CA IDMS™ DLI Transparency

DLI Transparency User Guide

Release 18.5.00

CA Technologies Product References

This document references the following CA product:

- CA IDMS™/DB Database

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 1: Introduction

This section contains the following topics:

Overview

CA IDMS DLI Transparency allows DL/I application programs to perform processing against CA IDMS/DB databases. DL/I applications can run in the IMS-DB batch or DL/I batch environment or the DL/I CICS environment.

Note: DL/I refers to the DBMS in the z/OS or z/VSE environment.

This chapter presents an overview of the components you use to set up your CA IDMS DLI Transparency environment to access a CA IDMS/DB database. CA IDMS Database Transparency Option for DLI permits application programs to execute against a CA IDMS/DB Database. This guide explains how to use CA IDMS Transparency for DLI and includes all phases from designing and loading the CA IDMS/DB database(s) to executing the DL/I application programs.

This guide is intended to serve as a comprehensive reference for CA IDMS DLI Transparency.

This document is intended for the person responsible for setting up the CA IDMS Transparency for DLI environment who has a working knowledge of DL/I.

Introduction to CA IDMS DLI Transparency

What is CA IDMS DLI Transparency

CA IDMS DLI Transparency provides the basis for a gradual and orderly migration from DL/I to CA IDMS/DB. Specifically, it lets you:

- Convert existing DL/I database definitions to equivalent CA IDMS/DB database definitions
- Load the existing data from the DL/I databases to the new CA IDMS/DB database
- Produce a run-time interface module to translate DL/I database requests in existing applications to equivalent CA IDMS/DB database requests

CA IDMS DLI Transparency allows you to move from the DL/I environment to the CA IDMS/DB environment without having to sacrifice the investment in your existing DL/I applications.

Once you have used CA IDMS DLI Transparency to make the transition to CA IDMS/DB, you can convert your DL/I applications to native CA IDMS/DB applications at your own pace and in keeping with your site's manpower and machine resources.

CA IDMS DLI Transparency is Transparent to Applications

Because CA IDMS DLI Transparency is generally transparent to DL/I applications, you have to perform little program alteration. Recompilation of DL/I programs is required only if they contain nonsupported features such as logging calls. Batch and CICS programs must be relinked with the CA IDMS DLI Transparency language interface.

DL/I Application Conversion Not Required

Since your DL/I applications will continue to run as expected, you do not have to convert them. However, you may want to convert them to take advantage of CA IDMS/DB's advanced features, including its relational capabilities. Additionally, you may want to develop your own native CA IDMS/DB applications to run against the migrated DL/I databases.

Note: You cannot use CA IDMS/DB facilities to redesign a migrated DL/I database. The CA IDMS DLI Transparency data structures must be maintained to ensure that your DL/I applications will continue to work as expected.

The remainder of this section discusses the following topics:

- CA IDMS DLI Transparency concepts and facilities
- Usage requirements

CA IDMS DLI Transparency Concepts and Facilities

CA IDMS DLI Transparency is an Interface to CA IDMS/DB

CA IDMS DLI Transparency serves as an interface between DL/I application programs and CA IDMS/DB databases. The DL/I applications can be written in COBOL, Assembler, or PL/I.

What CA IDMS DLI Transparency Does at Run Time

At program run time, CA IDMS DLI Transparency intercepts DL/I retrieval and update requests and translates them into CA IDMS/DB requests. The CA IDMS/DB requests are then processed by the CA IDMS/DB database management system (DBMS) for retrieval or database update.

For data retrieval, CA IDMS/DB returns requested data and/or status information, including updated program control block (PCB) information, to CA IDMS DLI Transparency. CA IDMS DLI Transparency places the data in a DL/I segment format expected by the application. For updates, CA IDMS DLI Transparency places the updates in CA IDMS/DB record format and transmits them to CA IDMS/DB to apply to the database. CA IDMS/DB, in turn, sends the resulting status information to CA IDMS DLI Transparency for communication to the application.

CA IDMS DLI Transparency Components

CA IDMS DLI Transparency consists of the following major components:

- The CA IDMS DLI Transparency syntax generator
- The interface program specification block (IPSB) compiler
- The CA IDMS DLI Transparency run-time interface
- The CA IDMS DLI Transparency load utility

Each component is described briefly below and in detail in Appendix B, 'CA IDMS DLI Transparency Software Components.'

The CA IDMS DLI Transparency Syntax Generator

What is the CA IDMS DLI Transparency Syntax Generator

The CA IDMS DLI Transparency syntax generator helps to automate the conversion process on the database definition level. It accepts as input control blocks (load modules) for the program specification blocks (PSBs) and database definitions (DBDs). These are used by the DL/I application against the existing DL/I database(s).

The Syntax Generator Produces Source Statements

For output, the syntax generator produces the source statements necessary to create the interface program specification block (IPSB). It also produces source definitions needed to create an appropriate schema, DMCL, and subschema. Collectively, the schema, DMCL, and subschema definitions represent the database definitions for the new CA IDMS/DB database.

After producing the sets of source statements, you can check them and modify them (particularly the DMCL), to address capacity planning and performance and tuning concerns.You can then input the source statements to the CA IDMS/DB compilers and the IPSB compiler, respectively.

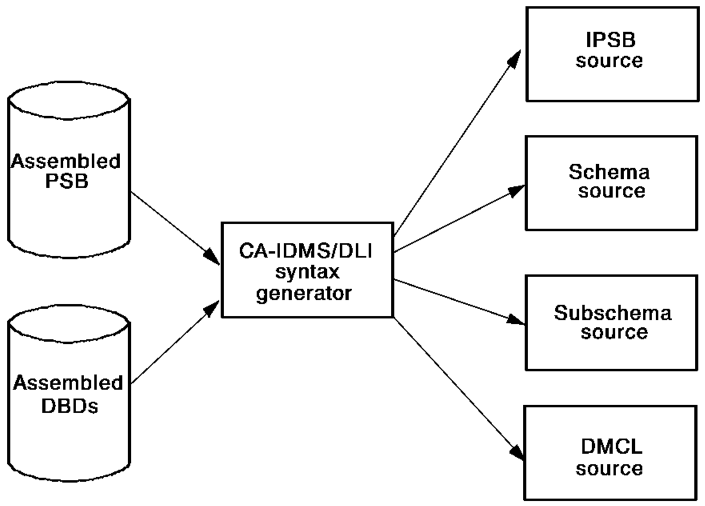


Figure 1. CA IDMS DLI Transparency syntax generator

The IPSB Compiler

What is the IPSB Compiler

The interface program specification block (IPSB) compiler establishes the correspondences between the CA IDMS/DB database and the DL/I databases, as expected by the DL/I application.

The IPSB Compiler Accepts Source Statements

The compiler accepts as input the source statements produced by the CA IDMS DLI Transparency syntax generator, after you have verified and modified these statements as necessary. The compiler also uses the associated subschema load module.

The IPSB Compiler Produces IPSB Load Module

For output, the compiler produces IPSB load modules used by the CA IDMS DLI Transparency run-time interface. The IPSB load modules provide the information required to convert the application's DL/I database requests to CA IDMS/DB database requests. They also provide the control information required to update the application's DL/I program communication blocks (PCBs). The updated PCBs are used at run time to pass status information to the application program.

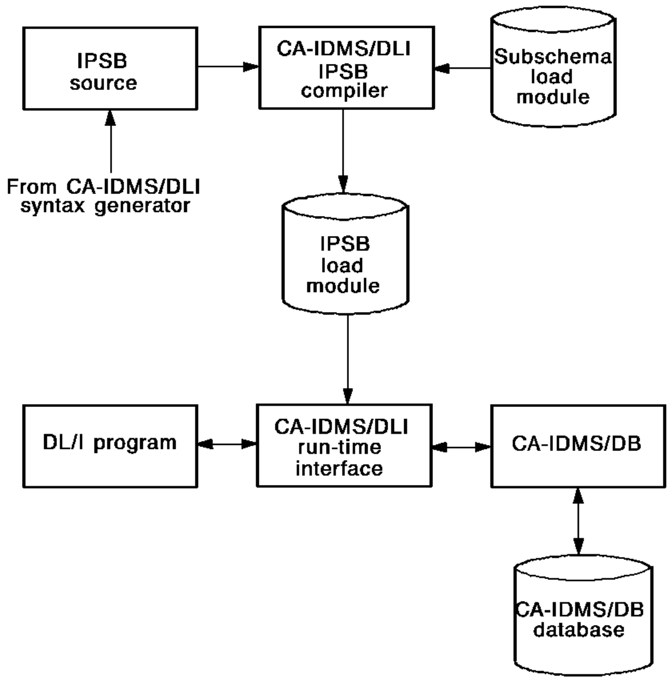


Figure 2. Role of the IPSB compiler in CA IDMS DLI Transparency

Run-Time Interface

What the Run-Time Interface Does

The CA IDMS DLI Transparency run-time interface accepts database calls from a DL/I application program, issues corresponding CA IDMS/DB calls, and returns data and/or status information to the DL/I application program. Note that a single DL/I call can result in several CA IDMS/DB requests. More specifically, CA IDMS DLI Transparency processing is divided between the interface's front-end and back-end processors.

Front-End Processor

The front-end processor intercepts DL/I requests from the application program, reformats the requests, and passes them to the back-end processor. When the back-end processor finishes with a request, it passes the results (data retrieved from the database and/or status information) back to the front-end processor. It also passes back PCB status information. The front-end processor then returns the status information to the DL/I application program.

Back-End Processor

Upon receiving a DL/I request from the front-end processor, the back-end processor accesses the IPSB load module to formulate the corresponding CA IDMS/DB requests. The back-end processor then passes the request to CA IDMS/DB. When CA IDMS/DB performs the requested operation(s), the back-end processor accepts the results from CA IDMS/DB and passes them, along with the PCB status information, to the front-end processor.

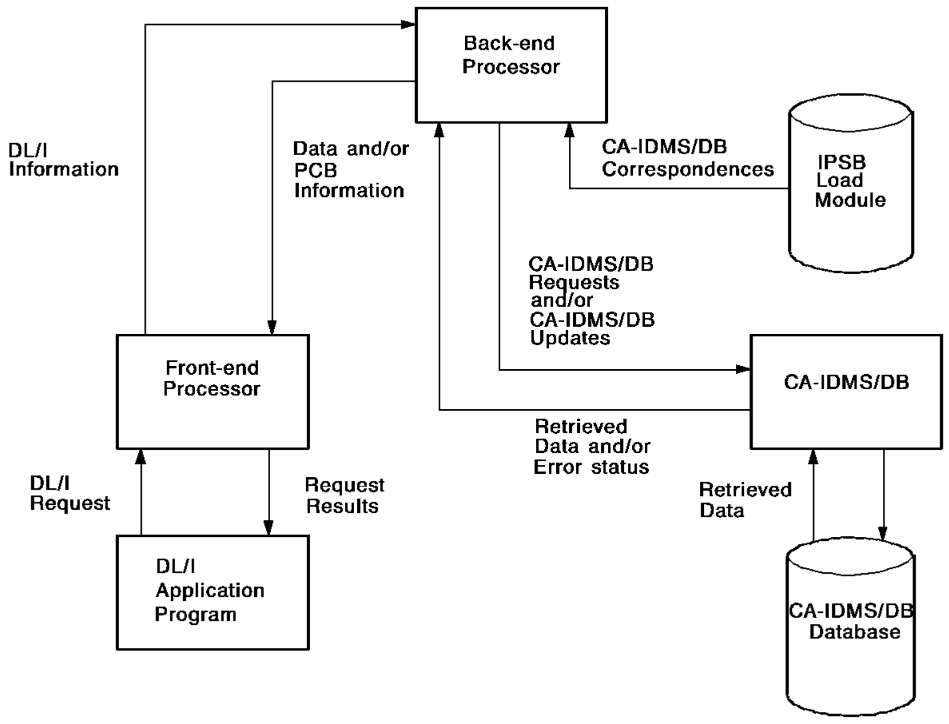


Figure 3. CA IDMS DLI Transparency runtime environment

The CA IDMS DLI Transparency Load Utility

What the Load Utility Does

The CA IDMS DLI Transparency load utility populates a CA IDMS/DB database with data unloaded from the existing DL/I database(s) used by the DL/I application.

Before You Run the Load Utility

Before you can run the load utility, you must have:

- An already created and initialized CA IDMS/DB database in which to receive the DL/I data. To do this, you must have created subschema and DMCL load modules for the database. These load modules are created by the appropriate CA IDMS/DB compilers when you input the schema, subschema, and DMCL source definitions produced by the CA IDMS DLI Transparency syntax generator.
- An IPSB load module for the CA IDMS/DB database. This load module is created by the IPSB compiler using the source statements produced by the CA IDMS DLI Transparency syntax generator.
- The unloaded DL/I database data, as formatted by the DL/I HD unload utility.

For output, the load utility stores the DL/I data in the CA IDMS/DB database in accordance with the supplied schema, subschema, DMCL, and IPSB load modules.

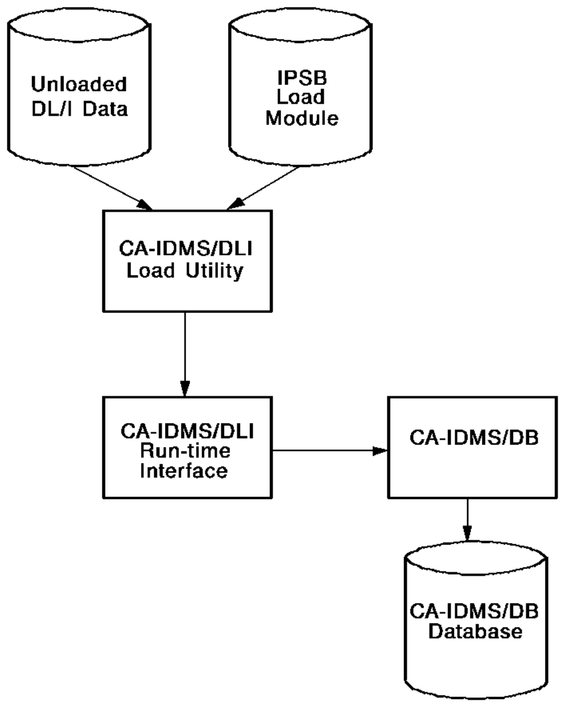


Figure 4. CA IDMS DLI Transparency load utility

Usage Requirements

Use of CA IDMS DLI Transparency involves the following six basic steps:

1. Assemble the source for your DL/I program specification block and database definitions using the CA-supplied macros. Input the assembled PSB and DBDs to the CA IDMS DLI Transparency syntax generator. The syntax generator produces IPSB source statements and the appropriate CA IDMS/DB schema, subschema, and DMCL source definitions. The use of the syntax generator is described in CA IDMS DLI Transparency Syntax Generator (see page 75).
2. Check the generated schema, subschema, and DMCL source definitions for compatibility with the DL/I definitions. Make any necessary changes and input the schema, subschema, and DMCL source definitions to the CA IDMS/DB compilers to produce the required load modules. DL/I, CA IDMS/DB and their correspondences are described in DL/I and CA IDMS/DB (see page 21).
3. Check the generated IPSB source statements for compatibility with the DL/I definitions. Make any necessary changes and input the IPSB source statements to the IPSB compiler to produce the IPSB load module, as described in IPSB Compiler (see page 93).
4. Create and initialize the new CA IDMS/DB database using the schema, subschema, and DMCL load modules from Step 2.
5. Load the DL/I data from the original database(s) into the new CA IDMS/DB database. Instructions for using the CA IDMS DLI Transparency load utility are provided in CA IDMS DLI Transparency Load Utility (see page 171).
6. Execute your DL/I application against the CA IDMS/DB database using the CA IDMS DLI Transparency run-time interface. The use of the run-time interface is described in CA IDMS DLI Transparency Run-Time Environment (see page 155).

Syntax Diagram Conventions

The syntax diagrams presented in this guide use the following notation conventions:

`UPPERCASE OR SPECIAL CHARACTERS`

Represents a required keyword, partial keyword, character, or symbol that must be entered completely as shown.

`lowercase`

Represents an optional keyword or partial keyword that, if used, must be entered completely as shown.

`italicized lowercase`

Represents a value that you supply.

`lowercase bold`

Represents a portion of the syntax shown in greater detail at the end of the syntax or elsewhere in the document.

```
◄─
```

Points to the default in a list of choices.

```
►►────────────────────
```

Indicates the beginning of a complete piece of syntax.

```
──────────────────►◄
```

Indicates the end of a complete piece of syntax.

```
────────────────────►
```

Indicates that the syntax continues on the next line.

```
►────────────────────
```

Indicates that the syntax continues on this line.

```
──────────────────►─
```

Indicates that the parameter continues on the next line.

```
─►──────────────────
```

Indicates that a parameter continues on this line.

```
►── parameter ────────►
```

Indicates a required parameter.

```
►──┬─ parameter ─┬─────►
   └─ parameter ─┘
```

Indicates a choice of required parameters. You must select one.

```
►──┬─────────────┬─────►
   └─ parameter ─┘
```

Indicates an optional parameter.

```
►──┬─────────────┬─────►
   ├─ parameter ─┤
   └─ parameter ─┘
```

Indicates a choice of optional parameters. Select one or none.

```
  ┌──────────────┐
►─▼─ parameter ──┴─────►
```

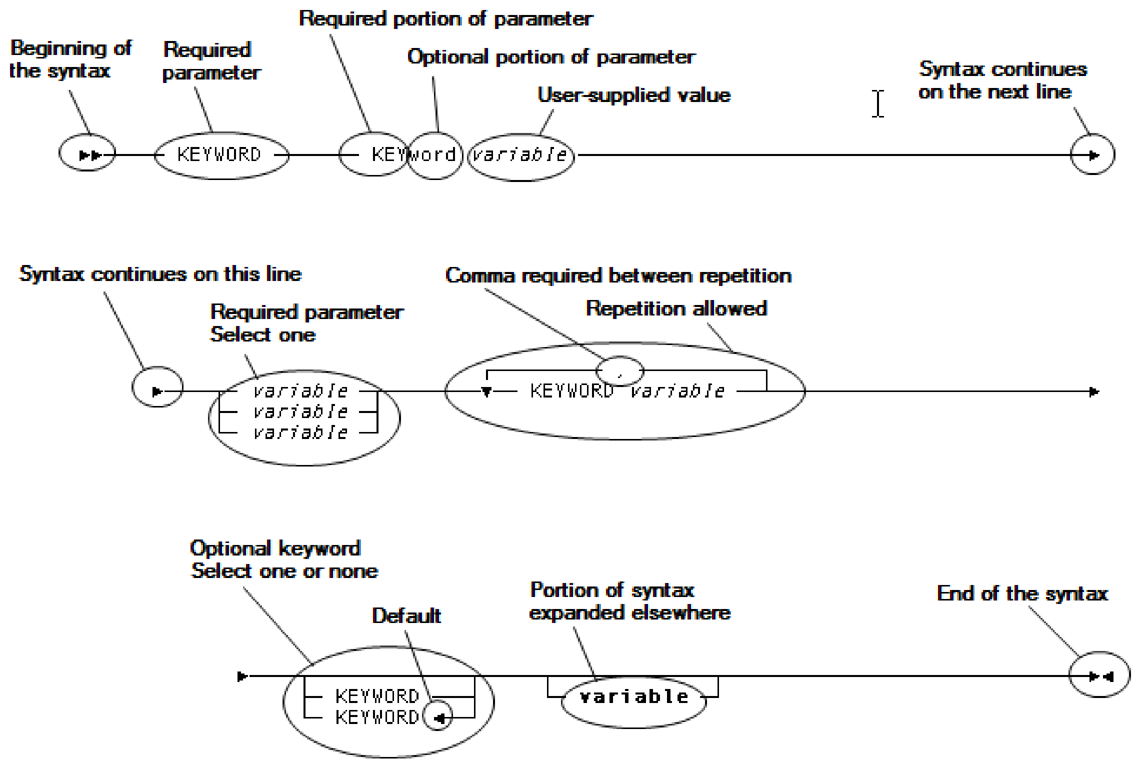
Indicates that you can repeat the parameter or specify more than one parameter.

```
  ┌─── , ─────────┐
►─▼─ parameter ───┴──────►
```

Indicates that you must enter a comma between repetitions of the parameter.

