatenated segment type from a physical parent, the logical child and its logical parent are concatenated and presented to the application program as one segment. When use of a logical parent to physical parent path is defined, the logical parent is the parent of the concatenated segment type. When an application program retrieves an occurrence of the concatenated segment type from a logical parent, an occurrence of the logical child and its physical parent are concatenated and presented to the application program as one segment.

In both cases, the physical parent or logical parent segment included in the concatenated segment is called the destination parent. For a physical parent to logical parent path, the logical parent is the destination parent in the concatenated segment. For a logical parent to physical parent path, the physical parent is the destination parent in the concatenated segment.

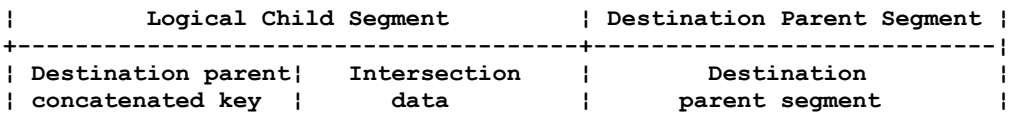
The Logical Child Segment

When defining a logical child in its physical database, the length specified for it must be large enough to contain the concatenated key of the logical parent. Any length greater than that can be used for intersection data.

To identify which logical parent is pointed to by a logical child, the concatenated key of the logical parent must be present. Each logical child segment must be present in the application program's I/O area when the logical child is initially presented for loading into the database. However, if the logical parent is in an HD database, its concatenated key might not be written to storage when the logical child is loaded. If the logical parent is in a HISAM database, a logical child in storage must contain the concatenated key of its logical parent.

For logical child segments, you can define a special operand on the PARENT= parameter of the SEGM statement. This operand determines whether a symbolic pointer to the logical parent is stored as part of the logical child segment on the storage device. If PHYSICAL is specified, the concatenated key of the logical parent is stored with each logical child segment. If VIRTUAL is specified, only the intersection data portion of each logical child segment is stored.

When a concatenated segment is retrieved through a logical database, it contains the logical child segment, which consists of the concatenated key of the destination parent, followed by any intersection data. In turn, this is followed by data in the destination parent. Figure below shows the format of a retrieved concatenated segment in the I/O area. The concatenated key of the destination parent is returned with each concatenated segment to identify which destination parent was retrieved. IMS gets the concatenated key from the logical child in the concatenated segment or by constructing the concatenated key. If the destination parent is the logical parent and its concatenated key has not been stored with the logical child, IMS constructs the concatenated key and presents it to the application program. If the destination parent is the physical parent, IMS must always construct its concatenated key.



Format of Concatenated Segment Returned to User I/O Area

Defining Sequence Fields for Databases Using Logical Relationships

To avoid potential problems in processing databases using logical relationships, unique sequence fields should be defined in all logical parent segments and in all segments a logical parent is dependent on in its physical database. When unique sequence fields are not defined in all segments on the path to and including a logical parent, multiple logical parents in a database can have the same concatenated key. When this happens, problems can arise during and after initial database load when symbolic logical parent pointers in logical child segments are used to establish position on a logical parent segment.

At initial database load time, if logical parents with non unique concatenated keys exist in a database, the resolution utilities attach all logical children with the same concatenated key to the first logical parent in the database with that concatenated key.

When inserting or deleting a concatenated segment and position for the logical parent, part of the concatenated segment is determined by the logical parent's concatenated key. Positioning for the logical parent starts at the root and stops on the first segment at each level of the logical parent's database that satisfies the key equal condition for that level. If a segment is missing on the path to the logical parent being inserted, a GE status code is returned to the application program when using this method to establish position in the logical parent's database.

Defining Sequence Fields for Real Logical Children

If the sequence field of a real logical child consists of any part of the logical parent's concatenated key, PHYSICAL must be specified on the PARENT= parameter in the SEGM statement for the logical child. This will cause the concatenated key of the logical parent to be stored with the logical child segment.

Defining Sequence Fields for Virtual Logical Children

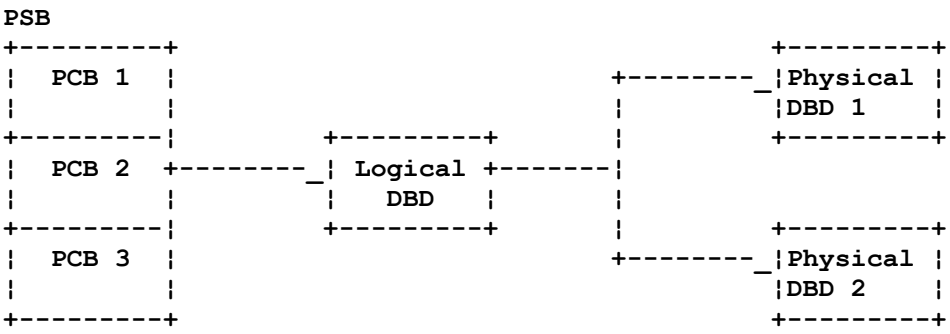
As a general rule, a segment can have only one sequence field. However, in the case of virtual pairing, multiple FIELD statements can be used to define a logical sequence field for the virtual logical child.

A sequence field must be specified for a virtual logical child if, when accessing it from its logical parent, you need real logical child segments retrieved in an order determined by data in a field of the virtual logical child as it could be seen in the application program I/O area. This sequence field can include any part of the segment as it appears when viewed from the logical parent (that is, the concatenated key of the real logical child's physical parent followed by any intersection data). Because it can be necessary to describe the sequence field of a logical child as accessed from its logical parent in discontiguous pieces, multiple FIELD statements with the SEQ parameter present are permitted. Each statement must contain a unique fldname1 parameter.