Sample Syntax Diagram

The following sample explains how the notation conventions are used:


Required portion of parameter
Beginning of the syntax
Required parameter
Optional portion of parameter
User-supplied value
Syntax continues on the next line
KEYWORD
KEYword
variable
Syntax continues on this line
Comma required between repetition
Required parameter Select one
Repetition allowed
variable
variable
variable
KEYWORD variable
,
Optional keyword Select one or none
Default
Portion of syntax expanded elsewhere
End of the syntax
KEYWORD
KEYWORD
variable

Chapter 2: DL/I and CA IDMS/DB

This section contains the following topics:

About This Chapter

As a DL/I database administrator (DBA) or application programmer, and CA IDMS/DB Correspondences/ you are already familiar with DL/I.

DL/I and CA IDMS/DB are similar in many ways. As database management systems, they both separate the logical definitions of data from the actual data as stored on disk. They both provide top-level definitions of the data and the relationships supported for the data. In addition, they provide second-level definitions that serve as application-specific **views** of the top-level definition.

This section describes DL/I, CA IDMS/DB and the correspondences between them.

The DL/I Environment

The Parent/Child Hierarchy

In DL/I, the basic structure is the parent/child hierarchy. A **parent segment** can own one or more **child segments**. (Segments are similar to records in conventional file-oriented, versus database-oriented, processing.) A child segment, however, can have only one parent segment. Using the basic parent/child structure, you can extend the hierarchy to deeper levels (that is, a child segment can also be a parent and have child segments of its own).

Database Description (DBD)

The top-level definition of the segments and their relationships is known as the **Database description** (DBD). A DBD defines all of the segments, the fields for each segment, and all of the possible segment relationships for a given database.

Program Specification Block (PSB)

The second-level definition is known as the **program specification block** (PSB). The PSB defines the run-time database interface for an application.

Program Communication Blocks

Each PSB contains one or more **program communication blocks** (PCBs). Each PCB defines a subset of the segments and possible relationships found in a specific DBD. Different PCBs within the same PSB can reference different DBDs or multiple views of the same DBD, thereby allowing an application to access several physical databases.

Each PCB also maintains status information so that the application can check on the results of its function calls against a particular database.

Taken collectively, the PCBs within a given PSB define an application's view of the available data.

Defining DL/I Databases

The database administrator defines DBDs and PSBs (including PCBs) using special source statements. The DBA then compiles the prepared source files using the DBDGEN and PSBGEN utilities. Finally, the compiled DBDs and PSBs are input to another utility that merges and expands them to produce an object-form control table for each PCB and DBD that it references.

Executing DL/I Applications

When DL/I is invoked, it loads the application's DBD and PCB control tables and passes control back to the application. The application is then ready to start issuing DL/I function calls for database operations.

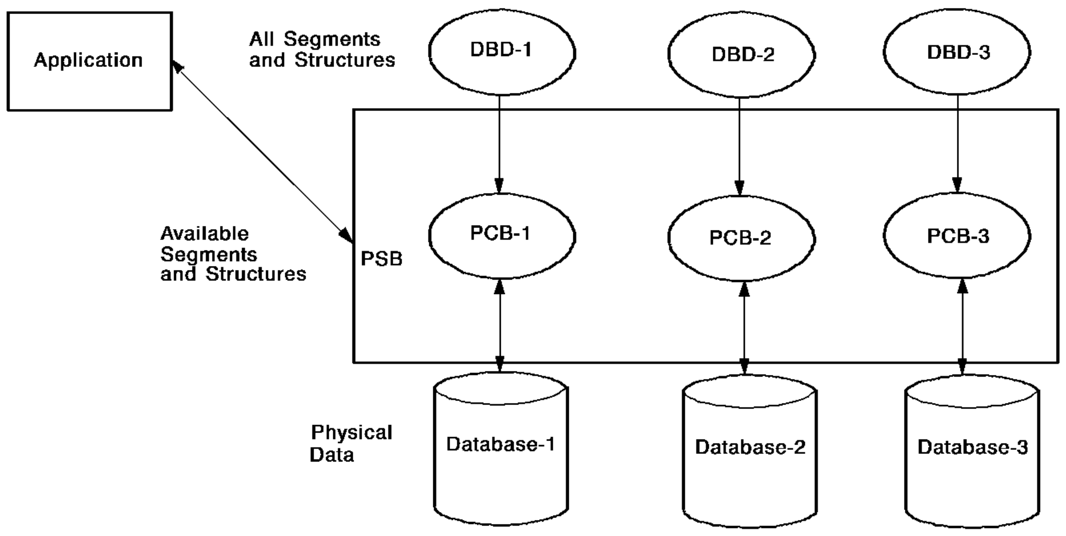


Figure 5. Basic DL/I components

Segments - The Basic Unit Of Data

What is a Segment

Segments are the basic units of data that an application can access in DL/I. Segments consist of one or more fields, which are the basic pieces of data that an application can use. For example, the EMPLOYEE segment might consist of the employee name, id, and address fields.

Segments can be either fixed length or variable length. Within a segment, individual fields can occur either once or multiple times.

What is a Segment Occurrence

A specific instance of a segment that is stored in the database is known as an **occurrence**. For example, the data for employee Bob Jones would be an occurrence of the EMPLOYEE segment. There can be any number of occurrences for a given segment.

Hierarchies - Physical Relationships Between Segments

What Hierarchical Relationships Do

In DL/I, segments are related physically in terms of parent/child hierarchies. These **hierarchical relationships** determine the physical organization of a database. They control how segments are stored in relation to each other. They also define the access paths for getting from one segment to another. In a hierarchical (physical) relationship, the parent segment is referred to as the **physical parent**, and the child segment is referred to as the **physical child**.

Parent and Child Segments

A parent segment can have zero, one, or more child segments, but a child segment can have only one parent. Each occurrence of a parent segment can have any number of occurrences of a dependent child segment. For example, if employee Bob Jones has two skills, there will be two occurrences of the SKILL child segment for the one occurrence of the EMPLOYEE parent segment.

Parent and Child Occurrences

A child occurrence requires an existing parent occurrence, but a parent occurrence does not require a child occurrence. Two or more child segment occurrences that have the same parent occurrence in a hierarchy are referred to as **physical twins**. Such occurrences are twins only in the sense that they have the same parent occurrence—not that they contain duplicate data.

Root Segments and Database Records

What is a Root Segment

In a DL/I hierarchical structure, the top-level parent segment is known as the **root segment**. There can be only one root segment in any hierarchy.

What is a Database Record

Collectively, all the parent/child occurrences that depend on a given root segment form a DL/I **database record**. Since there can be only one occurrence of a root segment, the addition of a new root segment occurrence (for example, a new employee) creates a new database record. Database records are variable in size because the number of occurrences for dependent child segments may vary (for example, new skills can be added for a given employee).

A DL/I Physical Database

All of the database records for a particular parent/child hierarchy form a DL/I **physical database**. Since each child segment can have only one parent segment, the resulting structure resembles an inverted tree, with the root segment at the top. The maximum number of segments in a DL/I structure is 255: one root and up to 254 dependent child segments.

Hierarchical Access Path

A DL/I Hierarchy

The basic parent/child structure is hierarchical in that it requires traversing higher levels to reach a specific lower level. In other words, to reach a given child segment occurrence, you must go from the root segment occurrence through all the intermediate parent segment occurrences. This path is known as a **hierarchical access path**. Hierarchical paths require that you traverse a structure in a top-to-bottom, left-to-right manner. There is a maximum of 15 levels (that is, 14 parent segments, including the root) in a DL/I hierarchical path.

The illustrations on the next few pages show different representations of the same DL/I hierarchy.

Physical Parent/Child Relationships

The illustration below illustrates the physical parent/child relationships among the segments. It is these physical relationships that define the hierarchy. The names of the segments are SEGA, SEGB, SEGC, and SEGD.

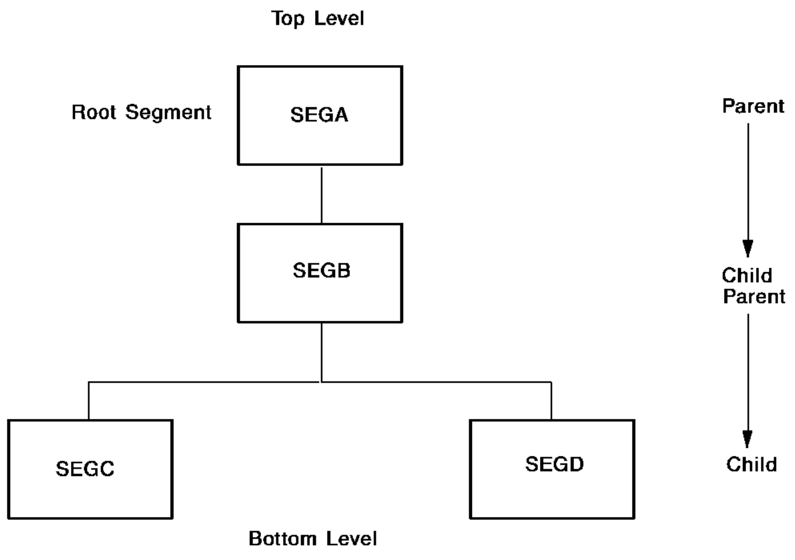


Figure 6. Physical segment relationships

DBD Source Statements For the Hierarchy

The sample below shows the Database Description (DBD) source statements used to define the hierarchy and the parent/child relationships among the segments.

```
DBD      NAME=DBD1,ACCESS=HDAM,RMNAME=(DLZHDC20,2,13000,4500)
DATASET  DD1=DBD1HDAM,DEVICE=3350,BLOCK=4096,SCAN=3
SEGM     NAME=SEGA,BYTES=31,PTR=H,PARENT=0
FIELD    NAME=(FIELDA,SEQ,U),BYTES=21,START=1
FIELD    NAME=FIELDB,BYTES=10,START=22
SEGM     NAME=SEGB,BYTES=30,PTR=H,PARENT=SEGA
FIELD    NAME=(FIELDC,SEQ,U),BYTES=30,START=1
SEGM     NAME=SEGC,BYTES=30,PTR=H,PARENT=SEGB
FIELD    NAME=(FIELDD,SEQ,U),BYTES=10,START=1
FIELD    NAME=FIELDE,BYTES=20,START=11
SEGM     NAME=SEGD,BYTES=60,PTR=H,PARENT=SEGB
FIELD    NAME=(FIELDF,SEQ,U),BYTES=10,START=1
FIELD    NAME=FIELDG,BYTES=50,START=11
DBDGEN
FINISH
END
```

Figure 7. DBD source statements for sample hierarchy

Hierarchy with Database Records

The illustration below shows a hierarchy with database records

Note that in the A1 record, segment SEGC has three occurrences. In the A2 record, segment SEGD has two occurrences. The hierarchical path to the D2b occurrence is by way of the following occurrences: A2, B2, C2, D2a (from top to bottom and left to right).

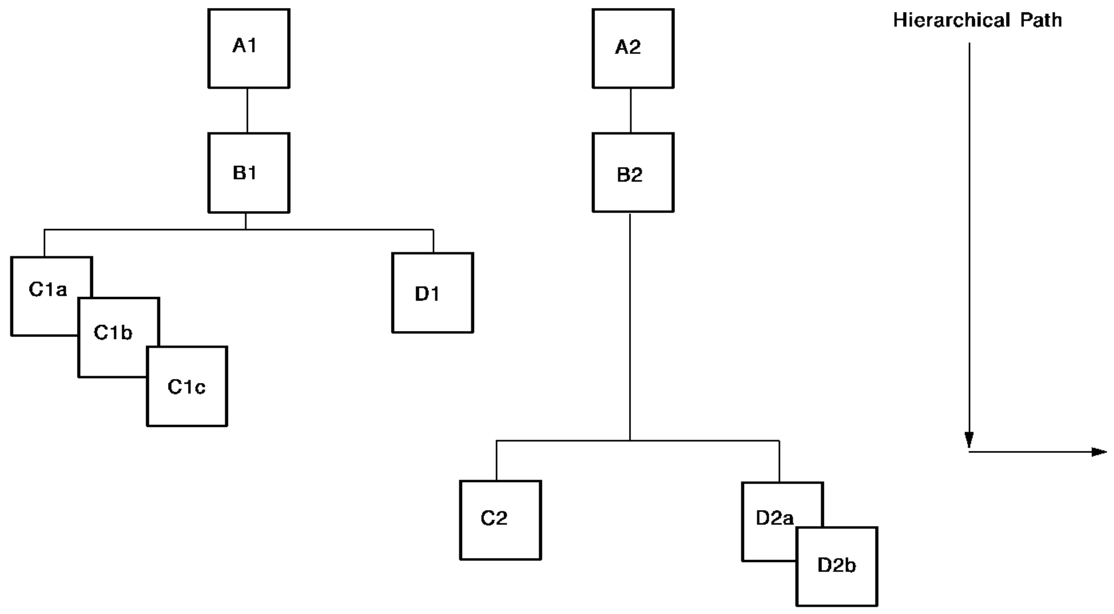


Figure 8. Hierarchy with database records

Defining Segments

A segment in DL/I is defined using a single SEGM statement and one or more FIELD statements.

SEGM Statement

The SEGM statement names and defines segments. For each child segment, the PARENT parameter specifies the name of the related parent segment. Note that the SEGM statement for SEGA (in Figure 7) specifies 0 (zero) for PARENT, indicating that this segment is the root (that is, it has no parent). The BYTES parameter specifies the length of each segment.

FIELD Statement

Each SEGM statement is followed immediately by one or more FIELD statements, which name and define the fields for the segment. An application can access the desired database records by specifying selection criteria for the segment fields. The application specifies the selection criteria in a **segment search argument** (SSA) on the appropriate function call. Only those records whose segment occurrences match the search criteria will be returned to the application.

Sequence Fields

If the NAME parameter on the FIELD statement contains the value SEQ, the field is a **sequence field**. A sequence field can have different functions depending on whether it is specified for a root segment or a dependent child segment. The differences are as follows:

- If specified for a root segment, a sequence field controls the physical placement of each root segment occurrence and provides direct access to the associated database record.
- If specified for a child segment, a sequence field causes occurrences of the segment to be stored in ascending order, based on the actual values in the sequence field.

  A sequence field for a child segment assumes that the segment can have more than one occurrence within a given parent occurrence (for example, C1a, C1b, and C1c in Figure 8). As the hierarchical path is traversed from right to left within the parent occurrence, the child occurrence with the lowest value will be found first, and the child occurrence with the highest value will be found last.

Unique or Duplicate Values in Sequence Fields

When defining child segments with sequence fields, you must also specify the value U or M in the NAME parameter. U declares that each occurrence's sequence field value must be unique under the same parent occurrence. M declares that multiple occurrences can have the same sequence field value under the same parent occurrence (that is, duplicate sequence field values are allowed).

Storage Sequence for Duplicate Values

If sequence fields have duplicate values, the RULES parameter for the SEGM statement lets you control how new occurrences of the child segment will be stored relative to existing occurrences under the same parent occurrence. The possible RULES values are:

- **FIRST**—Stores a new occurrence before all existing occurrences with the same value
- **LAST**—Stores a new occurrence after the existing occurrences
- **HERE**—Stores a new occurrence immediately before the current occurrence

Concatenated Keys

Concatenated keys provide an efficient way to access specific segment occurrences. Such a key is constructed by concatenating the value in an occurrence's sequence field with the values in the sequence fields from each higher level segment occurrence in the hierarchical path.

For example, using the hierarchical structure defined in Figure 7, the concatenated key for SEGC is made up of its own sequence field (FIELDD), the sequence field (FIELDC) for SEGB, and the sequence field (FIELDA) for SEGA. The key for a given SEGC occurrence would be determined by the actual values contained in the sequence fields.

Logical Relationships Between Segments

What Logical Relationships Do

Logical relationships provide a way of extending the basic hierarchical relationships. They have no effect on how segments are physically stored, but they do let you define multiple access paths to the same physical data. The segments defined in a logical relationship can be on the same hierarchical path or on different hierarchical paths.

Logical Parent and Logical Child

In a logical relationship, the parent segment is referred to as the **logical parent**, and the child segment is referred to as the **logical child**.

In a given logical relationship, a child segment can have only one physical parent and only one logical parent. Note that a parent segment can be both physical and logical parent to the same child segment. Also, the same child segment can have more than one logical parent, but in different logical relationships.

If two or more logical child segment occurrences have the same logical parent occurrence, they are referred to as **logical twins**. As with physical twins, they are twins only in the sense that they have the same parent occurrence.

Hierarchical (physical) relationships always occur within the same database. Logical relationships can occur within the same database or can involve segments from different databases.

DBD Source Statements for Two Databases

The example below shows sample DBD source statements for defining two databases (PHYSDB1 and PHYSDB2). Note that the DBD definitions define both hierarchical and logical relationships.

Each hierarchical relationship involves only segments that are in the same database. A logical relationship, though, can involve segments from its own database definition and segments from another database definition.

```
DBD      NAME=PHYSDBD1,ACCESS=HDAM
DATASET  DD1=HDAM1,DEVICE=3350,BLOCK=2048,SCAN=3
SEGM     NAME=SEG1,PTR=TWINBWD,RULES=LLV
FIELD    NAME=(FIELD1,SEQ,U),BYTES=60,START=1
FIELD    NAME=FIELD2,BYTES=15,START=61
FIELD    NAME=FIELD3,BYTES=75,START=76
LCHILD   NAME=(SEG6,PHYSDB2),PAIR=SEG2,PTR=DBLE
SEGM     NAME=SEG2,PARENT=SEG1,PTR=PAIRED
              SOURCE=(SEG6,DATA,PHYSDB2)
FIELD    NAME=(FIELD4,SEQ,U),BYTES=21,START=1
FIELD    NAME=FIELD5,BYTES=20,START=22
SEGM     NAME=SEG3,BYTES=200,PARENT=SEG1
FIELD    NAME=(FIELD6,SEQ,U),BYTES=99,START=1
FIELD    NAME=FIELD7,BYTES=101,START=100
SEGM     NAME=SEG4,BYTES=100,PARENT=SEG1
FIELD    NAME=(FIELD8,SEQ,U),BYTES=15,START=1
FIELD    NAME=FIELD9,BYTES=15,START=51
DBDGEN
FINISH
END


DBD      NAME=PHYSDBD2,ACCESS=HDAM,
              RMNAME=(DLZHDC20,7,700,250)
DATASET  DD1=HDAM2,DEVICE=3350,BLOCK=2048,SCAN=3
SEGM     NAME=SEG5,BYTES=31,PTR=TWINBWD,RULES=(VLV)
FIELD    NAME=(FIELD9,SEQ,U),BYTES=21,START,TYPE=P
FIELD    NAME=FIELD10,BYTES=10,START=22
SEGM     NAME=SEG6,
              PARENT=((SEG5,DBLE),(SEG1,P,PHYSDB1)),
              BYTES=80,PTR=(LPARNT,TWINBWD),RULES=VVV
FIELD    NAME=(FIELD11,SEQ,U),START=1,BYTES=60
FIELD    NAME=FIELD12,BYTES=20,START=61
SEGM     NAME=SEG7,BYTES=20,PTR=T,
              PARENT=(SEG6,SNGL)
FIELD    NAME=FIELD13,BYTES=9,START=1
FIELD    NAME=FIELD14,BYTES=11,START=10
SEGM     NAME=SEG8,BYTES=75,PTR=T,
              PARENT=(SEG6,SNGL)
FIELD    NAME=FIELD16,BYTES=50,START=1
FIELD    NAME=FIELD17,BYTES=25,START=51
DBDGEN
FINISH
END
```

Figure 9. DBD source statements for two databases

Three Types of Logical Relationships

DL/I supports three types of logical relationships:

- Unidirectional
- Bidirectional virtual
- Bidirectional physical

Unidirectional Relationship

Access Data in One Direction

In a unidirectional relationship, access can go in only one direction: from a logical child segment to its logical parent segment. A logical child segment cannot be accessed from its logical parent.

Unidirectional Structure

The illustration below illustrates the unidirectional logical structure. The structure shown involves segments from both of the physical hierarchies (PHYSDB1 and PHYSDB2) defined in earlier in this section. The logical child is SEG6 (in PHYSDB2), the physical parent is SEG5 (also in PHYSDB2), and the logical parent is SEG1 (in PHYSDB1).

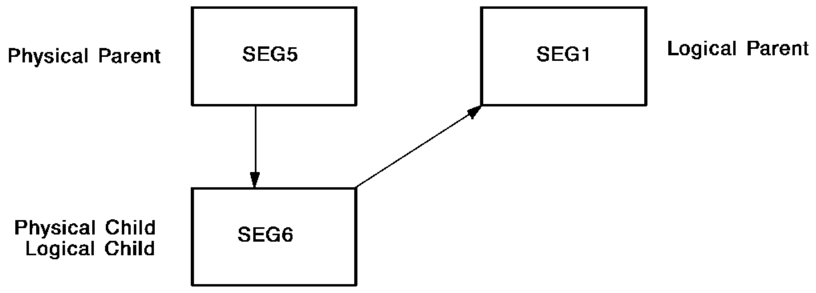


Figure 10. Unidirectional structure

Defining a Unidirectional Structure

You define a unidirectional structure in the logical child's SEGM statement. The SEGM statement names the logical child segment and identifies both the physical parent and the logical parent.

The PARENT parameter on the logical child's SEGM statement takes the following form:

Syntax

$$\text{PARENT} = \textit{(ppsegname2), (lpsegname,} \begin{Bmatrix} \text{VIRTUAL,} \\ \text{PHYSICAL} \end{Bmatrix} \textit{dbname)}$$

Parameters

ppsegname

Identifies the name of a physical parent segment and must match a name specified for the NAME parameter in a preceding SEGM statement.

lpsegname

Identifies the name of a logical parent segment and must match the name specified for the NAME parameter on the logical parent's SEGM statement. Note that this SEGM statement can be in the same DBD or a different DBD (see *Dbname* below).

VIRTUAL/PHYSICAL

Specifies whether the concatenated key of the logical parent is stored with the logical child (PHYSICAL) or is built at run time (VIRTUAL). For more details, see IPSB Compiler (see page 93).

dbname

Dbname is the name of the DBD that contains the logical parent's SEGM statement.

Bidirectional Virtual Relationship

Access Data in Two Directions

In a bidirectional virtual relationship, access can go in both directions: from a logical child segment to its logical parent segment, and from the logical parent segment to its logical child segment.

A bidirectional virtual relationship requires that you define a **virtual logical child segment**, as well as a **real logical child segment**. The virtual logical child is a pointer to the real logical child. (Compare to the bidirectional physical relationship, described below, in which the virtual logical child is a physical duplicate of the real logical child.)

Unidirectional relationships involve three segments; bidirectional relationships always involve four segments.

Bidirectional Virtual Structure

The example below shows the bidirectional virtual relationship defined by the DBD source statements shown in Figure 9 earlier in this section. In this relationship, SEG6 is the real logical child, SEG5 is the physical parent, SEG1 is the logical parent, and SEG2 is the virtual logical child. Note that SEG5 and SEG6 are in DBD PHYSDB2, and SEG1 and SEG2 are in DBD PHYSDB1.

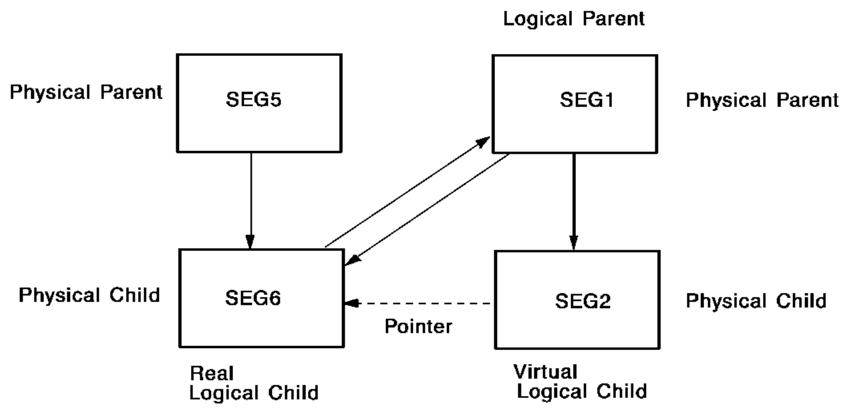


Figure 12. Bidirectional virtual structure

Defining the Virtual Logical Child

The physical parent, the physical child, the logical parent, and the real logical child are defined the same as for a unidirectional relationship (see Unidirectional Relationship (see page 32)). You define the virtual logical child in two places:

- In the logical parent's LCHILD statement. This statement follows the logical parent's SEGM and FIELD statements. It supplies the name of the real logical child segment and identifies the DBD in which it is defined. It also supplies the name of the segment in the logical parent's DBD that is to serve as the virtual logical child.
- In the virtual logical child's SEGM statement. The virtual logical child must be defined in the same DBD as the logical parent.

SEGM Statement for the Virtual Logical Child

The virtual logical child's SEGM statement must include the SOURCE parameter, which sets up a pointer to the real logical child and takes the following form:

Syntax

SOURCE=((*segname*,DATA,*dbname*))

Parameters

segname

Identifies the name of the real logical child segment, as specified for the NAME parameter in the real logical child's SEGM statement.

dbname

Dbname is the name of the DBD that contains the real logical child's SEGM statement.

Bidirectional Physical Relationship

What is a Bidirectional Physical Relationship

Bidirectional physical relationships provide access in both directions between a logical parent segment and a logical child segment. In this respect, they are the same as bidirectional virtual relationships. The difference between the two types of relationships is that bidirectional physical employs a physical duplicate of the real logical child, while bidirectional virtual employs a pointer to the real logical child, with no duplication of data.

Using Physical or Logical Virtual Bidirectional Relationships

The decision to use one type of bidirectional relationship instead of another depends on whether you want to optimize performance or space usage. Bidirectional physical relationships provide faster access times, but incur more space overhead because of the duplicate logical child data. They also require more maintenance overhead since updates made to one logical child must be duplicated in the other. Bidirectional virtual relationships conserve on space, but provide slower access times.

Bidirectional Physical Structure

The illustration below shows the bidirectional physical relationship defined by the DBD source statements in Figure 7. In this relationship, SEG6 is a physical child for SEG5 and a logical child for SEG1, SEG4 is a physical child for SEG1 and a logical child for SEG5. Note that SEG6 and SEG5 are in DBD PHYSDB2, and SEG4 and SEG1 are in DBD PHYSDB1.

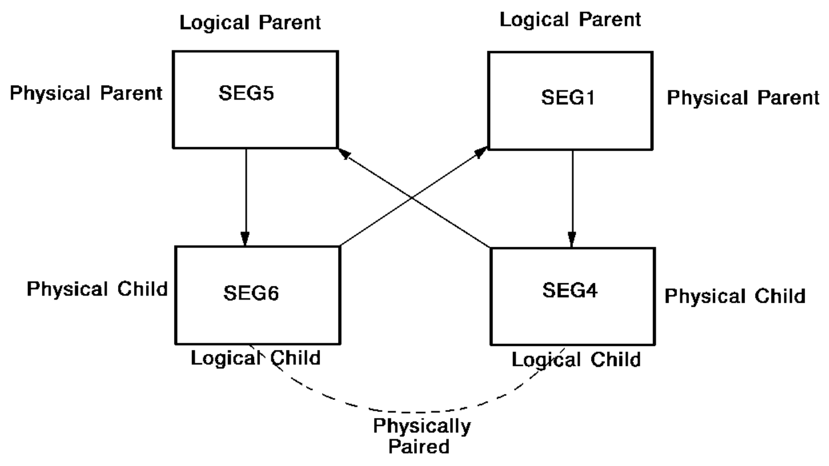


Figure 12. Bidirectional physical structure

Defining a Bidirectional Physical Relationship

To create a bidirectional physical relationship, you must define a child segment as both physical child and logical child for each parent, in each parent's physical hierarchy. In effect, you define the same unidirectional structure for each parent. The two logical child segments contain duplicate data and together are referred to as physically paired logical child segments. Note that the logical child SEGM statements cannot include the SOURCE parameter.

Physical Databases

A Physical Database is a DBD Definition

In DL/I, a **physical database** is a DBD definition that specifies the allowable segments, segment fields, and segment relationships for an actual database as stored on disk. Such a definition is known as a **physical DBD**. The term "physical" in this context is somewhat misleading because the DBD serves as the top-level logical definition (or template) for the database. All of the DBD definitions examined thus far are examples of physical databases, even though they define logical as well as hierarchical relationships.

What is a Physical DBD

A physical DBD maps the definition of segments and their hierarchical relationships to physical storage. The sequence in which the segments are defined in the DBD determines how their occurrences will be stored on disk. The hierarchical relationships determine the access path that must be navigated to reach a specific segment occurrence.

A Physical DBD Specifies an Access Method

In addition to defining segments and their relationships, a physical DBD specifies the physical data organization to be used and the corresponding **access method**. DL/I provides four physical access methods: HDAM, HISAM, HIDAM, and HSAM. The choice of access method is the responsibility of the database designer and depends on the contents of the database and the transaction load requirements. The choice of access method is described in more detail under Physical Access Methods (see page 38).

Sample DBD Statement

Physical DBDs can be easily identified because they specify one of the four access methods for the ACCESS parameter in the DBD statement. For example, the DBD for PHYSDB1 in Figure 9 is a physical DBD. The DBD statement is as follows:

```
DBD NAME = PHYSDB1,ACCESS=HDAM
```

The diagram below shows the physical database (hierarchy) derived from the DBD source statements in Figure 7.

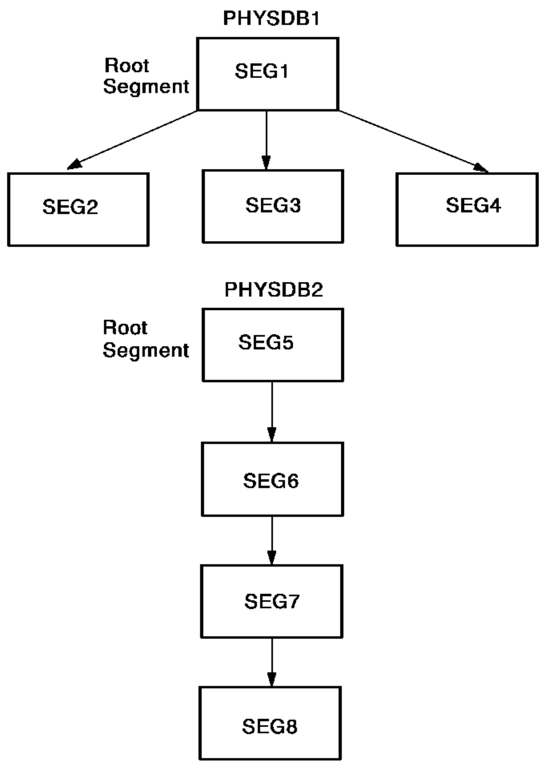


Figure 13. Sample physical databases

Physical Access Methods

What Physical Access Methods Do

Physical access methods determine the physical organization and available access paths for DL/I databases. Each physical DBD must be assigned an access method, which is specified for the ACCESS parameter in the DBD statement.

Sequential and Direct Access Methods

DL/I provides two general access methods: sequential and direct. The sequential method lays out the segment occurrences as physically contiguous, like records in a tape file. The direct method provides random access via pointers to segment occurrences, like records on a direct access storage device (disk). Each method is further qualified on the basis of whether or not it supports indexing.

DL/I Supports Four Access Methods

The combination of sequential/direct and indexing/no indexing yields the following four access methods for DL/I:

- HSAM––Hierarchical sequential access method
- HISAM––Hierarchical indexed sequential access method
- HDAM––Hierarchical direct access method
- HIDAM––Hierarchical indexed direct access method

Note that all four access methods are hierarchical (H). This reflects the fact that an application always views a database as hierarchical, regardless of the access method used or the physical location of the data.

HSAM Access

What HSAM Provides

The HSAM access method provides sequential access to root segments and child segments. The top-to-bottom, left-to-right hierarchical sequence is reflected in the physical contiguity of the database records.

Use HSAM for Sequential File Processing

The HSAM organization requires fixed-length records and is intended exclusively for conventional sequential file processing. There is no provision for making updates in place, without copying the database. Also, HSAM supports only hierarchical relationships, not logical relationships.

For more information about HSAM access method, see DL/I Access Methods in CA IDMS/DB (see page 65).

HISAM Access

What HISAM Provides

The HISAM access method provides indexed access to root segments and sequential access to child segments. The index contains the root segment sequence field values and is maintained in ascending order as part of the physical database.

As with the HSAM method, the hierarchical relationships are reflected in the physical contiguity of the database records.

HISAM uses two data sets: the primary data set and the overflow data set. Both data sets are defined with fixed-length physical records.

Primary Data Set

The **primary data set** contains the root segment occurrences and as many of their dependent segment occurrences as will fit. The primary data set supports indexing via the root segment sequence field values.

Overflow Data Set

The **overflow data set** contains the dependent occurrences that will not fit in the primary data set. Chains between the primary and overflow data sets maintain relationships and sequencing.

HISAM supports hierarchical relationships and unidirectional and bidirectional logical relationships with physical pairing. HISAM does not support bidirectional virtual relationships.

HDAM Access

What HDAM Provides

The HDAM access method provides hashed access to root segments and pointer access to child segments. The hashing algorithm calculates the physical address of a root segment occurrence based on the value in its sequence field.

HDAM Uses a Radomizing Routine

When a database record is first loaded, the HDAM method randomizes the root key value to a physical location, which consists of a block number and an offset into the block. The root segment occurrence and all dependent segment occurrences that will fit are loaded into the block. Dependent segment occurrences that will not fit are loaded into an overflow area. Physical child and physical twin pointers are created to establish the appropriate connections.

Fast and Direct Access to Root Segments

HDAM provides fast, direct access to a root segment occurrence. With, at most, one additional I/O, it is possible to access the first occurrence of the dependent segment at the next level by following the appropriate physical child pointer.

The HDAM method supports all of the DL/I hierarchical and logical relationships.

HIDAM Access

What HIDAM Provides

The HIDAM access method provides indexed access to root segments, via the root sequence field, and pointer access to child segments. The index contains the root segment sequence field values and is maintained in ascending order.

A HIDAM database is made up of two separate databases. One database contains all of the data. The other database is the index and contains the sequence field values for the root segment occurrences.

Index Database

The index database is never visible to an application, but it must be defined in its own set of DBD statements. A HIDAM index database requires the value INDEX for the ACCESS parameter in the DBD statement. The illustration below shows the DBD source statements for a HIDAM physical database (DB1) and its associated index database (DBINDEX).

```
DBD      NAME=DB1,ACCESS=HIDAM
DATASET  DD1=DBHIDAM,DEVICE=3350,BLOCK=42,RECORD=48,SCAN=1

SEGM     NAME=SEG1,BYTES=31,PTR=H,PARENT=0

FIELD    NAME=(FIELD1,SEQ,U),BYTES=21,START=1
FIELD    NAME=FIELD2,BYTES=10,START=22
LCHILD   NAME=(SEG2,DBINDEX),PTR=INDX
DBDGEN
FINISH
END



DBD      NAME=DBINDEX,ACCESS=INDEX
DATASET  DD1=DBINDEX,DEVICE=3350,BLOCK=44,RECORD=46,SCAN=1
SEGM     NAME=SEG2,BYTES=21

LCHILD   NAME=(SEG1,DB1),INDEX=FIELD1

FIELD    NAME=(FIELD3,SEQ,U),BYTES=21,START=1
DBDGEN
FINISH
END
```

Figure 14. DBD definitions for a HIDAM database and its index database

Index Pointer Segments

An index database can contain only one segment, which is referred to as the **index pointer segment**. The single SEGM statement in the index DBD names this segment. The index pointer segment points to the root segment in the physical DBD. The root segment is referred to as the **source segment** because it is the source of the data needed to construct the index pointer segment. The root segment is also the **target segment** because it is the segment that will be accessed by the index pointer. The index pointer segment contains one field, which will carry the sequence field values for the root segment occurrences. This field must also be defined as a sequence field.

LCHILD Statement Associates Databases

The physical (HIDAM) DBD and the index DBD both contain an LCHILD statement. Together, the two LCHILD statements establish the association between the databases. The NAME parameter in the physical DBD's LCHILD statement specifies the index pointer segment and the index DBD in which it is defined. The NAME parameter in the index DBD's LCHILD statement specifies the root segment and the physical DBD in which it is defined. The INDEX parameter in the index DBD's LCHILD statement specifies the sequence field in the named root segment.

The HIDAM method supports all of the DL/I hierarchical and logical relationships.

Secondary Indexing (Index Databases)

What is a Secondary Index

A **secondary index** defines an alternative (or secondary) access path that overrides the underlying hierarchical access path. DL/I supports the following types of secondary indexes:

- An index to a root or dependent segment on the basis of any field in the segment
- An index to a root or dependent segment on the basis of any field in a physically dependent segment

The key field for an index can be a single field or up to five fields in the same segment concatenated in any order. A physical database can have multiple secondary indexes.

Define Secondary Index as a Separate Database

Secondary indexes must be defined as separate databases. The segment occurrences in a secondary index database contain the values of the specified key field(s) and the pointers to the associated segment occurrences in the physical database. The secondary index segment is known as the **pointer segment**. The segment containing the key field(s) is known as the **source segment** and the segment to be accessed is known as the **target segment**. The source and target segments can be the same or different.

Secondary indexes differ from HIDAM indexes in that they allow you to index segments other than root segments. In a secondary index, the pointer segment can contain up to five concatenated fields, rather than just one field. Also, the source and target segments do not have to be the same.

Defining Secondary Indexes

Secondary indexes are defined in a manner similar to HIDAM index databases. Related statements must be included in both the index DBD and the associated physical DBD.

Sample DBD Definitions

The sample below shows the DBD definitions for a physical HDAM database (DB2) and an associated secondary index database (DBINDX2).

```
DBD      NAME=DB2,ACCESS=HDAM,
              RMNAME=(GLDHDC20,5,660,850)
DATASET  DD1=DBHDAM,DEVICE=3350,BLOCK=2048,SCAN=1

SEGM     NAME=SEG1,PARENT=0,BYTES=15

FIELD    NAME=(FIELD1,SEQ,U),BYTES=5,START=1
LCHILD   NAME=(SEG6,DBINDX2),PTR=INDX
XDFLD    NAME=XDFLD1,SEGMENT=SEG2,
              SRCH=FIELD2,DDATA=FIELD3

SEGM     NAME=SEG2,PARENT=SEG1,BYTES=25

FIELD    NAME=(FIELD2,SEQ,U),BYTES=5,START=1
FIELD    NAME=(FIELD3),BYTES=10,START=6
SEGM     NAME=SEG3,PARENT=SEG2,BYTES=15
FIELD    NAME=(FIELD4,SEQ,U),BYTES=10,START=1
SEGM     NAME=SEG4,PARENT=SEG2,BYTES=30
FIELD    NAME=(FIELD5,SEQ,U),BYTES=20,START=1
DBDGEN
FINISH
END



DBD      NAME=DBINDX2,ACCESS=INDEX
DATASET  DD1=INDX2,DEVICE=3350,BLOCK=23,
              RECORD=88,SCAN=1

SEGM     NAME=SEG6,PARENT=0,BYTES=15

FIELD    NAME=(FIELD6,SEQ,U),START=1,BYTES=15
LCHILD   NAME=(SEG1,DB2),POINTER=SINGL,INDEX=XDFLD1
DBDGEN
FINISH
END
```

Figure 15. DBD definitions for a physical and secondary database

Index DBD Statements

The **index DBD** must contain the following statements:

- **DBD statement** –– ACCESS parameter must specify INDEX.
- **SEGM statement** –– Defines the index pointer segment as the root for the index database. Only one SEGM statement is allowed.
- **FIELD statement** –– Defines the sequence field for the pointer segment. Only one FIELD statement is allowed.
- **LCHILD statement** –– Identifies the target segment and the physical DBD in which it is defined. The INDEX parameter specifies the name of the indexed field (XDFLD) in the associated physical DBD. Only one LCHILD statement is allowed.