Relationship of Control Blocks When a Logical Relationship Is Used

When a logical relationship is used, you must define the physical databases involved in the relationship to IMS. This is done using a physical DBD. In addition, you must define the logical structure to IMS since this is the structure the application program perceives. This is done using a logical DBD. A logical DBD is needed because the application program's PCB references a DBD, and the physical DBD does not reflect the logical data structure the application program needs to access. Finally, the application program needs a PSB, consisting of one or more PCBs. The PCB that is used when processing with a logical relationship points to the logical DBD. This PCB indicates which segments in the logical database the application program can process. It also indicates what type of processing the application program can perform on each segment.

Figure below shows the relationship between these three control blocks. It assumes that the logical relationship is established between two physical databases. The following sections explain how the physical and logical DBD are coded when a logical relationship is defined.



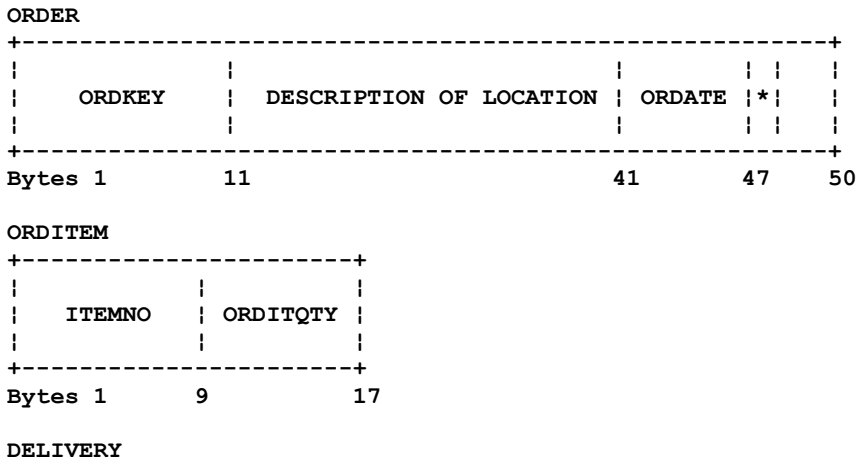
Relationship of Control Blocks When a Logical Relationship is Used

How to Specify Use of Logical Relationships in the Physical DBD

For each of the databases involved in a logical relationship, you must code a physical DBD. All statements in the physical DBD are coded with the same format used when a logical relationship is not defined, except for the SEGM and LCHILD statements. The SEGM statement, which describes a segment and its length and position in the database hierarchy, is expanded to include the new types of pointers. The LCHILD statement is added to define the logical relationship between the two segment types. Figure below(part 2 of 2) shows an example of how the physical DBD is coded.

In the SEGM statements of the following examples, only the pointers required with logical relationships are shown. No pointers required for use with HD databases are shown. When actually coding a DBD, you must ask for these pointers in the PTR= parameter. Otherwise, IMS will not generate them once another type of pointer is specified.

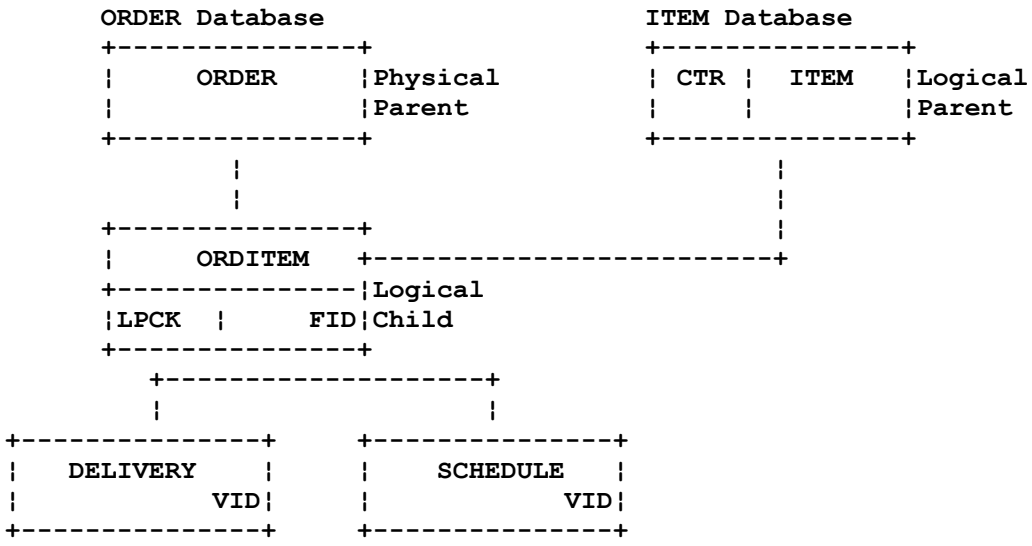
Figure below (part 1 of 2) shows the layout of segments. Figure below(part 2 of 2) shows physical DBDs for unidirectional relationships.



```
+-------------------------------------------------------+
|         |           |                           | |   |
| DELDAT  | QUANTITY  |       DESCRIPTION         |*|   |
|         |           |                           | |   |
+-------------------------------------------------------+
Bytes 1   7           15                          45    50

SCHEDULE
+-------------------------------------------------------+
|         |           | PLANNED   | INVEN-  |           |
| SCHEDAT | QUANTITY  | SHIPPING  | TORY    | DESCRIPTION |
|         |           |   DATE    | PLACE   |           |
+-------------------------------------------------------+
Bytes 1   7           15          21        25          50

ITEM
+-------------------------------------------------------------------+
|            |                                        |DATE OF| |   |
|  ITEMKEY   |              DESCRIPTION               |CREA-  |*|   |
|            |                                        |TION   | |   |
+-------------------------------------------------------------------+
Bytes 1      9                                        49      55    60

* = Flag
```

Part 1 of 2 Layouts of Segments Used in the Examples

This is the hierarchic structure of the two databases involved in the logical relationship. In this example, we are defining a unidirectional relationship using symbolic pointing. ORDITEM has an LPCK and FID, and DELIVERY and SCHEDULE are VID.