Physical DBD Statements

The **physical DBD** must contain the following statements:

- **DBD statement** –– ACCESS parameter must specify HISAM, HDAM, or HIDAM. While it is possible to set up a secondary index for a logical database (ACCESS=LOGICAL), it is not recommended for reasons of performance and data independence. HSAM databases are restricted to sequential access.
- **LCHILD statement** –– Identifies the pointer segment and the index DBD in which it is defined. The PTR (pointer) parameter must specify INDX (for index). The LCHILD statement must be included under the SEGM statement for the target segment.
- **XDFLD statement** –– Identifies a source segment and its index field. The value for the NAME parameter is referenced in the LCHILD statement in the index DBD. The SEGMENT parameter specifies the source segment. The SRCH parameter specifies the sequence field in the source segment to be used for indexing. The DDATA parameter specifies a data field in the source segment to be used for indexing.

  Note that the values for the SRCH and DDATA fields will be concatenated to produce the actual index-key field values. The XDFLD statement must be included under the SEGM statement for the target segment.

Restructuring a Hierarchy

If a secondary index points to a dependent segment, the effect is to restructure the hierarchy so that the dependent segment appears as the root. In the new hierarchy, the higher level segments in the original hierarchy become dependents of the new root. They appear as leftmost dependents in reverse hierarchical order. A secondary index is similar to a logical relationship in that they both restructure an underlying hierarchy. However, a secondary index is different from a logical relationship in that it can deal only with a single physical database. Logical relationships can combine segments from one or more physical databases.

The diagram below illustrates an original, underlying hierarchy and the new hierarchy that results from indexing a dependent segment (SEG2).

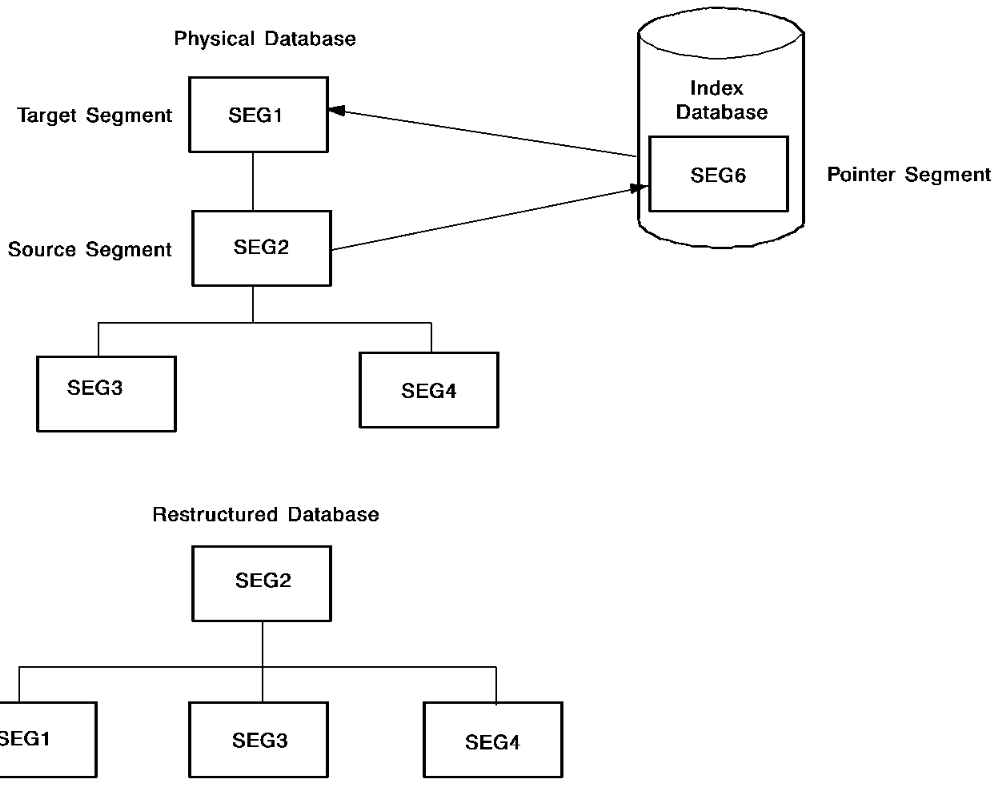


Figure 16. Hierarchical restructuring via a secondary index

Full and Sparse Indexing

A secondary index can be either full or sparse.

Full Index

A **full index** maintains an entry for each source segment occurrence in which the search field has a value.

Sparse Index

A **sparse index** maintains entries only for selected values in the search field. Because sparse indexing is more selective than full indexing, it provides faster search times for the desired target segments.

Logical Databases

What is a Logical Database

A **logical database** is a DBD definition that references structures already defined in one or more physical databases (physical DBDs). Such a definition is known as a **logical DBD**.

To an application, a logical DBD always appears as a single hierarchical physical DBD. However, a logical DBD is derived from the relationships (especially the logical relationships) defined in the associated physical DBDs.

Logical Databases Provide Flexibility

Logical databases provide flexibility for applications by allowing them to view the same physical data in many different ways. It is important to remember that each logical DBD is still hierarchical in nature for all of the DL/I calls that use it.

Defining a Logical Database

Specify LOGICAL for ACCESS Parameter

To define a logical DBD, you must specify LOGICAL for the ACCESS parameter in the DBD statement. The bulk of a logical DBD consists of SEGM statements that reference segments defined in one or more physical DBDs. (Segment fields can be defined only on the physical DBD level.)

SEGM Statement

The SEGM statement specifies a NAME for the segment and must contain a SOURCE clause to identify the segment as defined in a physical DBD. Similar to physical DBDs, the PARENT parameter specifies the parent segment within the logical structure. For example, the statement below declares that segment SEG7 is based on the segment of the same name in the PHYSDB2 physical DBD, and is a child of the LSEGB segment in this logical database:

Syntax

```
SEGM NAME=SEG7,SOURCE=((SEG7,PHYSDB2)),PARENT=LSEGB
```

Intersection and Concatenated Segments

Pointer and Target Segments

As mentioned above, logical databases are defined by referencing segments already defined in one or more physical DBDs. In particular, logical databases rely on the logical relationships defined in the physical DBDs. Logical relationships allow you to link a segment (logical child) in one physical DBD with a segment (logical parent) in another (or the same) physical DBD. In such a relationship, the logical child segment is referred to as the pointer segment, and the logical parent is referred to as the target segment.

Intersection and Concatenated Segments

In practice, the link between pointer and target segments is established via a pointer field in the pointer (logical child) segment. If the pointer segment contains data fields in addition to the pointer field, such fields are said to carry **intersection data** and the segment itself is referred to as an **intersection segment**. The intersection data is unique to the relationship between a pointer segment occurrence and its associated target segment occurrence. An application can retrieve and modify the data portions of the pointer and target segments separately, or it can retrieve and modify the pointer and target segments as one **concatenated segment**.

Defining a Concatenated Segment

The definition of individual pointer (logical child) and target (logical parent) segments occurs at the physical DBD level. The definition of concatenated segments occurs at the logical DBD level and is specified via the SOURCE parameter in the SEGM statement. The SOURCE parameter determines the contents of a concatenated segment, which can be:

- The concatenated key of the destination parent, the pointer segment's intersection data, and the destination parent's data
- The concatenated key of the destination parent and the pointer segment's intersection data
- The destination parent's data only

The Destination Parent

The **destination parent** can be either the physical or logical parent of the pointer (logical child) segment. The choice depends on the direction in which you want the access to proceed: from logical parent to physical parent via the logical child, or from physical parent to logical parent via the logical child.

Syntax

The SOURCE parameter in the logical DBD SEGM statement takes the following form:

$$\text{SOURCE} = ((\textit{psegname}), \left\{ \begin{matrix} \text{KEY,} \\ \text{DATA} \end{matrix} \right\} \textit{dbname1}), (\textit{dsegname}), \left\{ \begin{matrix} \text{KEY,} \\ \text{DATA} \end{matrix} \right\} \textit{dbname2}))$$

Parameters

psegname

Identifies the name of the pointer (logical child) segment as defined in the physical DBD *dbname1*. This segment can be either a virtual logical child or a real logical child. (See "Bidirectional Virtual Relationship" and "Bidirectional Physical Relationship" earlier in this section.)

KEY/DATA

KEY/DATA specifies whether an application will have access to only the key (sequence field) of the named segment, or will have access to the segment's data portion as well as the key. KEY is the default.

dsegname

Dsegname is the name of the destination parent as defined in the physical DBD *dbname2*. The destination parent can be either the physical or logical parent for the pointer (logical child) segment named in *psegname*.

Sample Logical Database

DBD Definition for a Logical Database

The sample below shows the DBD source statements for a logical database. The DBD definitions for the underlying physical databases are those shown in Figure 9. In the logical DBD shown below, LSEGB is the concatenated segment that combines the SEG6 and SEG1 segments from PHYSDB1 and PHYSDB2, respectively.

```
DBD      NAME=LOGDB,ACCESS=LOGICAL

DATASET  LOGICAL
SEGM     NAME=LSEGA,SOURCE=((SEG5,PHYSDB2))
SEGM     NAME=LSEGB,PARENT=LSEGA,
              SOURCE=((SEG6,DATA,PHYSDB2),(SEG1,DATA,PHYSDB1))
SEGM     NAME=SEG3,PARENT=(LSEGB,((SEG3,PHYSDB1)))
SEGM     NAME=SEG4,PARENT=LSEGB,SOURCE=((SEG4,PHYSDB1))
SEGM     NAME=SEG7,SOURCE=((SEG7,PHYSDB2)),PARENT=LSEGB
SEGM     NAME=SEG8,SOURCE=((SEG8,PHYSDB2)),PARENT=LSEGB
DBDGEN
FINISH
END
```

Logical Database Structure

The illustration below shows the logical database produced by the logical DBD definition in the source statements shown above. Compare the resulting logical structure with the hierarchical structures for the underlying physical databases.

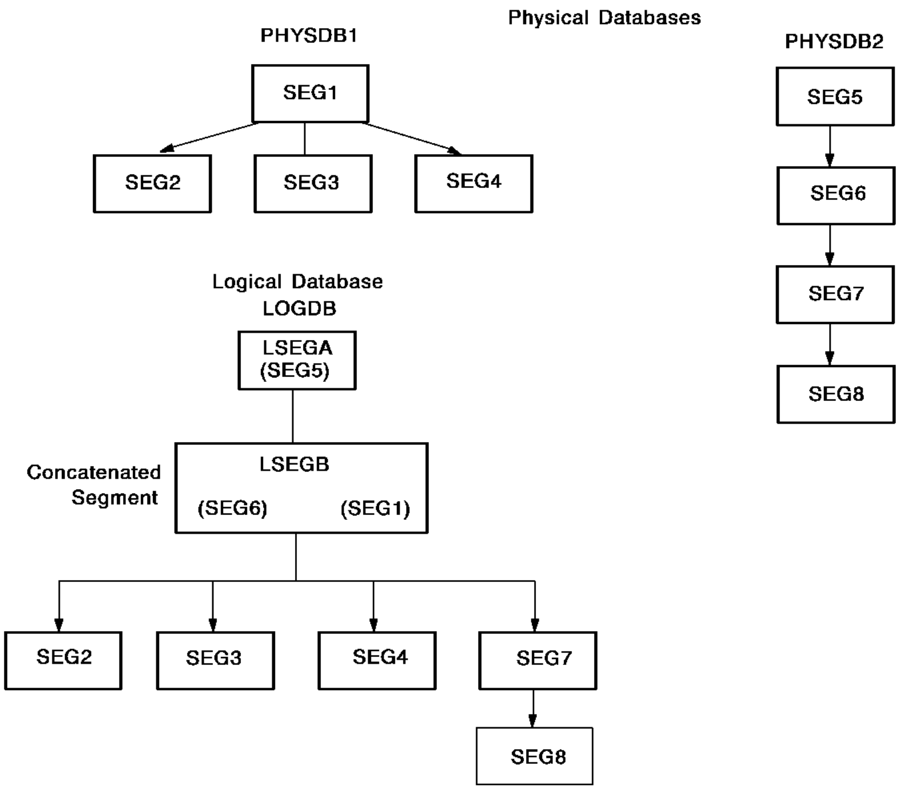


Figure 17. Logical database structure

Program Communication Blocks

What the Program Communication Block (PCB) Does

A **program communication block** (PCB) selects segments from a specific physical or logical DBD. An application using the PCB will have access to only those segments that are selected. Usually, a PCB selects only a subset (or subhierarchy) of the segments defined in a DBD, but it *can* select all of the segments.

Multiple PCBs

Multiple PCBs can be defined for the same DBD, each selecting a different subset of the defined segments. PCBs can overlap so that the same segment(s) can appear in different PCBs. Multiple applications can share the same PCB, but via different program specification blocks (described later in this section).

Data Sensitivity and the PROCOPT Options

What is Data Sensitivity

In DL/I, an application's **data sensitivity** refers to those segments that are available to the application via the PCBs it uses. In terms of data sensitivity, the basic purpose of a PCB is to effectively mask out segments from an application.

PROCOPT Processing Options

The PROCOPT processing options provide a number of access controls in addition to the basic access control based on including or excluding a segment. PROCOPT options let you further qualify access to specified segments. For example, PROCOPT=G permits the program to GET (that is, read) a segment. Some PROCOPT options can also be specified for the entire DBD, thereby restricting access on the database level itself.

The PROCOPT options include:

- **G** — Get (retrieve) access
- **R** — Replace (update) access
- **I** — Insert access (to store new segments)
- **D** — Delete access
- **P** — Path calls
- **O** — Get calls only (no hold)
- **A** — Any or all of the access options above
- **L** — Load access (for database loading)
- *x***S** — Ascending sequence only for the option indicated by *x* (G, R, etc.)
- **K** — Key access only

Multiple options can be specified in the same PROCOPT parameter.

The K Option

The K option allows a PCB to restrict an application to only the key portion of a segment, while masking out the data portion. The K option is important because it removes access to a segment but still retains the hierarchical access path to the segment's dependents. By default, when a PCB masks out a segment, it also masks out the segment's dependent segments. The K option provides a way around this restriction.

Defining a PCB

Sample PCB

The example below shows the source statements for a sample PCB

```
PCB      TYPE=DB,DBDNAME=DBDNEW,PROCOPT=G,
              KEYLEN=45,PROCSEQ=INDEX1
SENSEG   NAME=SEGRT1,PARENT=0
SENSEG   NAME=SEG3,PARENT=SEGRT1
SENSEG   NAME=SEG4,PARENT=SEG3
SENSEG   NAME=SEG2,PARENT=SEGRT1
PSBGEN   LANG=COBOL,PSBNAME=PSB1
END
```

Figure 18. Sample PCB definition

The PCB Statement

To define a PCB, you must first specify the PCB statement. On the PCB statement, the DBDNAME parameter identifies a physical or logical DBD from which to select segments. The PROCOPT parameter specifies a PROCOPT option for the entire database.

The KEYLEN Parameter

The KEYLEN parameter specifies the maximum key length to be used when the key (sequence field value) of a segment is concatenated with the keys of the higher-level segments in its hierarchical access path. The KEYLEN value is determined by adding up the lengths of the sequence fields necessary to reach the lowest-level segment in the hierarchy of available segments.

Sensitive Segment (SENSEG) Statements

The bulk of a PCB definition consists of SENSEG (sensitive segment) statements. Each SENSEG statement specifies a segment to be included from the named DBD. The SENSEG statement can also include a PROCOPT option for the segment.

Program Specification Block

The **program specification block** (PSB) defines all of the database views that are available to an application. A PSB consists of one or more PCB definitions similar to the one shown in Figure 18. One PSB can contain up to 255 separate PCBs.

For each PCB you define, you must include the PSBGEN statement. The PSBGEN statement names the PSB and specifies the language in which the current applications are written.

Parallel Processing

If a PSB contains multiple PCBs, an application using the PSB can engage in parallel processing. Since each PCB can reference a separate DBD, the application, by way of multitasking, can perform parallel processing on different databases or on different views of the same database. DL/I maintains a separate PCB control block for each database in use.

Definition Summary

The DL/I process of defining databases and application views of these databases involves the following steps:

1. **Define one or more physical databases** using DBD source statements. The physical DBDs specify segments, fields within segments, and the hierarchical relationships among segments. Logical relationships can also be defined to relate segments from one or more physical databases. Each physical DBD must also be assigned one of the four physical access methods. If using HIDAM access, the associated index database must also be defined.
2. If desired, **define one or more secondary index databases** for individual physical databases.
3. **Define one or more logical databases** using DBD statements that reference segments in already defined physical databases. Logical DBDs typically make explicit the logical relationships defined in the underlying physical DBDs. Concatenated segments can also be defined to specify run-time access regarding the physical parent, the logical parent, and the common (physical/logical) child.
4. **Define one or more program communication blocks** to define the application sensitivity for segments in a physical or logical database. Run-time access options for segments and the entire database can also be specified via the PROCOPT parameter.
5. **Define a program specification block** to collect all of the PCBs that can be used by an application.

DL/I Commands

The DL/I commands constitute the run-time database interface for an application. Collectively, the DL/I commands are a procedural language for data access, data retrieval, and data manipulation. They are implemented as a set of subroutine calls or preprocessed commands with various parameters. An application requests desired database operations by embedding the appropriate calls at specific points in the source code. Separate DL/I compilers are provided for a number of application programming languages.

The DL/I commands are both navigation- and access-path-oriented.

Basic Operations

The basic DL/I commands are:

- **GET UNIQUE (GU)** —— Retrieves a named segment occurrence (direct retrieval)
- **GET NEXT (GN)** —— Retrieves the next segment occurrence in the hierarchical access path
- **GET NEXT WITHIN PARENT (GNP)** —— Retrieves the next segment occurrence under the current parent occurrence
- **GET HOLD UNIQUE (GHU)** —— Same as GU, but permits a subsequent DELETE or REPLACE
- **GET HOLD NEXT (GHN)** —— Same as GN, but permits a subsequent DELETE or REPLACE
- **GET HOLD NEXT WITHIN PARENT (GHNP)** —— Same as GNP, but permits a subsequent DELETE or REPLACE
- **REPLACE (REPL)** —— Updates an existing segment occurrence in the database
- **INSERT (ISRT)** —— Inserts a new segment occurrence in the database
- **DELETE (DLET)** —— Deletes an existing segment occurrence and all of its dependents

Call Format

Syntax

Call-level DL/I has the following format:

```
CALL langDLI((#PARMS,)function,pcb-name,user-io-area,(ssa...))
```

Parameters

langDLI

Specifies the language of the calling program (for example, PLITDLI for a PL/I program).

#PARMS

Specifies the number of parameters for the call, not including the *#parms* parameter itself.

function

Specifies one of the DL/I command codes (GU, GN, REPL, etc.).

pcb-name

Specifies the PCB to be used with the call.

user-io-area

Identifies the name of the I/O area. See Program Communication (see page 55).

ssa

Specifies one or more optional segment search arguments (SSAs). There can be from 0 to 15 SSAs.

Segment Search Arguments

A **segment search argument** (SSA) specifies criteria for selecting a segment occurrence along the hierarchical access path. SSAs take the following form:

Syntax

segment-name(*field-name operator field-value*)

Parameters

segment-name

Identifies the name of the desired segment.

field-name

Identifies the name of a field in the segment.

operator

Specifies a standard relational operator (=, >=, <= , etc.).

field-value

Specifies an actual value for *field-name*.

Call-level DL/I Example

The example below selects the EMPLOYEE segment occurrence whose ID field has the value 123456:

```
GU, EMP-PCB, IO-A, EMPLOYEE (ID = 123456)
```

DL/I searches the EMPLOYEE segment occurrences within the database identified by EMP-PCB (the PCB name) and returns the contents of the found occurrence to IO-A (the user I/O area). If duplicate values are allowed for the search field, DL/I returns the first qualifying occurrence into the I/O area.

You can construct more complex selection criteria by combining SSAs with logical operators (AND, OR, etc.). By combining SSAs, you can direct the search to any level in the hierarchy.

Command-level DL/I (EXEC DLI) Example

DL/I database access is also possible using EXEC DLI commands. These commands allow similar functionality and content as call-level DL/I. The example below selects the EMPLOYEE segment occurrence whose ID field has the value 123456 (assuming EMP-ID contains ‘123456’), and is equivalent to the call-level example above:

```
EXEC DLI GU USING PCB(EMP-PCB) SEGMENT(EMPLOYEE) WHERE(ID=EMPID) INTO(IO-A);
```

You can construct more complex selection criteria by combining multiple SEGMENT statements. By combining SEGMENT statements, you can direct the search to any level in the hierarchy.

Program Communication

The Program Communication Block (PCB)

When an application performs operations against a database, it always does so through a program communication block (PCB). The PCB restricts the application's access to specific segments selected from the definition of an underlying database.

Regardless of whether the definition is for a physical or a logical database, the database always appears to the application as hierarchical. This is an important point because the flow of the application's processing must always conform to a specific hierarchical path. In other words, application access to a database always starts at a root segment occurrence and proceeds downward through the hierarchy, moving from left to right among segment occurrences on the same level.

PCB Provides for Transfer and Control of Information

At program run time, DL/I maintains an I/O area for each PCB defined in the PSB. The PCB area provides for the transfer of data and control information between the application and DL/I. The PCB I/O area contains a control block with a number of fields. DL/I updates the control fields after each DL/I call. An application's access to these fields is established by declaring the fields as program variables. It is the application's responsibility to check the control fields, as appropriate, after each DL/I call.

Basic DL/I Control Fields

The basic DL/I control fields (with sample names) are:

- **DBD-NAME** –– Name of the DBD referenced by the PCB. This DBD determines the access path available to the application.
- **SEG-LEVEL** –– The current segment level in the hierarchy.
- **STATUS-CODE** –– DL/I result status code.
- **PROCOPTS** –– Processing options in effect, as specified in the PCB definition.
- **SEG-NAME** –– Segment name for the segment occurrence last accessed.
- **KEY-LENGTH** –– Length of the concatenated key for the segment occurrence last accessed.
- **SEN-SEGS** –– Number of segments available to the application, as specified in SENSEG statements in the PCB.
- **KEY-AREA** –– Key feedback area for the concatenated key of the segment occurrence last accessed.

Database Positioning

The SEG-LEVEL, SEG-NAME, and KEY-AREA fields in the PCB help the application to keep track of its current position in the database. The application can use the current contents of these fields to direct subsequent database navigation and/or retrieval operations.

The CA IDMS/DB Environment

Set is the Basic Structure

In CA IDMS/DB, the basic structure is the **set**. A set consists of **record types** that are related as **owner** and **member**. Individual record types can participate in more than one relationship (set) either as owner or member (that is, a member record type can have more than one owner record type).

Multiple Ownership Support

Support for multiple ownership is the most basic difference between DL/I and CA IDMS/DB. In DL/I, a child segment (equivalent to a member record type) can have one and only one parent segment (equivalent to an owner record type).

Note: DL/I's support for bidirectional logical relationships provides the functional equivalent of multiple ownership.

Schema: The Top-Level Definition

In CA IDMS/DB, the top-level definition is known as the **schema**. The schema names all of the allowable record types and defines the **elements** (fields) that can appear in each record type. The schema also names and defines the possible relationships among the record types; these defined relationships are the sets.

Subschema: The Second-Level Definition

The second-level definition in CA IDMS/DB is known as the **subschema**. As its name indicates, the subschema defines a subset of the top-level schema definition. (While a subschema is usually a subset of a schema, it can also duplicate a schema in its entirety.) Any number of subschemas can be defined for a given schema.

Defining CA IDMS/DB Databases

Use Data Description Language (DDL)

The database administrator prepares the schema definition using source statements provided in the **schema data description language** (Schema DDL). The database administrator codes the subschema definitions using similar source statements provided in the **subschema data description language** (Subschema DDL).

You use CA IDMS physical data definition language statements to create a DMCL module that maps the schema areas to physical files and defines buffers for database operations.

Note: For more information about defining a DMCL, see the *CA IDMS Database Administration Guide.*

DDL Compilers Process Source Statements

Separate schema and subschema DDL compilers process the source statements. The CA IDMS Command Facility is used to produce assembler source for the DMCL module. The DMCL assembler source must then be assembled to produce object modules that map the logical areas into physical files and set up the necessary buffers.

Executing CA IDMS/DB Applications

At application run time, CA IDMS/DB loads the object-form subschemas and DMCL modules. The application is then ready to start issuing **data manipulation language** (DML) calls for database operations. The object-form subschemas serve as control tables for the application. These subschemas maintain status information so the application can check the results of database requests.

Basic CA IDMS/DB Components

CA IDMS/DB Components

The diagram below illustrates the basic components in the CA IDMS/DB environment.

Note: For more information about complete descriptions of all of the CA IDMS/DB components, see the *CA IDMS Database Design Guide* and *CA IDMS Database Administration Guide*.

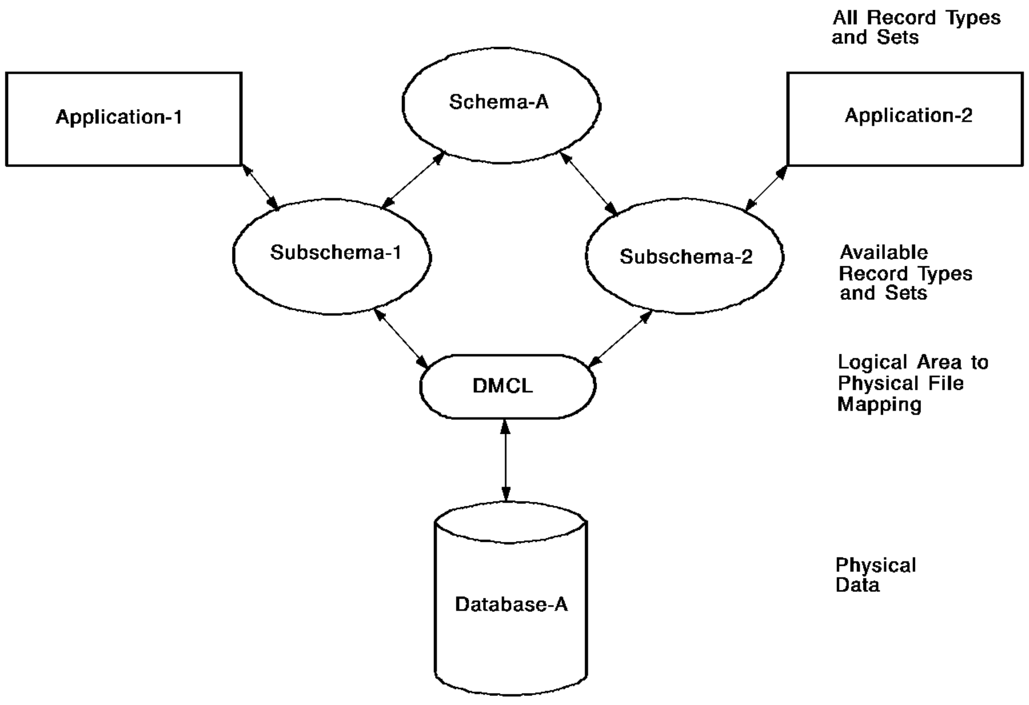

All Record Types and Sets
Application-1
Schema-A
Application-2
Subschema-1
Subschema-2
Available Record Types and Sets
DMCL
Logical Area to Physical File Mapping
Database-A
Physical Data

Figure 19. CA IDMS/DB components

DL/I and CA IDMS/DB Correspondences

CA IDMS DLI Transparency allows a DL/I application program to access a CA IDMS/DB database. To support this access, you must define a CA IDMS/DB schema and subschema that correspond to the specific DBD and PSB definitions expected by the application. For example, for each DL/I segment, hierarchy, and logical relationship, you must make sure that there is a corresponding CA IDMS/DB record type and set structure.

Note: For more information about the rules for defining schemas and subschemas, see the *CA IDMS Database Administration Guide*.

The following table summarizes the required correspondences between DL/I and CA IDMS/DB.

DL/I structure	CA IDMS/DB equivalence
Segment	Record
Parent segment	Owner record
Child segment	Member record
Parent/child relationship	Set relationship where parent is the owner and the child is the member
Child segment with a sequence field	Member record of a sorted set
Sequence field for a dependent segment	Sort key
Unsequenced child segments with insert rules as follows: ■ HERE ■ FIRST ■ LAST	Member record of a set with an order option as follows: ■ PRIOR ■ FIRST ■ LAST
Child segments with nonunique sequence fields with an insert rule as follows: ■ FIRST ■ LAST	Member record of a sorted set with the following duplicates option: ■ FIRST ■ LAST
Child segments with nonunique sequence fields with an insert rule of HERE*	Member record of a set with a set order option of PRIOR
Logical relationship ■ Physical parent segment ■ Logical parent segment ■ Logical child segment	Many to many relationship ■ Owner record ■ Owner record ■ Junction record

DL/I structure	CA IDMS/DB equivalence
Dependent segments in a physical access database	Members of a CA IDMS/DB set
Root segment in an ACCESS=HDAM database	CALC record
Root segment in an ACCESS=HISAM database	DIRECT record in a SYSTEM-owned indexed set; ascending or descending sort order on the record's symbolic key (equivalent to the sequence field of the HISAM root segment)
Root segment in an ACCESS=INDEX database (pointer segment)	Member record in a SYSTEM-owned indexed set; ascending sort order on the record's symbolic key (equivalent to the sequence field of the index pointer segment); also, VIA member in a set owned by the target record
Pointer segment (root segment in the corresponding ACCESS=INDEX database)	Member record in a SYSTEM-owned indexed set; ascending sort order on the record's symbolic key (equivalent to the sequence field of the index pointer segment); also, VIA member in a set owned by the target record
Target segment (root segment)	CALC record that owns the pointer record in the target-pointer set
Secondary index target segment	Owner record of a target-pointer set
Pointer segment (root segment of the corresponding ACCESS=INDEX database)	Record that is a member of an indexed set sorted in ascending order on the value of the sort key (i.e. the sequence field for the equivalent segment); record is also a VIA member of a set owned by the target record
ACCESS=HIDAM database: ■ Root segment (target segment)	Record equivalents for ACCESS-HIDAM database: ■ CALC record that owns a pointer record through the target-pointer set

Note: *Special considerations apply to insert rules (see Sequenced and Unsequenced Child Segments (see page 61)).

Segments and Record Types

For each segment defined in a physical DBD, there must be a corresponding record type in a CA IDMS/DB schema. Segments from logical DBDs are exceptions and must not appear in the schema. Logical segments are redefinitions of segments already defined in physical DBDs.

Note: The CA IDMS DLI Transparency syntax generator creates the necessary correspondences in the resulting schema, subschema, and DMCL source. You do not have to code the CA IDMS/DB definitions manually.

Sequenced and Unsequenced Child Segments

CA IDMS/DB requires different definitions for child segments, depending on whether they are sequenced or unsequenced.

Sequenced Child Segments

Sequenced child segments correspond to member record types in sorted sets. The child segment's sequence field is used for the sort key in the CA IDMS/DB sorted set.

- If the child segment is defined for U (unique) sequence field values, duplicates are not allowed for the set (DUPLICATES NOT ALLOWED).
- If the child segment is defined for M (multiple or duplicate) sequence field values, the value for the insert RULES parameter determines where duplicate fields are stored within the set sequence:
  - **FIRST** —— the set is ordered DUPLICATES FIRST.
  - **LAST** —— the set is ordered DUPLICATES LAST.
  - **HERE** —— the corresponding record type is a member in an unsorted set with a set order option of PRIOR.

Unsequenced Child Segments

Unsequenced child segments correspond to member record types in unsorted sets. The set ORDER option is determined by the value for the RULES parameter on the child's SEGM statement:

- **HERE** —— Corresponds to an order option of PRIOR.
- **FIRST** —— Corresponds to an order option of FIRST.
- **LAST** —— Corresponds to an order option of LAST. Note that LAST is the DL/I default for unsequenced segments. If a child segment does not have RULES specified, LAST is used for the order option in the corresponding unsorted set.

Deletable Segments

If a DL/I segment can be deleted, the corresponding record type must have prior pointers. As a rule, *all* sets should have prior pointers.

Hierarchies and Sets

Parent/Child Relationships Correspond to Sets

DL/I parent/child relationships (hierarchies) correspond to CA IDMS/DB sets. There must be a CA IDMS/DB set for each physical parent/child relationship. In a CA IDMS/DB set, the owner record type corresponds to the parent segment, and the member record type corresponds to the child segment.

With CA IDMS DLI Transparency, CA IDMS/DB sets can have only one member record type. Multi-member sets are not allowed.

DL/I Hierarchies and CA IDMS/DB Sets

The diagram below shows a sample DL/I hierarchy converted to a series of CA IDMS/DB sets.

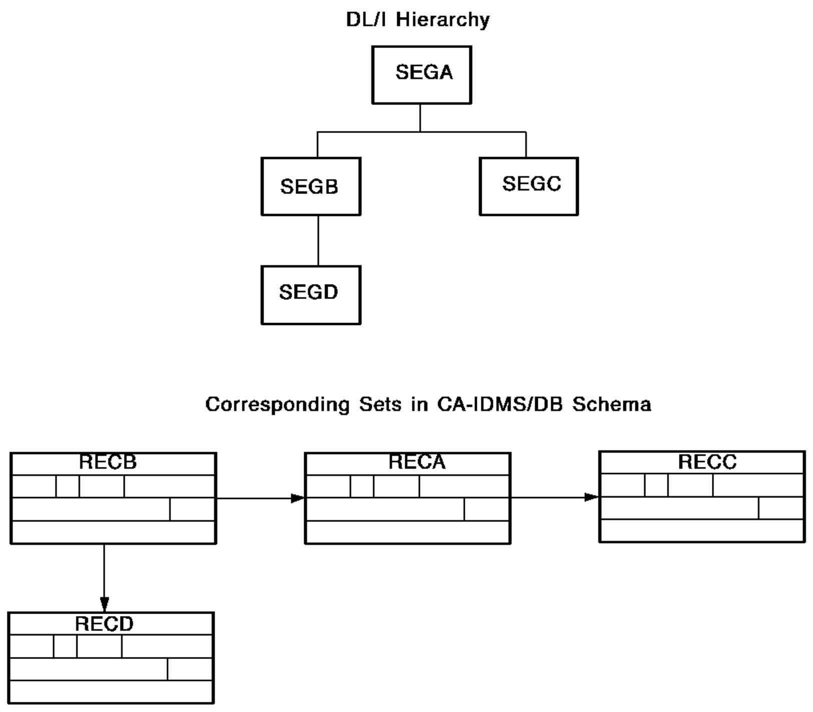


Figure 20. DL/I hierarchies and CA IDMS/DB sets

Logical Relationships and Sets

In CA IDMS DLI Transparency, each segment in a logical relationship corresponds to one of three CA IDMS/DB record types.

Junction Record

The **logical child segment** is defined as a junction record that is a member of two sets. The owner of one set corresponds to the **physical parent segment**; the owner of the other set corresponds to the **logical parent segment**.

Owner and Member Records

If the DL/I physical and logical parents are the same segment, one record type is used to represent both parents. In this case, the record type is the owner of the two sets of which the junction record (equivalent to the logical child) is the member.

The junction record must always have a location mode of VIA. The VIA set is the set of which the record type for the physical parent is the owner. Note that database load procedures can override this consideration.

CA IDMS DLI Transparency requires that all logical relationships (that is, unidirectional, bidirectional virtual, and bidirectional physical) be implemented as bidirectional virtual relationships. The conversion to bidirectional virtual is transparent to an application. However, the conversion of bidirectional physical relationships requires special consideration.

Implementing a Bidirectional Physical Relationship

To implement a bidirectional physical relationship as a bidirectional virtual relationship in CA IDMS/DB, a record type is defined for each of the parent segments. Additionally, a single record type is used to represent the physically paired child segments. This record type is defined as a VIA junction record in the set owned by each of the parent record types. In bidirectional virtual terms, the junction record type becomes the equivalent of the real logical child and the virtual logical child.

DL/I Logical Relationship and CA IDMS/DB Sets

The following diagram illustrates a DL/I bidirectional virtual relationship and the CA IDMS/DB set structures used to implement it.

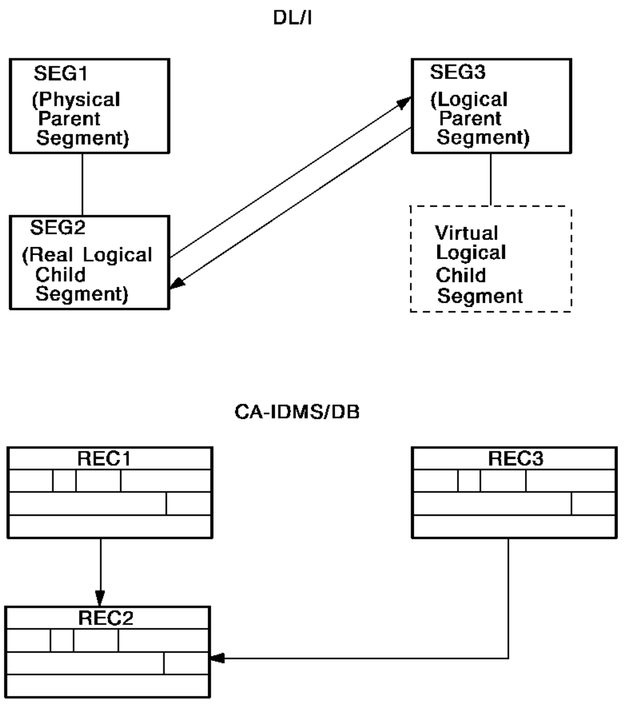

DL/I
SEG1 (Physical Parent Segment)
SEG3 (Logical Parent Segment)
SEG2 (Real Logical Child Segment)
Virtual Logical Child Segment
CA-IDMS/DB
REC1
REC3
REC2

Figure 21. DL/I logical relationship and corresponding CA IDMS/DB sets

DL/I Access Methods in CA IDMS/DB

Each of the four access methods allowed for physical DBDs requires a different implementation in CA IDMS/DB. The HSAM, HISAM, HDAM, and HIDAM access methods are discussed below.

HSAM

CA IDMS DLI Transparency does not implement HSAM databases directly. However, the indirect implementation is transparent to any DL/I application using an HSAM database. The CA IDMS DLI Transparency implementation depends on whether the HSAM database is sequenced or unsequenced:

- A **sequenced HSAM database** has a root segment that is sorted on the basis of its sequence field. A sequenced HSAM database is defined in the same way as a HISAM database (see "HISAM" below). In the schema, the root segment of the HSAM database is treated as the root segment of a HISAM database. Once the HSAM database is defined as a HISAM database, the appropriate structures are defined in the corresponding CA IDMS/DB schema.
- An **unsequenced HSAM database** is defined as a separate area in the CA IDMS/DB schema. This area contains only the record types and sets needed to reflect the HSAM segments and their hierarchies. All record types are defined with a location mode of DIRECT.