This is the DBD for the ORDER database:

```
  DBD     NAME=ORDDB
  SEGM    NAME=ORDER,BYTES=50,FREQ=28000,PARENT=0
  FIELD   NAME=(ORDKEY,SEQ),BYTES=10,START=1,TYPE=C
  FIELD   NAME=ORDATE,BYTES=6,START=41,TYPE=C
  SEGM    NAME=ORDITEM,BYTES=17,PARENT=((ORDER),(ITEM,P,ITEMDB))
  FIELD   NAME=(ITEMNO,SEQ),BYTES=8,START=1,TYPE=C
  FIELD   NAME=ORDITQTY,BYTES=9,START=9,TYPE=C,
```

```
  SEGM    NAME=DELIVERY,BYTES=50,PARENT=ORDITEM
  FIELD   NAME=(DELDAT,SEQ),BYTES=6,START=1,TYPE=C
  SEGM    NAME=SCHEDULE,BYTES=50,PARENT=ORDITEM
  FIELD   NAME=(SCHEDAT,SEQ),BYTES=6,START=1,TYPE=C
  DBDGEN
  FINISH
  END
```

This is the DBD for the ITEM database:

```
  DBD     NAME=ITEMDB
  SEGM    NAME=ITEM,BYTES=60,FREQ=50000,PARENT=0
  FIELD   NAME=(ITEMKEY,SEQ),BYTES=8,START=1,TYPE=C
  LCHILD  NAME=(ORDITEM,ORDDB)
  DBDGEN
  FINISH
  END
```

Part 2 of 2 Physical DBDs for Unidirectional Relationship Using Symbolic Pointing

Notes to above Figure :

In the ORDER database, the DBD coding that differs from normal DBD coding is that for the logical child ORDITEM.

In the SEGM statement for ORDITEM:

1. The BYTES= parameter is 17. The length specified is the length of the LPCK, plus the length of the FID. The LPCK is the key of the ITEM segment, which is 8 bytes long. The length of the FID is 9 bytes.

2. The PARENT= parameter has two parents specified. Two parents are specified because ORDITEM is a logical child and therefore has both a physical and logical parent. The physical parent is ORDER. The logical parent is ITEM, specified after ORDER. Because ITEM exists in a different physical database from ORDITEM, the name of its physical database, ITEMDB, must be specified. Between the segment name ITEM and the database name ITEMDB is the letter P. The letter P stands for physical. The letter P specifies that the LPCK is to be stored on DASD as part of the logical child segment.

In the FIELD statements for ORDITEM:

1. ITEMNO is the sequence field of the ORDITEM segment and is 8 bytes long. ITEMNO is the LPCK. The logical parent is ITEM, and if you look at the FIELD statement for ITEM in the ITEM database, you will see ITEM's sequence field is ITEMKEY, which is 8 bytes long. Because ITEM is a root segment, the LPCK is 8 bytes long.

2. ORDITQTY is the FID and is coded normally.

In the ITEM database, the DBD coding that differs from normal DBD coding is that an LCHILD statement has been added. This statement names the logical child ORDITEM.

Because the ORDITEM segment exists in a different physical database from ITEM, the name of its physical database, ORDDB, must be specified.

Specifying Bi-directional Logical Relationships

Figure above shows the coding for a unidirectional relationship. When defining a bi-directional relationship with physical pairing, you need to include an LCHILD statement under both logical parents. In addition to other pointers, you need to include the PAIRED operand on the POINTER= parameter of the SEGM statements for both logical children.

When defining a bi-directional relationship with virtual pairing, you need to code an LCHILD statement only for the real logical child. On the LCHILD statement, you code POINTER=SNGL or DBLE to get logical child pointers. You code the PAIR= operand to indicate the virtual logical child that is paired with the real logical child. When you define the SEGM statement for the real logical child, the PARENT= parameter identifies both the physical and logical parents. You should specify logical twin pointers (in addition to any other pointers) on the POINTER= parameter. Also, you should define a SEGM statement for the virtual logical child even though it does not exist. On this SEGM statement, you specify PAIRED on the POINTER= parameter. In addition, you specify a SOURCE= parameter. On the SOURCE= parameter, you specify the SEGM name and DBD name of the real logical child. DATA must always be specified when defining SOURCE= on a virtual logical child SEGM statement.

Checklist of Rules for Defining Logical Relationships in Physical Databases

This section provides the list of rules that must be followed when defining logical relationships in physical databases. In all cases, references are to segment types, not occurrences.

Logical Child

- A logical child must have a physical and a logical parent.
- A logical child can have only one physical and one logical parent.
- A logical child is defined as a physical child in the physical database of its physical parent.
- A logical child is always a dependent segment in a physical database, and can, therefore, be defined at any level except the first level of a database.
- A logical child in its physical database cannot have a physical child defined at the next lower level in the database that is also a logical child.
- A logical child can have a physical child. However, if a logical child is physically paired with another logical child, only one of the paired segments can have physical children.

Logical Parent

- A logical parent can be defined at any level in a physical database, including the root level.

- A logical parent can have one or more logical children. Each logical child related to the same logical parent defines a logical relationship.

- A segment in a physical database cannot be defined as both a logical parent and a logical child.