HISAM

CA IDMS DLI Transparency relates the root segment in the HISAM database to the member record type in a system-owned indexed set. The member record has a location mode of DIRECT; its symbolic key corresponds to the root segment's sequence field. If it is necessary to keep the member record (the root segment equivalent) in physical sequential order, ascending or descending order is defined for its symbolic key.

Note: For more information about indexed sets, see the *CA IDMS Database Administration Guide*.

Sample HISAM Database and CA IDMS/DB Sets

The diagram below, shows a sample HISAM database and the CA IDMS/DB sets used to implement it.

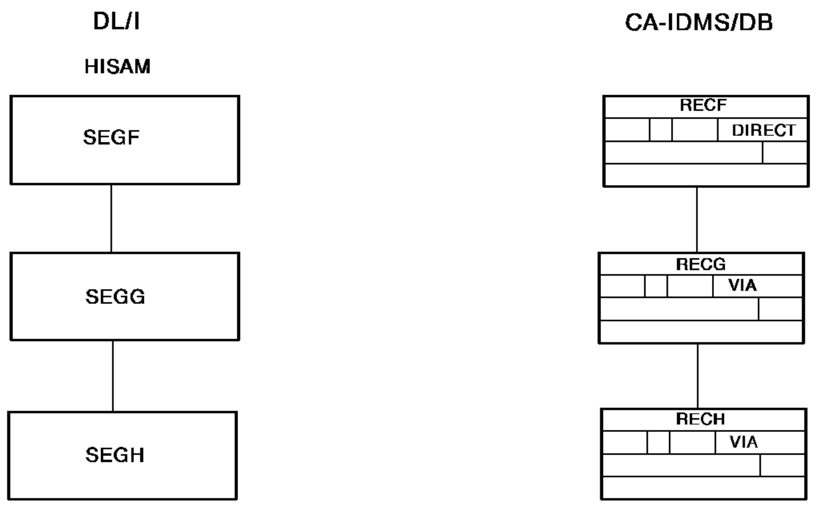


Figure 22. Sample HISAM database and corresponding CA IDMS/DB sets

HDAM

In CA IDMS DLI Transparency, the root segment in an HDAM database corresponds to an owner record type with a location mode of CALC. The root segment's sequence field is defined as the CALC key.

Sample HDAM Hierarchy and CA IDMS/DB Sets

The diagram below shows a sample HDAM hierarchy and the corresponding CA IDMS/DB set structures.

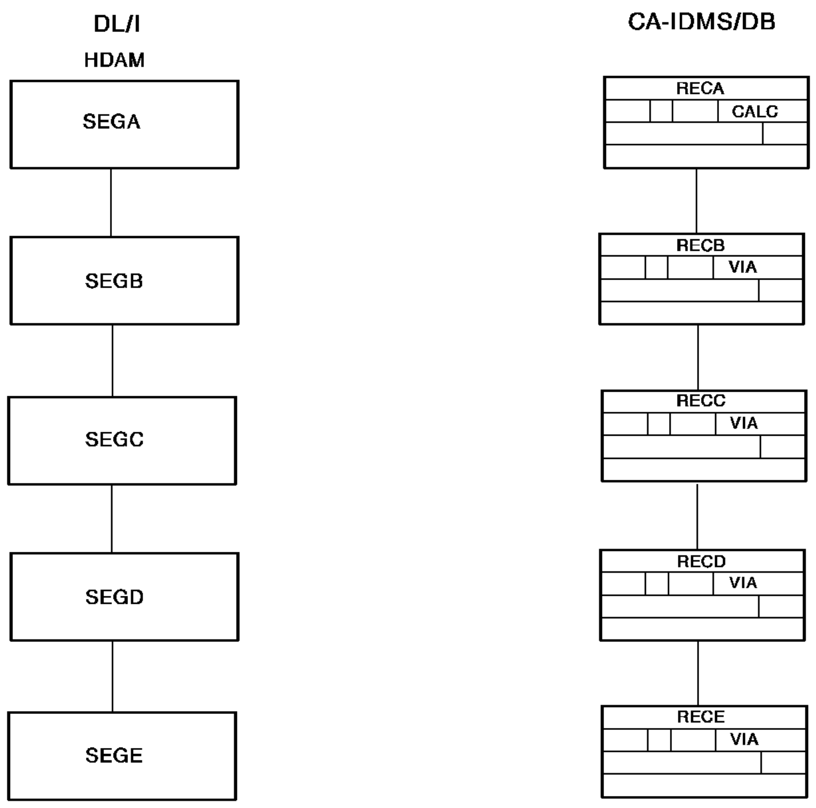


Figure 23. Sample HDAM hierarchy and corresponding CA IDMS/DB sets

HIDAM

As with an HDAM database, the HIDAM root segment is defined as an owner record with a location mode of CALC. The root segment's sequence field becomes the CALC key.

In a HIDAM database, the root segment is also the source and target segment for the associated index database. To account for the index pointer segment, a member record type is defined with a location mode of VIA within an indexed set owned by the CALC owner record type. The index record contains a single element to match the root segment's sequence field (CALC key in the owner record type). The index record also contains any data fields defined in the index. During processing, CA IDMS/DB maintains matching occurrences between the index (member) record and the owner of the set.

Sample HIDAM Hierarchy and CA IDMS/DB Sets

The diagram below shows a sample HIDAM hierarchy and the corresponding CA IDMS/DB set structures.

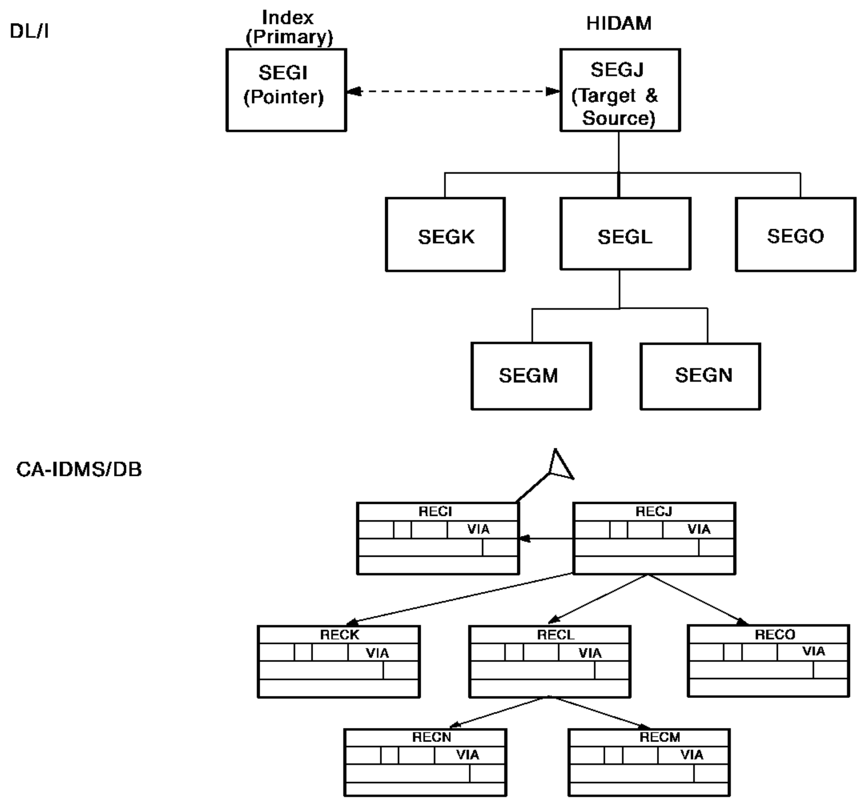


Figure 24. Sample HIDAM hierarchy and corresponding CA IDMS/DB sets

DL/I Secondary Indexes in CA IDMS/DB

A DL/I secondary index involves a primary database and an index database. The primary database contains a source and a target segment. The index database contains an index pointer segment, which is also the root segment.

Define Index Pointer Segment as Member Record

In CA IDMS DLI Transparency, the index pointer segment is defined as a member record type with a location mode of VIA in a set owned by the target record. The pointer segment is also defined as a member record in a system-owned indexed set. This set is sorted in ascending order on the pointer record's symbolic key, which is equivalent to the sequence field in the pointer segment.

CA IDMS/DB does not require a separate set to reflect the source segment and pointer segment relationship.

Implementing Pointer and Target Relationships in CA IDMS/DB

The illustration below illustrates the CA IDMS/DB set structure that relates the pointer and target segments. Note that this relationship is the same for a secondary index and a HIDAM index database.

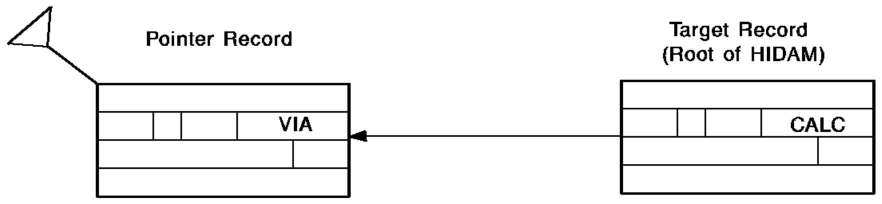


Figure 25. CA IDMS/DB implementation of pointer and target relationship

DL/I Secondary Index and CA IDMS/DB Sets

The following diagram shows a secondary index for an HDAM primary database and the corresponding CA IDMS/DB set structures. Note that the primary database is an HDAM database and the pointer segment is in the index (secondary) database.

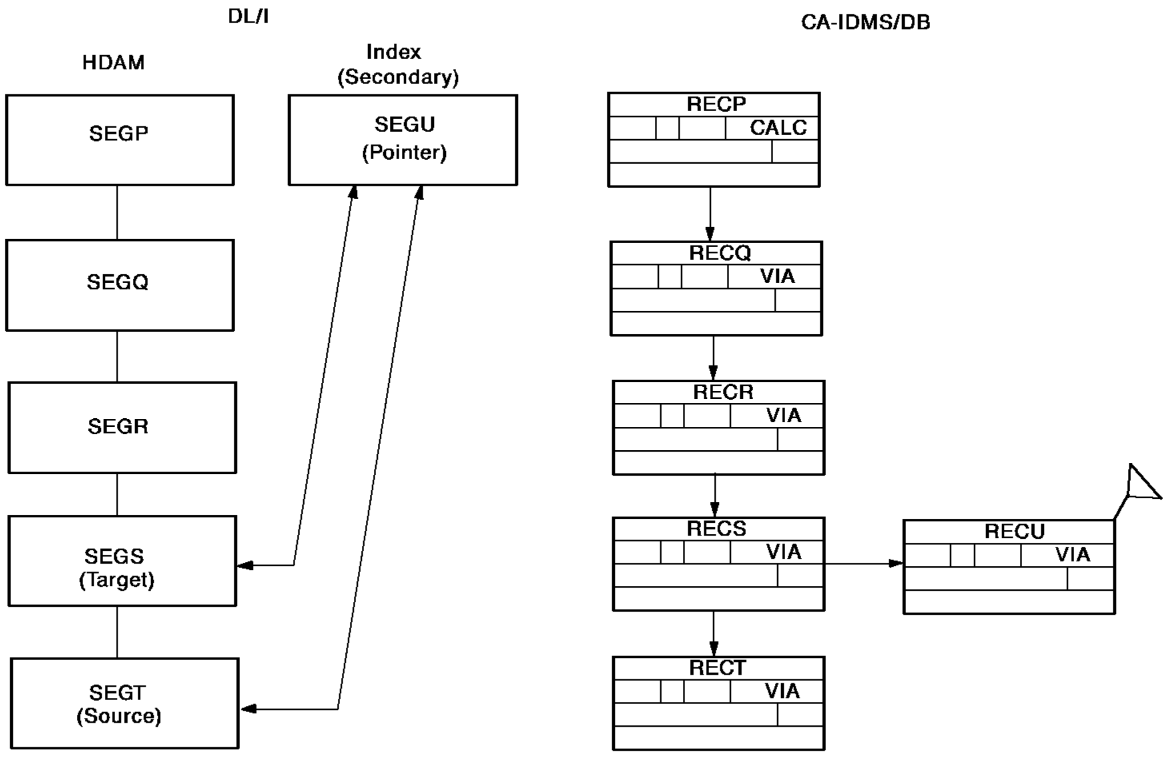


Figure 26. DL/I secondary index and corresponding CA IDMS/DB sets

Parallel Processing Support in CA IDMS/DB

CA IDMS DLI Transparency supports DL/I parallel processing in two ways:

- **Multiple PCBs** — A CA IDMS/DB subschema can include definitions to reflect any number of PCBs in a corresponding PSB, with no limitation on the DL/I structures contained in the PCBs. For example, when two PCBs that define the same hierarchy are both used by a DL/I application, CA IDMS DLI Transparency will maintain database positioning (currency) independently for each PCB.
- **Multiple positioning** — The DL/I PCB statement allows you to optionally specify separate positioning for each hierarchical path in a database definition. CA IDMS DLI Transparency will maintain separate currency for each CA IDMS/DB structure that corresponds to one of the DL/I hierarchies.

DL/I Calls in CA IDMS/DB

DL/I Database Calls

CA IDMS DLI Transparency supports all of the DL/I database calls and all of the DL/I command codes shown in the tables below.

Call Function	Meaning
GU	GET UNIQUE
GN	GET NEXT
GNP	GET NEXT WITHIN PARENT
GHU	GET HOLD UNIQUE
GHN	GET HOLD NEXT
GHNP	GET HOLD NEXT WITHIN PARENT
ISRT	INSERT
DLET	DELETE
REPL	REPLACE

DL/I Command Codes

Code	Purpose
C	Allows use of concatenated keys in SSAs
D	Specifies path calls (that is, allows retrieval, modification, or insertion of several segments with one call)
F	Permits search for a segment to start at the first occurrence under its parent, regardless of positions.
L	Causes the last occurrence of a segment type to be used in satisfying a call
N	Prevents the replacement of the specified segment(s) following a path retrieval call
P	Establishes parentage at the specified level when used with a retrieval call
U	Maintains current positioning at the specified level
V	Maintains current positioning at all levels higher than the specified level
- (null command code)	Causes no special processing to occur

Extensions to Basic Calls

As extensions to the basic calls shown in the DL/I command codes table above, CA IDMS DLI Transparency also supports:

- **Path calls** —— Calls used to retrieve, modify, or insert multiple segments in a hierarchical path.
- **Qualified and unqualified calls** —— Calls specified with or without segment search arguments (SSAs).
- **Qualified and unqualified SSAs** —— SSAs with qualification statements or qualified by segment type only.

DL/I System Service Calls

The following DL/I system service calls are also supported under CA IDMS DLI Transparency:

- **PCB** —— Schedules a PSB call (used only with CICS)
- **TERM** —— Terminates a PSB call (used only with CICS)
- **ROLL and ROLB** —— Treated as a DML ROLLBACK request
- **CHKP (CHECKPOINT)** —— Treated as a DML COMMIT request

Usage Considerations

When defining the CA IDMS/DB equivalents for your DL/I structures, keep in mind the following usage considerations:

- CA IDMS DLI Transparency does not support multiple noncontiguous sequence fields for a virtual logical segment. A single sequence field, however, is supported.
- CA IDMS DLI Transparency always uses the following delete rules: **physical**, **virtual**, **logical**, for the physical parent, logical child, and logical parent, respectively. Refer to the appropriate DL/I documentation for a description of the delete rules.
- CA IDMS DLI Transparency supports sparse indexing through null-value specifications or index suppression exits. If you use index suppression exits, you must convert the exits to CA IDMS/DB database procedures.

  For more information about a detailed description of index suppression exits, see Index Suppression Exit Support (see page 253).
- You must convert segment edit/compression exits to CA IDMS/DB database procedures.

- CA IDMS DLI Transparency supports PROCOPT E on the PCB statement. To reflect this processing option, you must specify EXCLUSIVE for the CA IDMS/DB area ready option.
- CA IDMS DLI Transparency automatically supports PROCOPT O and requires no additional specification for it.

Unsupported DL/I Features

CA IDMS DLI Transparency does not support the following DL/I features:

Feature	Comment
GSAM databases, which are sequential files.	You must modify DL/I application programs that issue calls to GSAM databases by removing the GSAM calls. Alternatively, you can replace the GSAM calls with standard sequential file processing requests.
The PCB PROCOPTs of L and LS.	You can obtain the same results by changing the L or LS to an I. This substitution is invalid when the application program is used in conjunction with the DL/I calls load utilities.
The PCB PROCOPT of GS.	You can obtain the same results by changing the GS to a G.
Field-level sensitivity in PCBs.	You can reflect field-level sensitivity by excluding the corresponding elements from their record type definitions in the subschema.
DL/I utilities.	CA IDMS/DB provides a complete set of utilities that perform all the necessary functions.
DL/I logging.	Remove calls for logging in the DL/I application / programs. CA IDMS/DB journaling is used in place of DL/I logging.
The CHECKPOINT/RESTART function.	However, CA IDMS DLI Transparency does support the checkpoint call when used alone. CA IDMS DLI Transparency honors the checkpoint call by issuing a CA IDMS/DB COMMIT. Therefore, remove all restart calls from the DL/I application program, and consider removing the checkpoint part of the call as well.

Feature	Comment
The DL/I calls: PURG GSCD CHGN XRS CMD DEQ GCMD LOG SNAP STAT	
The Q command code.	CA IDMS DLI Transparency bypasses this command code and returns a blank status. If a DL/I program contains Q codes, you don't have to remove them.
Use of the L command code to override a DL/I ISRT call.	CA IDMS DLI Transparency does support the L command code when used with a DL/I GET call.
Use of MPS Batch	EXEC DLI usage does not support MPS Batch
The EXEC DLI LOAD function	CA IDMS DLI Transparency supports call-level DL/I load programs using ISRT calls, and provides an independent load utility. An 'AD' status will be returned for this call.

Chapter 3: CA IDMS DLI Transparency Syntax Generator

This section contains the following topics:

About This Chapter

The CA IDMS DLI Transparency syntax generator translates DL/I database definitions into syntax statements for a CA IDMS DLI Transparency interface program specification block (IPSB) and corresponding CA IDMS/DB schema, DMCL, and subschema definitions. This chapter describes how to use the CA IDMS DLI Transparency syntax generator.

The CA IDMS DLI Transparency Syntax Generator

Syntax Generator Input

Input to the syntax generator consists of the following control blocks created by the CA-supplied macros:

- **Database definition (DBD) control blocks**—Define the segment types, the physical hierarchical structure, and other characteristics of each database for which a view is defined in the PSB. The DBD control blocks are used to produce the CA IDMS/DB schema, DMCL, and subschema source statements.
- **Program specification block (PSB)**—Defines the views of all physical and/or logical databases available to a DL/I application that uses the PSB. The PSB control block is used to produce the IPSB source statements.

Syntax Generator Output

The syntax generator produces source statements for a CA IDMS/DB schema, DMCL, and subschema and a CA IDMS DLI Transparency IPSB.

Schema, DMCL, and Subschema Source

The schema source statements produced by the syntax generator define CA IDMS/DB areas, record types, and set types corresponding to the databases, segments, and parent/child (hierarchical) relationships defined in the DL/I DBD control blocks.

The DMCL source statements define how the CA IDMS/DB areas are to be mapped to the physical database files. They are derived from information in the DL/I DBD control blocks.

The subschema source statements define the CA IDMS/DB logical views that correspond to the views defined in the DL/I DBD control blocks.

You can input the generated source definitions to the appropriate CA IDMS/DB compilers to create load modules for use with the IPSB compiler, the CA IDMS DLI Transparency load utility, and the CA IDMS DLI Transparency run-time interface.

IPSB Source

The IPSB source statements define the correspondences between the DL/I database referenced by the DL/I application and the CA IDMS/DB database accessed by the CA IDMS DLI Transparency run-time interface.

The resulting IPSB source statements are organized as follows:

- **IPSB SECTION**—Relates the IPSB being defined to the corresponding DL/I program specification block (PSB)
- **AREA SECTION**—Identifies the CA IDMS/DB database areas that are to be readied by the CA IDMS DLI Transparency run-time interface in any usage mode other than shared retrieval (the default)
- **RECORD SECTION**—Names the CA IDMS/DB record types needed to service DL/I calls and defines the DL/I fields to be referenced by the DL/I calls
- **INDEX SECTION**—Provides the information necessary to relate CA IDMS/DB records and sets to DL/I secondary index structures and HIDAM index structures that are used and/or maintained by the CA IDMS DLI Transparency run-time interface
- **PCB SECTION**—Replaces the program communication blocks (PCBs) defined for the PSB

After reviewing the IPSB source statements, you can input them to the IPSB compiler to create an IPSB load module for use with the CA IDMS DLI Transparency run-time interface.

Special IPSB Load Module

To execute the CA IDMS DLI Transparency load utility, you need a special IPSB load module. You produce a load IPSB by specifying the LOAD option in the GENERATE IPSB statement.

For specific considerations that apply only to the load IPSB, see CA IDMS DLI Transparency Load Utility (see page 171).

Syntax Generator Operation

Operation of the CA IDMS DLI Transparency syntax generator involves the following steps:

1. Select, assemble, and link edit all of the DBDs, including logical DBDs, associated with the PSB you want to use. Select, assemble, and link edit the PSB. The PSB represents an application's view of the DL/I database(s) defined in the DBDs.

   Note: The DBDs and PSB must be assembled using the CA-supplied macros.

2. Code the appropriate syntax generator statements.
3. Execute the syntax generator.

Preparing Syntax Generator Input

The syntax generator analyzes the DBD control blocks to produce schema, DMCL, and subschema source statements. It analyzes one PSB control block to produce a set of IPSB source statements.

You must assemble the DBDs and the PSB using the macros supplied with CA IDMS DLI Transparency. You must then link edit the resulting assemblies to populate a new load library that contains a load module for each DBD and a load module for the PSB. The load library must be available to the syntax generator when you run it. Be sure to keep your DL/I and CA IDMS DLI Transparency load libraries separate.

When you execute the syntax generator, it will attempt to load the PSB and all referenced DBDs. Since it can be difficult to keep track of all the DBD dependencies, you may find that the easiest course is simply to assemble and link edit all of your DBDs.

DBD Control Blocks

Each database definition (DBD) control block defines the segment types, hierarchical structure, and other characteristics of a database referenced in the PSB.

Note: Any given PSB can reference many DBDs, thus providing access to many databases.

You must create a CA IDMS DLI Transparency DBD control block for each physical or logical DBD associated with the PSB. You must also create a DBD control block for each physical DBD that is referenced in a logical DBD.

Creating the DBD Control Block

To create the DBD control blocks, perform the following steps for each DBD:

1. Select the DL/I source code for the DBD.
2. Assemble and link edit the source code for the DBD. You must use the CA IDMS DLI Transparency-supplied macros when assembling the DBD source.

Assembly and Link Edit of a DBD

To assemble and link edit a DBD, use the DBD JCL shown in CA IDMS DLI Transparency JCL (see page 257).

Note: A resulting load module has the same name as the DL/I DBD, but it can be used only with CA IDMS DLI Transparency. Do not attempt to use a DBD load module in your native DL/I environment.

PSB Control Block

Creating a PSB Control Block

To create a PSB control block for use with the syntax generator, perform the following steps:

1. Select the DL/I source code for the PSB you want to use.
2. Assemble and link edit the source code for the PSB. You must use the CA IDMS DLI Transparency-supplied macros when assembling the PSB source.

Assembly and Link Edit of a PSB

To assemble and link edit the PSB, use the PSB JCL shown in CA IDMS DLI Transparency JCL (see page 257).

Note: The resulting load module has the same name as the DL/I PSB, but it can be used only with CA IDMS DLI Transparency. Do not attempt to use the PSB load module in your native DL/I environment.

Coding Syntax Generator Statements

The syntax generator statements fall into three groups:

- **Control statements** -- Specify input formatting and checking controls and output formatting for the syntax generator's report listing
- **GENERATE statement** -- Names the input DBD and PSB control blocks; also specifies the names for the output CA IDMS/DB schema, DMCL, and subschema source and the output IPSB source
- **Modification statements** -- Specify names for the output areas, records, and sets; also redefine the output areas, including the area usage modes

Control Statements

The control statements allow you to specify:

- The amount of storage to be used by the syntax generator
- The range of input columns for syntax generator statements
- Sequence checking for input statements
- Formatting for the syntax generator report listing

Syntax

```
►►─┬───────────────────────────────────────────────────┬──────────────►
   └─ CORe size = ─┬─ (48) ◄────────────── k ─┘
                   └─ (nnnnnn) ─┘

►─┬───────────────────────────────────────────────────────────────┬──►
  └─ ICTL = ──┬─ (1,80) ◄───────────────────────────────────────────┤
              └─ (start-column-number,end-column-number) ──────────┘

►─┬──────────────────────────────────┬───────────────────────────────►
  └─ OCTL = ──┬─ (60) ◄────────────────┤
              └─ (line-count) ─────────┘

►─┬───────────────────────────────────────────────────────────┬──────►
  └─ ISEQ = ─── (start-column-number,end-column-number) ──────┘

►─┬───────────────────────────────────┬──────────────────────────────►
  │ ┌───────────────────────────┐    │
  └─▼-- SPACE space-count ──────┴────┘

►─┬──────────────────────────┬───────────────────────────────────────►
  │ ┌─────────────────┐      │
  └─▼-- EJECT ────────┴──────┘

►─┬─────────────────┬───────────────────────────────────────────────►◄
  └─ *comments* ────┘
```

Parameters

CORe size=*(nnnnnn) k*

Specifies the amount of storage that the syntax generator will acquire to process the PSB and DBD control blocks. Storage acquisition is performed by a GETMAIN under OS or a GETVIS or COMREG under z/VSE.

Nnnnnn is a 1- to 6-digit numeric value. If the K option is included, it specifies an *nnnnnn* multiple of 1,024 (1K) bytes. If K is omitted, *nnnnnn* specifies the number of storage bytes desired (which the syntax generator rounds up to the next doubleword).

The CORE SIZE default is 48K bytes.

ICTL=*(start-column-number,end-column-number)*

Specifies a range of columns for coding input generator statements. The default and valid range of input columns is 1 through 80. If specified, ICTL must precede all noncontrol statements.

OCTL=*(line-count)*

Specifies the page length (number of lines) for the printed syntax generator report.

The OCTL default is 60 lines per page. Valid values are from 1 to 66. If specified, OCTL must precede all noncontrol statements.

ISEQ=*(start-column-number,end-column-number)*

Specifies sequence checking for input syntax generator statements. The start column and end column values identify the column range in which sequence numbers will appear. Valid values for the column start and end are 1 and 80, respectively. The column range cannot be more than 10 column positions wide.

The default is no sequence checking. If specified, ISEQ must precede all noncontrol statements.

SPACE *space-count*

Specifies line spacing for the printed syntax generator report. Valid values are 1 through 9. Note that only one blank is allowed between SPACE and *space-count*. You can specify any number of SPACE statements and include them anywhere in the syntax generator input statements.

EJECT

Specifies a page break for the printed syntax generator report. You can specify any number of EJECT statements and include them anywhere in the syntax generator input statements. The EJECT statement must appear on its own line.

comments

Designates comment text. You can embed comment text anywhere in the syntax generator input statements. Comment text is automatically terminated at the end of a line. To include comment text within a line, begin and end the text with asterisks (be sure to keep track of the number of asterisks). An odd number turns on comment text; an even number turns comment text off.

Example

```
ICTL=(1,72)
OCTL=(45)
ISEQ=3,72
EJECT
SPACE 2
*Begin comments with an asterisk
```

Figure 27. Sample control statements

GENERATE Statement

The GENERATE statement identifies the DL/I DBD and PSB control blocks to be input to the syntax generator.

The syntax generator uses the DBD control blocks to produce the CA IDMS/DB schema, DMCL, and subschema source definitions. It uses the PSB control block to produce the source statements for the IPSB compiler.

Deriving Record, Set, and Area Names

The syntax generator derives the record, set, and area names for the output source statements from the DL/I control blocks, as follows:

- **Record names** -- Derived from DL/I segment names.
- **Set names** -- Derived from the parent segment name and the child segment name in each DL/I hierarchy. The syntax generator concatenates the names with the literal "-".

  The resulting set names have a maximum length of 16 characters. If both names are 8 characters long, the syntax generator truncates the last character in the child name. Note that the truncation may cause duplicate set names.
- **Area name** -- Derived from the DL/I DBD name. The syntax generator appends the literal "-REGION" to the resulting area name.

You can override the generated names and specify different names using the modification statements (described later in this section).

Four Forms of the GENERATE Statement

The syntax generator provides four forms of the GENERATE statement:

- GENERATE SCHEMA
- GENERATE DMCL
- GENERATE SUBSCHEMA
- GENERATE IPSB

Specify the GENERATE statement appropriate for the type of output you want.

Process One GENERATE Statement at a Time

Include only one GENERATE statement for each execution of the syntax generator. The syntax generator places its output in a single SYSPCH file. The syntax generator can process multiple GENERATE statements, but all the output files would go to the same file, and you would have to separate the output yourself.

The GENERATE statement must be coded immediately after the control statements and before any modification statements.

GENERATE SCHEMA Statement

Syntax

```
►►─┬─ GENerate ─┬──────┬──────── SCHema name is schema-name ─┬──────►
   │            └ LOAD ┘                                     │
   │            ┌─── , ───┐                                  │
   └─ FOR dbd -▼- dbd-name. ─┴───────────────────────────────┘

 ►── DICTionary name is dictionary-name. ─────────────────────────►◄
```

Parameters

GENerate SCHema name is *schema-name*

Specifies that you want the syntax generator to produce a schema source definition.

Schema-name is the 1- to 8-character name of the output source definition. This is the name that you will supply as input to the CA IDMS/DB schema compiler.

LOAD

Produces a schema definition suitable for use with the CA IDMS DLI Transparency load utility. Specifically, it creates a schema in which the sets are defined as OPTIONAL MANUAL. Alternatively, you can use an already generated schema definition and change its sets to OPTIONAL MANUAL. If you do this, be sure to change the set definitions back to their original state after the load.

FOR DBD *dbd-name*

Specifies the DBD control block(s) from which to derive the schema source. You can specify multiple DBDs, separated by commas, to match the DBDs referenced in the associated PSB. Each *dbd-name* must be a 1- to 8-character name.

Be sure to specify all the DBDs associated with the PSB you will be using; this includes all physical, index, and logical DBDs.

DICTionary name is *dictionary-name*

Optionally identifies a dictionary name to be used in the SIGNON statement in the generated schema syntax.

If you omit the DICTIONARY NAME statement, the syntax generator will omit the DICTIONARY NAME IS clause in the SIGNON statement. As a result, the generated schema source will be placed in the default dictionary.

Example

```
GENERATE SCHEMA NAME IS SCHEMA1 FOR DBD PHYSDB1, PHYSDB2, INDXDBD.
DICTIONARY NAME IS PRODDIC.
```

Figure 28. Sample Schema GENERATE and NAME statements

GENERATE DMCL Statement

Syntax

```
►►─┬─── GENerate DMCL name is dmcl-name ──┬────────────────────────►
   │                                    │
   │            ┌──────── , ────────┐    │
   └─── FOR dbd ─▼── dbd-name. ───────┴────┘

 ►─┬──────────────────────────────────┬──────────────────────────►◄
   └─ SEGMENT name is segment-name . ─┘
```

Parameters

GENerate DMCL name is *dmcl-name*

Specifies that you want the syntax generator to produce a DMCL source definition. *Dmcl-name* is the 1- to 8-character name of the output source definition. This is the name that you will supply as input to the CA IDMS Command Facility.

FOR DBD *dbd-name*

Specifies the DBD control block(s) from which to derive the DMCL source. You can specify multiple DBDs, separated by commas, to match the DBDs referenced in the associated PSB. Each *dbd-name* must be a 1- to 8-character name.

Be sure to specify all the DBDs associated with the PSB you will be using; this includes all physical, index, and logical DBDs.

SEGment name is *segment-name*

Optionally supplies the name of a designated segment.

If you omit the SEGMENT NAME statement, the syntax generator will supply a default name for the segment. You will then have to edit the output source definition to reflect your CA IDMS/DB naming conventions.

Example

```
GENERATE DMCL NAME IS DMCL1 FOR DBD PHYSDB1, PHYSDB2, INDXDBD.
SEGMENT NAME IS ESCAPE.
```

Figure 29. Sample DMCL GENERATE and NAME statements

GENERATE SUBSCHEMA Statement

Syntax

```
►►─┬─── GENerate SUBschema name is subschema name ─┬──────────────────►
   │           ┌────── , ─────┐                  │
   └─── FOR dbd ─▼─ dbd-name. ──┴──────────────────┘

 ►─┬────────────────────────────────────────────────────────┬──────►◄
   ├─ SCHema ─────┬─ name is ─┬─ schema-name ─────┬─ . ─────┘
   └─ DICTionary ─┘           └─ dictionary-name ─┘
```

Parameters

GENerate SUBschema name is *subschema-name*

Specifies that you want the syntax generator to produce a subschema source definition.

Subschema-name is the 1- to 8-character name of the output source definition. This is the name that you will supply as input to the CA IDMS/DB subschema compiler.

FOR DBD *dbd-name*

Specifies the DBD control block(s) from which to derive the subschema source. You can specify multiple DBDs, separated by commas, to match the DBDs referenced in the associated PSB. Each *dbd-name* must be a 1- to 8-character name.

Be sure to specify all the DBDs associated with the PSB you will be using; this includes all physical, index, and logical DBDs.

SCHema name is *schema-name*

Optionally supplies the name of the associated schema. If you omit the *schema-name*, the syntax generator will supply a default schema name. You can only include one SCHEMA NAME statement.

DICTionary name is *dictionary-name*

Optionally identifies the name of the dictionary to be used in the SIGNON statement in the generated subschema syntax.

If you omit the DICTIONARY NAME statement, the syntax generator will omit the DICTIONARY NAME IS clause in the SIGNON statement. As a result, the generated subschema source will be placed in the default dictionary.

You can only include one DICTIONARY NAME statement.

Example

```
GENERATE SUBSCHEMA NAME IS SUBSCHA FOR DBD PHYSDB1, PHYSDB2, INDXDBD.
SCHEMA NAME IS SCHEMA1.
DICTIONARY NAME IS PRODDIC.
```

Figure 30. Sample Subschema GENERATE and NAME statements

GENERATE IPSB Statement

Syntax

```
►►─┬───────────────────────────────────────────────┬────────────────────────►
   └─ GENerate ─┬──────────┬── IPSB for PSB psb-name ─┘
                └─ LOAD ─┘

 ►─┬──────────────────────────────────────┬────────────────────────────────►◄
   └─ using SUBschema subschema-name ── . ──┘
```

Parameters

GENerate IPSB FOR PSB *psb-name*

Specifies the PSB you want to use. *Psb-name* must specify the 1- to 8-character name of the PSB control block.

LOAD

Optionally creates a special IPSB for use with the load utility. Specifically, the resulting sets will be defined as OPTIONAL MANUAL for loading purposes. LOAD also automatically creates the load processing option required by the load utility.

using SUBschema *subschema-name.*

Specifies the 1- to 8-character name of the subschema that will be used by the CA IDMS DLI Transparency run-time interface in conjunction with the IPSB load module.

Example

```
GENERATE IPSB FOR PSB PSB1 USING SUBSCHEMA SUBSCH1.
```

Figure 31. Sample GENERATE IPSB statement

Modification Statements

The CA IDMS DLI Transparency syntax generator modification statements allow you to override area, record, and set definitions in generated schema, DMCL, subschema, and IPSB source. The modification statements can be used in conjunction with any of the four GENERATE statements.

Note: Make sure that the schema, subschema, and IPSB source definitions remain consistent. That is, any modifications made to a subschema must also be made to the associated schema. Any modifications made to an IPSB must also be made to its associated schema and subschema. For example, if you add an area to the generated IPSB source, you must also add the same area to both the associated schema and subschema source.

Different Types

The modification statements are as follows:

- **ADD AREA statement** -- Generates source statements for defining a CA IDMS/DB database area
- **MODIFY AREA statement** -- Overrides a generated area name or changes the usage mode for a generated area
- **MODIFY RECORD statement** -- Overrides a generated record name
- **MODIFY SET statement** -- Overrides a generated set name

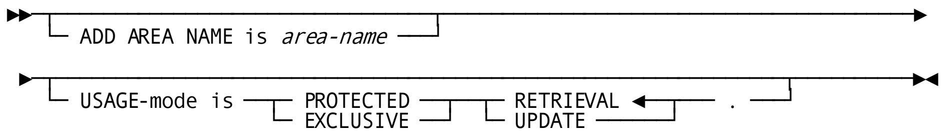
Each statement is described separately below.

ADD AREA Statement

The ADD AREA statement generates the source statements needed to define a CA IDMS/DB database area.

If you want to maintain index records in a separate area, you must include one ADD AREA statement for each index area. Specify the ADD AREA statement with the GENERATE statement for the IPSB and with the GENERATE statements for the associated schema and subschema.

Syntax



Parameters

ADD AREA NAME IS *area-name*

Specifies the CA IDMS/DB database area to be added.

Area-name must be a 1- to 16-character name.

USAGE-mode is

Specifies the usage mode in which an application can ready the area. The usage mode specifies the run-time operations that an application can perform against the CA IDMS/DB database area.

If neither PROTECTED nor EXCLUSIVE is specified, SHARED is the default. SHARED specifies that other concurrently executing applications can access the named area.

PROTECTED

PROTECTED prohibits update of the area by another concurrently executing application.

EXCLUSIVE

EXCLUSIVE prohibits access to the area by another concurrently executing application.

RETRIEVAL

Permits only retrieval (read-only) access for the database area

UPDATE

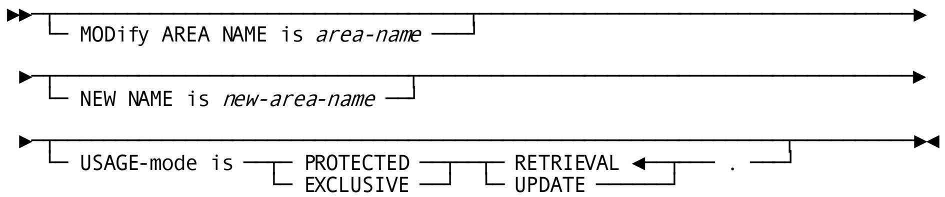
Allows all DML functions (STORE, ERASE, MODIFY, etc.) for the database area

MODIFY AREA Statement

The MODIFY AREA statement allows you to specify a name for a generated area. The specified name overrides the name supplied by the syntax generator. Note that the default area name consists of the DL/I DBD name concatenated with the literal "-REGION".

If the name of an area in the associated schema is different from the syntax generator-supplied name, you must include the MODIFY AREA statement to supply the correct schema-specific area name.

Syntax



Parameters

MODify AREA NAME is *area-name*

Identifies the generated area for which you want to specify a new name. *Area-name* must be a 1- to 16-character name.

NEW NAME is *new-area-name*

Specifies the new CA IDMS/DB database area name. *New-area-name* must be a valid 1- to 16-character CA IDMS/DB area name.

USAGE-mode is

Specifies the usage mode in which an application can ready the area. The usage mode specifies the run-time operations that an application can perform against the CA IDMS/DB database area.

If neither PROTECTED nor EXCLUSIVE is specified, SHARED is the default. SHARED specifies that other concurrently executing applications can access the named area.