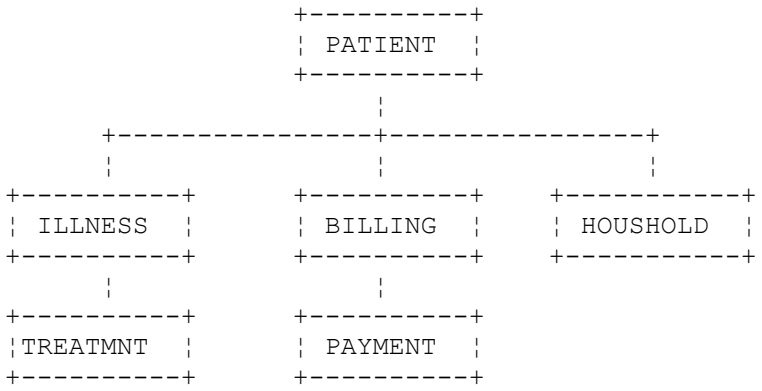
- A logical parent can be defined in the same physical database as its logical child, or in a different physical database.

Physical Parent
A physical parent of a logical child cannot also be a logical child.

Lab Exercise

This exercise defines a logical relationship between the Patient Data Base and a new Doctor data base. Both data bases are described below. **You are strongly recommended to just change the userid from USER01 to your ID and not change the data set names. This is because there are job interdependencies where the output data set from one step is an input for the next!!!.**

medical database



PATIENT SEGMENT, PATIENT has a unique key field: PATNO. PATIENT segments are stored in ascending order of their patient numbers.

PATNO	NAME	ADDR
5	10	30

ILLNESS SEGMENT, The key field is ILLDATE. Because it is possible for a patient to come to the clinic with more than one illness on the same date, this key field is nonunique; For segments with equal keys or no keys, the RULES keyword determines where the segment is inserted. Where RULES=LAST, ILLNESS segments that have an equal key are stored on a first-in first-out basis among those with equal keys. ILLNESS segments with unique keys are stored in ascending order on the date field, regardless of RULES. ILLDATE is specified in the format YYYYMMDD.

ILLDATE	ILLNAME
8	10

TREATMNT SEGMENT, The key field of the TREATMNT segment is DATE. Because a patient may receive more than one treatment on the same date, DATE is a nonunique key field. TREATMNT, like ILLNESS, has been specified as having RULES=LAST. TREATMNT segments with equal date keys are also stored on a first-in-first-out basis. DATE is specified in the same format as ILLDATE--YYYYMMDD.

SRNO	DOCTOR	DATE	MEDICINE	QUANTITY
8	10	8	10	4

BILLING SEGMENT, BILLING has no key field.

BILLING
6

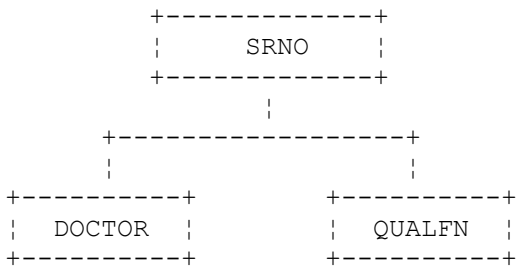
PAYMENT SEGMENT,the PAYMENT segment has no key field.

PAYMENT
6

HOUSHOLD SEGMENT, RELNAME is the key field.

RELNAME	RELATN
10	8

doctors database



SRNO SEGMENT, the SRNO segment has one field, NUMBER, which is the key field.

NUMBER
8

DOCTOR SEGMENT, DOC is the key field.

DOC	ADDRESS

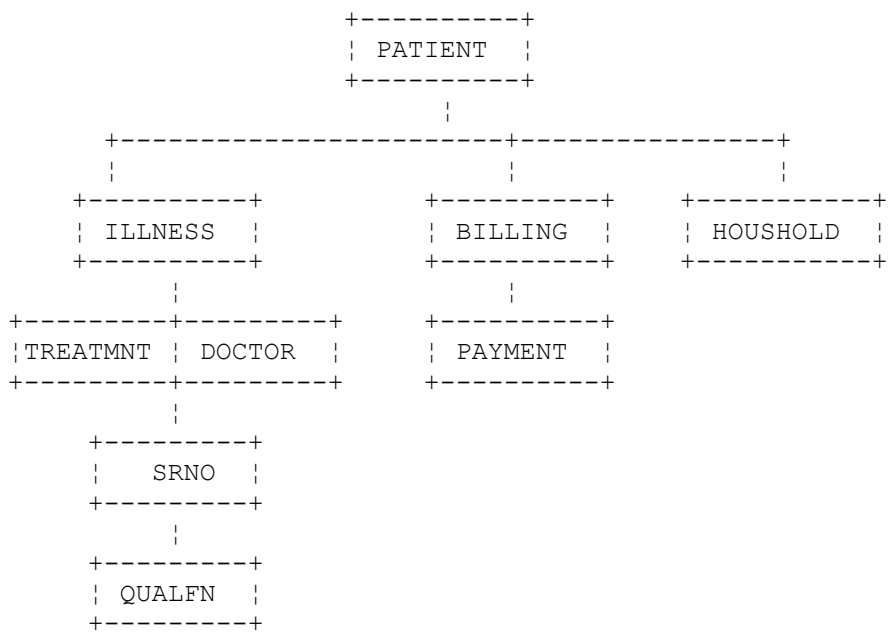
```
¦  10    ¦   50   ¦
+-----------------+
```

QUALFN SEGMENT, which has no key field.

```
+-----------+
¦    QUAL   ¦
+-----------¦
¦     8     ¦
+-----------+
```

logical relationships

THE LOGICAL DATABASE



DBD'S For the Doctors HIDAM Database

DOCDBHIL

```
DBD   NAME=DOCDBHIL,ACCESS=(HIDAM,VSAM)
DATASET DD1=DOCDBHIL
SEGM NAME=SRNO,BYTES=8,PARENT=0
LCHILD NAME=(INDXSEG,DOCDBHII),PTR=INDX
FIELD NAME=(NUMBER,SEQ,U),BYTES=8,START=1,TYPE=C
SEGM  NAME=DOCTOR,BYTES=60,PTR=T,PARENT=SRNO
LCHILD NAME=(TREATMNT,PNTDBHIL),PTR=DBLE,PAIR=TREATSEG,          X
       RULES=LAST
FIELD NAME=(DOC,SEQ,U),BYTES=10,START=1,TYPE=C
FIELD NAME=ADDRESS,BYTES=50,START=11,TYPE=C
```

```
 SEGM  NAME=TREATSEG,PTR=PAIRED,SOURCE=((TREATMNT,D,PNTDBHIL)),   X
         PARENT=DOCTOR
 SEGM  NAME=QUALFN,BYTES=8,PTR=T,PARENT=((SRNO,SNGL))
 FIELD NAME=QUAL,BYTES=8,START=1,TYPE=C
 DBDGEN
 FINISH
 END
```

DOCDBHII

```
DBD  NAME=DOCDBHII,ACCESS=INDEX
DATASET DD1=DOCDBHII
SEGM NAME=INDXSEG,BYTES=8
LCHILD NAME=(SRNO,DOCDBHIL),INDEX=NUMBER
FIELD NAME=(INDXSEQ,SEQ,U),BYTES=8,START=1,TYPE=C
DBDGEN
FINISH
END
```

DBD'S for the Patient HIDAM database

PNTDBHIL

```
DBD  NAME=PNTDBHIL,ACCESS=(HIDAM,VSAM)
DATASET DD1=PNTDBHIL
SEGM NAME=PATIENT,BYTES=45,PARENT=0
LCHILD NAME=(INDXSEG,PNTDBHLI),PTR=INDX
FIELD NAME=(PATNO,SEQ,U),BYTES=5,START=1,TYPE=C
FIELD NAME=NAME,BYTES=10,START=6,TYPE=C
FIELD NAME=ADDR,BYTES=30,START=16,TYPE=C
SEGM  NAME=ILLNESS,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=(ILLDATE,SEQ,M),BYTES=8,START=1,TYPE=C
FIELD NAME=ILLNAME,BYTES=10,START=9,TYPE=C
SEGM NAME=TREATMNT,BYTES=40,POINTER=(TWIN,LTWIN),                X
       PARENT=((ILLNESS,SNGL),(DOCTOR,V,DOCDBHIL)),              X
       RULES=(LLV,LAST)
FIELD NAME=DOCTOR,BYTES=18,START=1,TYPE=C
FIELD NAME=DATE,BYTES=8,START=19,TYPE=C
FIELD NAME=MEDICINE,BYTES=10,START=27,TYPE=C
FIELD NAME=QUANTITY,BYTES=4,START=37,TYPE=C
SEGM  NAME=BILLING,BYTES=6,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=BILLING,BYTES=6,START=1,TYPE=C
SEGM  NAME=PAYMENT,BYTES=6,PTR=T,PARENT=((BILLING,SNGL))
FIELD NAME=PAYMENT,BYTES=6,START=1,TYPE=C
SEGM  NAME=HOUSHLD,BYTES=18,PTR=T,PARENT=((PATIENT,SNGL))
FIELD NAME=RELNAME,BYTES=10,START=1,TYPE=C
FIELD NAME=RELATN,BYTES=8,START=11,TYPE=C
DBDGEN
FINISH
END
```

PNTDBHLI

```
DBD  NAME=PNTDBHLI,ACCESS=INDEX
DATASET DD1=PNTDBHII
SEGM NAME=INDXSEG,BYTES=5
LCHILD NAME=(PATIENT,PNTDBHIL),INDEX=PATNO
FIELD NAME=(INDXSEQ,SEQ,U),BYTES=5,START=1,TYPE=C
DBDGEN
FINISH
END
```

The Logical DBD to access the database after it has been loaded DBDLOGCL

```
DBD   NAME=DBDLOGCL,ACCESS=LOGICAL
DATASET LOGICAL
SEGM NAME=PATIENT,PARENT=0,SOURCE=((PATIENT,D,PNTDBHIL))
SEGM NAME=ILLNESS,PARENT=PATIENT,SOURCE=((ILLNESS,D,PNTDBHIL))
SEGM NAME=TREATMNT,PARENT=ILLNESS,                                X
       SOURCE=((TREATMNT,D,PNTDBHIL),(DOCTOR,D,DOCDBHIL))
SEGM  NAME=SRNO,PARENT=TREATMNT,SOURCE=((SRNO,D,DOCDBHIL))
SEGM  NAME=QUALFN,PARENT=SRNO,SOURCE=((QUALFN,D,DOCDBHIL))
SEGM  NAME=BILLING,PARENT=PATIENT,SOURCE=((BILLING,D,PNTDBHIL))
SEGM  NAME=PAYMENT,PARENT=BILLING,SOURCE=((PAYMENT,D,PNTDBHIL))
SEGM  NAME=HOUSHLD,PARENT=PATIENT,SOURCE=((HOUSHLD,D,PNTDBHIL))
DBDGEN
FINISH
END
```

The PSB's to load / access the Patient database

PNTLOGL (loading)

```
PCB   TYPE=DB,NAME=PNTDBHIL,PROCOPT=LS,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PNTLOGL,LANG=PL/I
END
```

PNTLOGA

```
PCB   TYPE=DB,NAME=PNTDBHIL,PROCOPT=A,KEYLEN=21
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PNTLOGA,LANG=PL/I
END
```

The PSB's to load / access the Doctor database

DOCLOGL (loading)

```
PCB   TYPE=DB,NAME=DOCDBHIL,PROCOPT=LS,KEYLEN=18
SENSEG NAME=SRNO,PARENT=0
SENSEG NAME=DOCTOR,PARENT=SRNO
SENSEG NAME=QUALFN,PARENT=SRNO
PSBGEN PSBNAME=DOCLOGA,LANG=PL/I
END
```

DOCLOGA (accessing)

```
PCB   TYPE=DB,NAME=DOCDBHIL,PROCOPT=A,KEYLEN=18
SENSEG NAME=SRNO,PARENT=0
SENSEG NAME=DOCTOR,PARENT=SRNO
SENSEG NAME=QUALFN,PARENT=SRNO
PSBGEN PSBNAME=DOCLOGA,LANG=PL/I
```

```
        END
```

PSB to access the joined databases after the logical relationships have been established

PSBLOGCL

```
PCB   TYPE=DB,NAME=DBDLOGCL,PROCOPT=A,KEYLEN=40
SENSEG NAME=PATIENT,PARENT=0
SENSEG NAME=ILLNESS,PARENT=PATIENT
SENSEG NAME=TREATMNT,PARENT=ILLNESS
SENSEG NAME=SRNO,PARENT=TREATMNT
SENSEG NAME=QUALFN,PARENT=SRNO
SENSEG NAME=BILLING,PARENT=PATIENT
SENSEG NAME=PAYMENT,PARENT=BILLING
SENSEG NAME=HOUSHLD,PARENT=PATIENT
PSBGEN PSBNAME=PSBLOGCL,LANG=PL/I
END
```

STEP 1

PREORGL

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//     JCLLIB ORDER=(USER01.PROCLIB)
//STEP1   EXEC PREREORG
//SYSPRINT  DD SYSOUT=A,DCB=BLKSIZE=1200
//G.DFSURCDS  DD DSNAME=USER01.RLCDS.LOGICAL,DISP=(NEW,KEEP),
//             UNIT=SYSDA,VOL=SER=IMSMSC,DCB=(BLKSIZE=1600),
//             SPACE=(CYL,1)
//G.SYSIN     DD *,DCB=BLKSIZE=80
DBIL=PNTDBHIL,DOCDBHIL
/*
//
```

STEP 2

1. Load the databases separately.
2. Create the DBD'S and PSB's using the JCL shown in earlier sections.
3. Create the VSAM clusters using the guidelines in the DBD listing. See the sample JCL below which is based on DBD generation recommendations.

VSAM CLUSTER GENERATION

The clusters generated are
USER01.PNTDBHIL
USER01.PNTDBHII
USER01.DOCDBHIL
USER01.DOCDBHII

IDCHILD

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
```

```
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
   DELETE USER01.DOCDBHII CLUSTER
   DELETE USER01.DOCDBHIL CLUSTER
   DEFINE CLUSTER (NAME(USER01.DOCDBHIL) NONINDEXED -
   RECORDSIZE(2041,2041) CONTROLINTERVALSIZE(2048) -
   TRACKS(2 2))
   DEFINE CLUSTER (NAME(USER01.DOCDBHII) INDEXED KEYS(8,5) -
   RECORDSIZE(14,14) TRACKS(1,1)) DATA(CONTROLINTERVALSIZE(1024))
//
```

IDCHILP

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
   DELETE USER01.PNTDBHIL CLUSTER
   DELETE USER01.PNTDBHII CLUSTER
   DEFINE CLUSTER (NAME(USER01.PNTDBHIL) NONINDEXED -
   RECORDSIZE(2041,2041) CONTROLINTERVALSIZE(2048) -
   TRACKS(2 2))
   DEFINE CLUSTER (NAME(USER01.PNTDBHII) INDEXED KEYS(5,5) -
   RECORDSIZE(10,10) TRACKS(1,1)) DATA(CONTROLINTERVALSIZE(1024))
//
```

Data for the doctor data base (DATA10)

```
SRNO      00000001
DOCTOR    DR.DOBBS  18,NORTH STREET,CHENNAI
QUALFN    PHD
QUALFN    MS
SRNO      00000002
DOCTOR    DR.JAMES  22, TMBUKTOO, AFRICA
QUALFN    DPM
QUALFN    FRCS
SRNO      00000003
DOCTOR    DR.PILOO  11, LONG ISLAND, NY
QUALFN    FRCS
SRNO      00000004
DOCTOR    DR.TOM    12, CALCUTTA
QUALFN    MS
SRNO      00000005
DOCTOR    DR.YOUNG  13, LONDON
QUALFN    MBBS
```

Data for the patient database (DATA11)

```
PATIENT   00001ABCDEF1   18,CHN 600023-1
ILLNESS   01012000MALARIA
TREATMNT  00000001DR.DOBBS  01012000QUININE   0004
BILLING   000600
PAYMENT   000600
HOUSHLD   MOHAN     FATHER
PATIENT   00002ABCDEF2   18,CHN 600023-2
ILLNESS   01012000JAUNDICE
TREATMNT  00000002DR.JAMES  01012000AYURVEDIC 0004
```

```
BILLING   000500
PAYMENT   000400
PAYMENT   000100
HOUSHLD   MEERA     MOTHER
PATIENT   00003ABCDEF3   18,CHN 600023-3
ILLNESS   01012000FLU
TREATMNT  00000003DR.PILOO  01012000CROCIN    0004
BILLING   000400
PAYMENT   000400
HOUSHLD   JAYA      SISTER
PATIENT   00004ABCDEF4   18,CHN 600023-4
ILLNESS   01012000MEASLES
TREATMNT  00000004DR.TOM    01012000NEEMLEAVES0004
BILLING   000300
PAYMENT   000200
PAYMENT   000100
HOUSHLD   MAYA      SISTER
PATIENT   00005ABCDEF5   18,CHN 600023-5
ILLNESS   01012000TYPHOID
TREATMNT  00000005DR.YOUNG  01012000ANTIBIOTIC0004
BILLING   000200
PAYMENT   000200
HOUSHLD   LATA      SISTER
```

Loading

Use PSB DOCLOGL to load the DOCTOR database. Use PSB PNTLOGL to load the patient Data Base. In either case use PLIPGM5 for loading the database. Use DATA10 data member for the Doctor database and DATA11 for the Patient database.

A data set DDNAME DFSURWF1 is created in each of the two load runs. The data sets are:-
USER01.DFSURWF.DOC.LOGICAL for the Doctor Database, and
USER01.DFSURWF.PNT.LOGICAL for the Patient Database
Remember to change the DSN appropriately for each run.

Remember to delete USER01.IMSLOG each time, as we are not contemplating any recovery. If the load fails in between, delete the VSAM clusters and redefine them again, rerun the load after correcting any input data errors.

Batch JCL

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC DLIBATCH,MBR=PLIPGM5,PSB=ßPNTLOGL | DOCLOGLà,DBRC=N
//G.SYSPRINT DD SYSOUT=*
//*
//*  THIS IS FOR THE HIDAM PATIENT DATABASE
//G.DOCDBHIL DD DSN=USER01.DOCDBHIL,DISP=SHR
//G.DOCDBHII DD DSN=USER01.DOCDBHII,DISP=SHR
//*
//*  THIS IS FOR THE HIDAM DOCTOR DATABASE
//G.PNTDBHIL DD DSN=USER01.PNTDBHIL,DISP=SHR
//G.PNTDBHII DD DSN=USER01.PNTDBHII,DISP=SHR
```

```
//*
//* THIS IS THE LOAD DATA, DATA1 FOR PATIENT, DATA10 FOR DOCTOR
//G.SYSIN DD DSN=USER01.LOAD.DATA(ßDATA11 | DATA10à),DISP=SHR
//*
//* USE THIS ONLY FOR LOAD OF LOGICAL RELATIONSHIP / SECONDARY INDEXES
//G.DFSURWF1 DD DSN=USER01.DFSURWF.DOC.LOGICAL,DISP=(NEW,KEEP),
//    DCB=(RECFM=VB,LRECL=900,BLKSIZE=1208),SPACE=(TRK,(1,1))
//*
//* USE THIS ONLY FOR LOGICAL RELATIONSHIPS
//* WHICH IS THE OUTPUT OF THE PREREORG UTILITY
//* AND IS AN INPUT HERE
//DFSURCDS  DD DSN=USER01.RLCDS.LOGICAL,DISP=SHR
//*
//G.DFSVSAMP DD *
VSRBF=2048,4
/*
//
```

DLIBATCH

```
//        PROC MBR=TEMPNAME,PSB=,BUF=7,
//             SPIE=0,TEST=0,EXCPVR=0,RST=0,PRLD=,
//             SRCH=0,CKPTID=,MON=N,LOGA=0,FMTO=T,
//             IMSID=,SWAP=,DBRC=,IRLM=,IRLMNM=,
//              BKO=N,IOB=,SSM=,APARM=,
//             RGN=2048K,
//*            SOUT=A,LOGT=2400,SYS2=,
//             SOUT=A,SYS2=,
//             LOCKMAX=,GSGNAME=,TMINAME=
//G       EXEC PGM=DFSRRC00,REGION=&RGN,
//             PARM=(DLI,&MBR,&PSB,&BUF,
//             &SPIE&TEST&EXCPVR&RST,&PRLD,
//             &SRCH,&CKPTID,&MON,&LOGA,&FMTO,
//             &IMSID,&SWAP,&DBRC,&IRLM,&IRLMNM,
//             &BKO,&IOB,&SSM,'&APARM',
//             &LOCKMAX,&GSGNAME,&TMINAME)
//STEPLIB  DD DSN=IMS.&SYS2.RESLIB,DISP=SHR
//         DD DSN=IMS.&SYS2.PGMLIB,DISP=SHR
//         DD DSN=USER01.LOADLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.&SYS2.RESLIB,DISP=SHR
//IMS      DD DSN=USER01.PSBLIB,DISP=SHR
//         DD DSN=USER01.DBDLIB,DISP=SHR
//PROCLIB  DD DSN=IMS.&SYS2.PROCLIB,DISP=SHR
//IEFRDER  DD DSN=USER01.IMSLOG,DISP=(NEW,KEEP),
//         DCB=(RECFM=VB,BLKSIZE=1920,
//         LRECL=1916,BUFNO=2),SPACE=(TRK,(1,1))
//SYSUDUMP DD SYSOUT=&SOUT,
//         DCB=(RECFM=FBA,LRECL=121,BLKSIZE=605),
//         SPACE=(605,(500,500),RLSE,,ROUND)
//IMSMON   DD DUMMY
```

STEP 3

Make sure you have this procedure in your userid.PROCLIB

```
//        PROC
//G       EXEC PGM=DFSRRC00,REGION=1024K,
//         PARM='ULU,DFSURPR0,,,1,,,,,,,,N,N'
//STEPLIB DD DSN=IMS.RESLIB,DISP=SHR
```

```
//IMS      DD DISP=SHR,DSN=SCBKSR.DBDLIB
//SYSPRINT DD SYSOUT=A
//SYSUDUMP DD SYSOUT=A
```

The input for this step is
USER01.DFSURWF.DOC.LOGICAL from the Doctor Database load
USER01.DFSURWF.PNT.LOGICAL from the Patient Database load
USER01.RLCDS.LOGICAL from the PRERORG run

The output is
USER01.DFSURIDX.DOC.LOGICAL
USER01.WF3.DOC.LOGICAL
Note that USER01.WF2.DOC.LOGICAL is an intermediate work data set and need not be carried to step 4.
Note that the ddname DFSURIDX is also not needed unless secondary indexes are present.

PREFXGNL

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
//PREFXRES EXEC PGM=DFSURG10
//STEPLIB  DD DSN=IMS.RESLIB,DISP=SHR
//SYSUDUMP DD SYSOUT=A
//SYSPRINT DD SYSOUT=A,DCB=BLKSIZE=1200
//SYSOUT   DD SYSOUT=A
//SORTLIB  DD DSN=SYS1.SORTLIB,DISP=SHR
//SORTWK01 DD  UNIT=SYSDA,SPACE=(1008,(60),,CONTIG)
//SORTWK02 DD  UNIT=SYSDA,SPACE=(1008,(60),,CONTIG)
//SORTWK03 DD  UNIT=SYSDA,SPACE=(1008,(60),,CONTIG)
//SORTIN   DD DSN=USER01.DFSURWF.DOC.LOGICAL,DISP=SHR
//         DD DSN=USER01.DFSURWF.PNT.LOGICAL,DISP=SHR
//DFSURWF2 DD DSN=USER01.WF2.DOC.LOGICAL,
//         UNIT=SYSDA,SPACE=(1008,(30),,CONTIG),
//         DISP=(NEW,KEEP),DCB=(RECFM=VB,LRECL=900,BLKSIZE=1008)
//DFSURWF3 DD DSN=USER01.WF3.DOC.LOGICAL,
//         UNIT=SYSDA,SPACE=(1008,(30),,CONTIG),
//         DISP=(NEW,KEEP),DCB=(RECFM=VB,LRECL=900,BLKSIZE=1008)
//DFSURIDX DD DSN=USER01.DFSURIDX.DOC.LOGICAL,
//        UNIT=SYSDA,SPACE=(1008,(30),,CONTIG),
//         DISP=(NEW,KEEP),DCB=(RECFM=VB,LRECL=900,BLKSIZE=1008)
//DFSURCDS DD DSN=USER01.RLCDS.LOGICAL,DISP=(SHR)
//PNTDBHIL DD DSN=USER01.PNTDBHIL,DISP=SHR
//PNTDBHII DD DSN=USER01.PNTDBHII,DISP=SHR
//DOCDBHIL DD DSN=USER01.DOCDBHIL,DISP=SHR
//DOCDBHII DD DSN=USER01.DOCDBHII,DISP=SHR
/*
```

STEP 4

The input is
USER01.DFSURIDX.DOC.LOGICAL from step 3
USER01.WF3.DOC.LOGICAL from step3
USER01.DFSURIDX.DOC.LOGICAL is needed only if secondary indexes are present.

PRFXUPDL

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1),REGION=0M
```

```
//PREFXRES EXEC PGM=DFSRRC00,PARM='ULU,DFSURGP0,,,,,,,,,,,N'
//STEPLIB  DD DSN=IMS.RESLIB,DISP=SHR
//IMS      DD DSN=USER01.DBDLIB,DISP=SHR
//SYSUDUMP DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSOUT   DD SYSOUT=A
//DFSURWF3 DD DSN=USER01.WF3.DOC.LOGICAL,DISP=SHR
//DFSURIDX DD DSN=USER01.DFSURIDX.DOC.LOGICAL,DISP=SHR
//PNTDBHIL DD DSN=USER01.PNTDBHIL,DISP=SHR
//PNTDBHII DD DSN=USER01.PNTDBHII,DISP=SHR
//DOCDBHIL DD DSN=USER01.DOCDBHIL,DISP=SHR
//DOCDBHII DD DSN=USER01.DOCDBHII,DISP=SHR
//DFSVSAMP DD *
2048,8
/*
//IEFRDER  DD DSN=USER01.IMSLOG,DISP=(NEW,KEEP),
//         DCB=(RECFM=VB,BLKSIZE=1920,
//         LRECL=1916,BUFNO=2),SPACE=(TRK,(1,1))
//
```

STEP5

Use the PSB which points to the logical database to dump the database using PLIPGM4 And the PSB PSBLOGCL.

Batch JCL

```
//USER011  JOB NOTIFY=&SYSUID,CLASS=A,MSGLEVEL=(1,1)
// JCLLIB ORDER=(USER01.PROCLIB)
//STEP1 EXEC DLIBATCH,MBR=PLIPGM4,PSB= PSBLOGCL,DBRC=N
//G.SYSPRINT DD SYSOUT=*
//*
//*  THIS IS FOR THE HIDAM PATIENT DATABASE
//G.DOCDBHIL DD DSN=USER01.DOCDBHIL,DISP=SHR
//G.DOCDBHII DD DSN=USER01.DOCDBHII,DISP=SHR
//*
//*  THIS IS FOR THE HIDAM DOCTOR DATABASE
//G.PNTDBHIL DD DSN=USER01.PNTDBHIL,DISP=SHR
//G.PNTDBHII DD DSN=USER01.PNTDBHII,DISP=SHR
//*
//G.DFSVSAMP DD *
VSRBF=2048,4
/*
//
```