PROTECTED

PROTECTED prohibits update of the area by another concurrently executing application.

EXCLUSIVE

EXCLUSIVE prohibits access to the area by another concurrently executing application.

RETRIEVAL

Permits only retrieval (read-only) access for the database area

UPDATE

Allows all DML functions (STORE, ERASE, MODIFY, etc.) for the database area

MODIFY RECORD Statement

The MODIFY RECORD statement allows you to specify a name for a generated record. The specified name overrides the name supplied by the syntax generator. Note that the default record names are derived from the corresponding DL/I segment names.

If the name of a record in the associated schema is different from the syntax generator-supplied name, you must include the MODIFY RECORD statement to supply the correct schema-specific record name.

Syntax

```
►►─┬──────────────────────────────────┬─────────────────────────────────►
   └─ MODify RECord NAME is record-name ─┘

 ►─┬──────────────────────────────────┬─────────────────────────────────►◄
   └─ NEW NAME is new-record-name ── . ─┘
```

Parameters

MODify RECord NAME is *record-name*

Identifies the record for which you want to specify a new name. *Record-name* must be a 1- to 16-character name.

NEW NAME is *new-record-name*

Specifies the new CA IDMS/DB database record name. *New-record-name* must be a valid 1- to 16-character CA IDMS/DB record name.

MODIFY SET Statement

The MODIFY SET statement allows you to specify a name for a generated set. The specified name overrides the name supplied by the syntax generator. Note that the default set names are derived from the DL/I parent segment names and their associated child segment names. The syntax generator concatenates each parent/child name pair with the literal "-".

If the name of a set in the associated schema is different from the syntax generator-supplied name, you must include the MODIFY SET statement to supply the correct schema-specific set name.

Syntax

```
►►─┬──────────────────────────────┬──────────────────────────────────────►
   └─ MODify SET NAME is set-name ─┘

 ►─┬──────────────────────────────────┬─────────────────────────────────►◄
   └─ NEW NAME is new-set-name ── . ──┘
```

Parameters

MODify SET NAME is *set-name*

Identifies the set for which you want to specify a new name. *Set-name* must be a 1- to 16-character name.

NEW NAME is *new-set-name*

Specifies the new CA IDMS/DB database set name. *New-set-name* must be a valid 1- to 16-character CA IDMS/DB set name.

Executing the CA IDMS DLI Transparency Syntax Generator

Input

As described earlier in this section, syntax generator input consists of:

- The assembled DBD and PSB control blocks
- Control, GENERATE, and modification statements

Output

Depending on the GENERATE statements coded, output from a single execution of the syntax generator consists of:

- Source statements required to create a CA IDMS/DB schema in the data dictionary
- Source statements required to create a CA IDMS/DB DMCL and/or subschema load module
- Source statements required to create one IPSB load module
- A report listing the generated source statements

You must execute the syntax generator once for each set of IPSB source statements you want to produce. To execute the syntax generator, use the JCL shown in CA IDMS DLI Transparency JCL (see page 257).

Syntax Generator Execution

The diagram below illustrates the activities involved in executing the syntax generator.

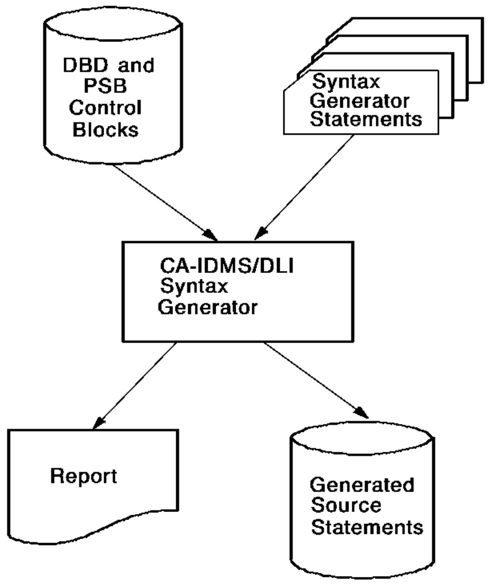

DBD and PSB Control Blocks
Syntax Generator Statements
CA-IDMS/DLI Syntax Generator
Report
Generated Source Statements

Figure 32. Syntax generator execution

Chapter 4: IPSB Compiler

This section contains the following topics:

About This Chapter

The IPSB Compiler

The CA IDMS DLI Transparency interface program specification block (IPSB) compiler converts user-supplied entries into assembler statements that are assembled into load modules, known as IPSBs. The IPSBs are later used by the CA IDMS DLI Transparency run-time interface as a source of control information for satisfying the database requests issued by a DL/I application program.

DL/I and CA IDMS/DB Correspondences

The control information in the IPSB is, in fact, a series of correspondences between DL/I structures and CA IDMS/DB structures. These correspondences serve two general purposes, as follows:

- To provide the run-time interface with the information needed to convert retrieval and update requests issued by the DL/I application program into CA IDMS/DB requests.
- To provide the run-time interface with the information needed to update the DL/I application's program communication blocks (PCBs). The updated PCBs are used to deliver the requested data and/or status information to the DL/I application program.

Topics

This section details the IPSB source statements that serve as input to the compiler. The following topics are discussed:

- Considerations for preparing IPSB compiler input
- Compiler-directive statements
- IPSB SECTION
- AREA SECTION
- RECORD SECTION
- INDEX SECTION
- PCB SECTION
- IPSB compiler execution

Considerations For Preparing IPSB Compiler Input

Input to the IPSB Compiler

Input to the IPSB compiler consists of source statements that define the correspondences between the DL/I database referenced by the application and the CA IDMS/DB database accessed by the run-time interface. The CA IDMS DLI Transparency syntax generator produces these source statements from the program specification block (PSB) used by the DL/I application.

Review Statements Before Executing the IPSB Compiler

Before inputting the generated statements to the compiler, you should review them using the material in this section. In particular, you should make sure that the generated source statements reflect the dependencies in the DL/I definitions, especially with regard to logical child/logical parent relationships.

To review the IPSB statements, you will need the original source for the DL/I PSB and DBDs and the generated CA IDMS/DB schema source. If you have to modify the IPSB statements, use the IPSB syntax presented in this section. When reviewing the IPSB source, consult the table below, for a list of the IPSB and DL/I correspondences.

Note: If you plan to use the resulting IPSB module with the load utility, there are special load considerations that you must also incorporate in the IPSB source. See CA IDMS DLI Transparency Load Utility (see page 171) for a detailed description of the IPSB load considerations.

IPSB Source Statements

In a single execution of the IPSB compiler, you can compile one IPSB. You must define and compile one IPSB for each PSB expected by a DL/I application program. The IPSB source statements are organized into five sections and must appear in the following order:

- **IPSB SECTION** -- This section relates the IPSB to the corresponding PSB.
- **AREA SECTION** -- This section identifies the CA IDMS/DB database areas, included in the subschema, that are to be readied by the CA IDMS DLI Transparency run-time interface in any usage mode other than shared retrieval (the default).
- **RECORD SECTION** -- This section names the CA IDMS/DB records to be used either explicitly or implicitly to satisfy DL/I calls and defines the DL/I fields to be referenced in parameter lists used in the application program.
- **INDEX SECTION** -- This section provides the information necessary to relate CA IDMS/DB records and sets to secondary index and HIDAM index structures to be used and/or maintained by the CA IDMS DLI Transparency run-time interface.
- **PCB SECTION** -- This section corresponds to the associated DL/I PCBs within a PSB.

Section Titles and Statements

The syntax generator automatically produces section titles and appropriate statements for each section. Each section must appear in every IPSB. Even if there are no statements for a specific section, do not remove the section title. In addition to the sections, you can include compiler-directive statements before any of the IPSB sections. Note that the syntax generator does not produce the compiler-directive statements for you.

Locating IPSB Entries Within PSB and DBDs

Although the IPSB input is free form, you must locate specific information within the PSB and DBDs. To simplify this task, the table below,

- Lists each IPSB clause by section (see IPSB SECTION (see page 100)).
- Identifies the DL/I DBD or PSB statement and operand to which the clause corresponds; and indicates those clauses that specify information pertinent only to CA IDMS/DB.
- The syntax rules for each statement contain, where necessary, references to pertinent DL/I parameters in the PSB and DBDs.

For more information about locating IPSB entries within the PSB and DBDs, see DL/I and CA IDMS/DB (see page 21).

IPSB Input			DBD or PSB Correspondence		
Section	**Statement**	**Clause**	**Phase**	**Statement**	**Operand**
IPSB	IPSB	NAME	PSB	PSBGEN	PSBNAME=
		OF SUBSCHEMA	*		
		LANGUAGE	PSB	PSBGEN	LANG=
		IOAREA	PSB	PSBGEN	IOASIZE=
		SSA	PSB	PSBGEN	SSASIZE
		COMPATIBILITY	PSB	PSBGEN	COMPAT=
AREA	AREA		*		
RECORD	RECORD	NAME	*		
		LENGTH	DBD	SEGM	BYTES=
RECORD	FIELD	NAME	DBD	FIELD	NAME= *fldname1*
		STARTING	DBD	FIELD	START=
		LENGTH	DBD	FIELD	BYTES=
		USAGE	DBD	FIELD	TYPE=
INDEX	INDEX	NAME	DBD	XDFLD	NAME=
			in indexed database (for secondary indexes)		
			DBD	DBD	NAME=
			in INDEX database (for HIDAM)		
		USING INDEXED-SET	*		

IPSB Input			DBD or PSB Correspondence		
Section	**Statement**	**Clause**	**Phase**	**Statement**	**Operand**
		TARGET	DBD	LCHILD	NAME=
			in INDEX database		
		POINTER	DBD	SEGM	NAME=
			in INDEX database		
		THRU SET	*		
		SOURCE	DBD	XDFLD	SEGMENT=
			in indexed database (for secondary indexes)		
			DBD	SEGM	NAME
			in HIDAM database (for HIDAM)		
		CONSTANT	DBD	XDFLD	CONST=
		SEARCH	DBD	XDFLD	SRCH=
			in indexed database (for secondary indexes)		
			DBD	FIELD	NAME=
			in HIDAM database (for HIDAM database)		
		SUBSEQUENCE	DBD	XDFLD	SUBSEQ=
		DUPLICATE	DBD	XDFLD	DDATA=
		NULL VALUE	DBD	XDFLD	NULLVAL=
		EXIT ROUTINE	DBD	XDFLD	EXTRTN=
PCB	PCB	ACCESS	DBD	DBD	ACCESS=
		DBDNAME	DBD	DBD	NAME=
		OPTIONS	PSB	PCB	PROCOPT=
		POSITIONING	PSB	PCB	POS=
		SEQUENCE	*		
PCB	SEGMENT	NAME	DBDGEN	SEGM	NAME=
		RECORD	*		
		PARENT	DBDGEN	SEGM	PARENT= *segname2*
		THRU SET	*		

IPSB Input			DBD or PSB Correspondence		
Section	**Statement**	**Clause**	**Phase**	**Statement**	**Operand**
		LOGICAL DEST PARENT	DBDGEN	SEGM	PARENT= *lpsegname*
		PHYSICAL DEST PARENT	DBDGEN	LCHILD	NAME=
		INSERT/ REPLACE RULES	DBDGEN	SEGM	RULES=
			(combined from a logical and physical database)		
		USE	DBDGEN	SEGM	SOURCE=

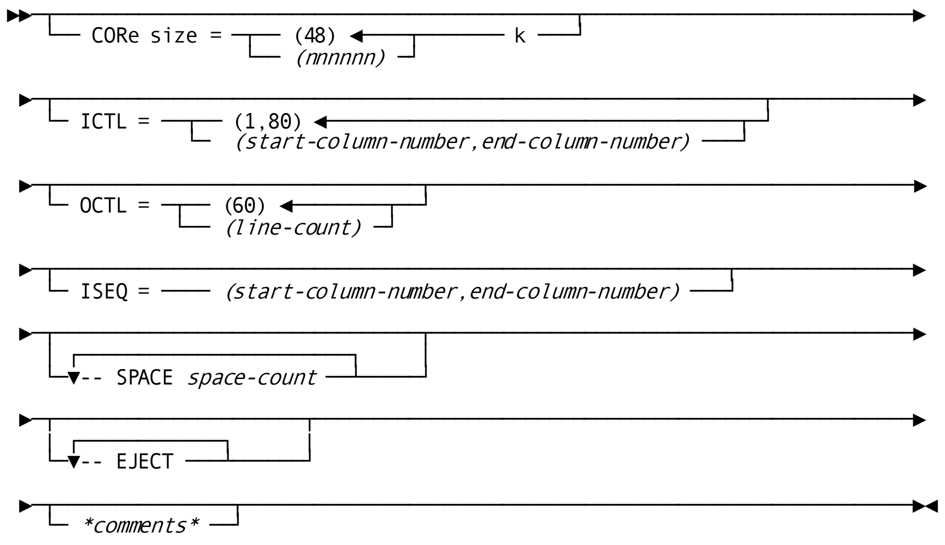
Note: *For CA IDMS/DB use only.

Compiler-Directive Statements

IPSB compiler-directive statements allow you to:

- Specify the amount of storage required by the IPSB compiler to compile the IPSB
- The range of input columns in which IPSB statements can be coded
- Sequence checking of input to the ISPB compiler
- Formatting of reports output by the IPSB compiler

Syntax



Parameters

CORE *size=nnnnnn k*

Specifies the amount of storage the IPSB compiler is to acquire (by a GETMAIN under OS and a GETVIS or COMREG under z/VSE) for the IPSB being generated.

Nnnnnn is a 1- to 6-digit numeric value.

If the optional K is included, the amount of storage acquired is *nnnnnn* increments of K (1,024 bytes). If K is omitted, *nnnnnn* represents the actual number of bytes of storage acquired, which the compiler rounds up to the next doubleword.

If this statement is omitted, the IPSB compiler acquires 48K of storage.

ICTL=*(start-column-n,end-column-n)*

Specifies the columns within which IPSB input statements can be coded. This compiler-directive statement, if coded, must precede the input for the five IPSB sections.

Valid values for both *start-column* and *end-column* are 1 through 80.

The default values for *start-column* and *end-column* are 1 and 80, respectively.

OCTL=*(line-count-number)*

Specifies the number of lines to print per page of printed output. If coded, this compiler-directive statement must precede the input for the five IPSB sections.

The default value for *line-count* is 60: acceptable values are 1 through 66.

ISEQ=*(start-column-number,end-column-number)*

Specifies that the compiler is to perform sequence checking on all input and specifies the start and end columns of the sequence number generated for each input statement.

If coded, this statement must precede all IPSB input statements. If this statement is omitted, sequence checking is not performed.

Valid values for *start-column-number* and *end-column-number* are in the range 1 through 80. The minimum allowable difference between the entry for *start-column* and the entry for *end-column* is 10.

SPACE=*space-count*

Directs the compiler to skip the specified number of lines on the output report. Only one blank is allowed between SPACE and the value specified for *space-count*.

Acceptable values for *space-count* are 1 through 9. Several SPACE statements can appear in the compiler input.

EJECT

Directs the compiler to stop printing the current page and begin printing a new page. This statement must be on a line by itself and can be interspersed among IPSB input control statements (that is, EJECT statements can appear throughout compiler input).

comments*

Directs the compiler to interpret subsequent characters as comments.

Comments can be embedded in IPSB statements and are terminated automatically at the end of the input line, unless the compiler encounters a second asterisk (*) in the input line, which causes explicit termination.

Be sure to keep track of the number of asterisks. An odd number turns on comment text; an even number turns it off.

Example

```
ICTL=(1,72)
OCTL=(45)
ISEQ=3,72
EJECT
SPACE 2
*Begin comments with an asterisk
```

Figure 33. Sample compiler-directive statements

IPSB SECTION

The IPSB SECTION relates the IPSB to a particular PSB expected by the DL/I application program in a native DL/I environment. The IPSB section contains one statement--the IPSB statement. This statement identifies the IPSB and specifies global information related to the corresponding PSB.

The information supplied in the IPSB SECTION corresponds to the information that is specified to DL/I by the PSBGEN statement in the PSB phase. The PSBGEN statement is located at the end of the PSB.

Syntax

```
►►─── IPSB SECTION ── . ────────────────────────────────────────────────────►

►─── IPSB name is ipsb-name ────── of SUBSchema subschema-name ─────────────►

►─┬────────────────────────────────────┬────────────────────────────────────►
  └── LANGuage is ───┬─ CObol ◄ ──┬────┘
                     ├─ PL/i ─────┤
                     └─ ASsembler ┘

►─┬─────────────────────────────────────────────────┬───────────────────────►
  └── MAXimum IOAREA size is maximum-io-area-size ──┘

►─┬────────────────────────────────────────────┬────────────────────────────►
  └── MAXimum SSA size is maximum-ssa-size ────┘

►─┬───────────────────────────────────────────┬─────────────────────────────►◄
  └── COMPATibility is ──┬─ yes ──┬──── . ────┘
                         └─ no ◄ ─┘
```

Parameters

IPSB SECTION

IPSB SECTION must be the first entry in the IPSB section, followed by one IPSB statement.

IPSB name is *ipsb-name*

Identifies the IPSB being generated.

Ipsb-name is the 1- to 8-character PSB name used by the application program in a native DL/I environment. When the IPSB is link edited, the load module or phase name is the same as the *ipsb-name*.

of SUBSchema *subschema-name*

Identifies the subschema to be used by the CA IDMS DLI Transparency run-time interface.

Subschema-name is the 1- to 8-character name of the subschema used by CA IDMS DLI Transparency to access the CA IDMS/DB database.

LANGuage IS *CObol/ PL/i /ASsembler*

Specifies the programming language of the application program using this IPSB. The language specified in the LANGUAGE parameter of the PSBGEN statement must be entered. The default is COBOL.

MAXimum IOAREA size is *maximum-io-area-size*

Specifies the amount of space to be allocated for the application program's I/O area.

If this clause is omitted, the compiler calculates this size as the total length of all sensitive segments in the longest possible path call issued by programs using this IPSB. Include this clause if a value is specified in the IOASIZE parameter of the PSBGEN statement.

If the parameter is missing from the PSBGEN statement, the compiler calculates the space to be allotted for the application program's I/O area. Refer to the appropriate DL/I documentation for further details on I/O area allocation.

MAXimum SSA size is *maximum-ssa-size*

Specifies the maximum total length of all segment search argument (SSA) strings to be used in a given DL/I call issued by programs using this IPSB.

If this clause is omitted, the compiler calculates the size as 280 times the maximum number of levels associated with any PCB statement within this IPSB. Include this clause if a value is specified in the SSASIZE parameter of the PSBGEN statement.

If this parameter is missing from the PSBGEN statement, the compiler calculates the maximum SSA size. Refer to the appropriate DL/I documentation for further details on SSA size specification.

COMPATibility is yes/no

Specifies whether the application program expects to find an I/O PCB in the PSB.

The default is NO.

If YES is specified, the CA IDMS DLI Transparency run-time interface creates a dummy I/O PCB as the first PCB. Note that you do not define this dummy I/O PCB, nor is it to be used by the application program. You should include this clause if CMPAT=YES is specified in the PSBGEN statement; otherwise, the compiler uses the default.

Usage

The PSBGEN statement in this example serves as the source for the IPSB SECTION.

In this example, the IPSB has a name of PSB1 and relates to DL/I requests to structures in the CA IDMS/DB subschema SUBSCH1. As indicated in the PSBGEN statement:

- The application program is written in COBOL
- IOSIZE=2000
- SSASIZE=1500
- CMPAT=NO

The PSBGEN statement values above correspond in the IPSB SECTION to:

- IOAREA SIZE IS 2000
- MAX SSA SIZE IS 1500
- COMPATIBILITY IS NO

```
              DL/I PSBGEN Statement

PSBGEN  LANG=COBOL,PSBNAME=PSB1,MAXQ=0,CMPAT=NO,IOSIZE=2000,
            SSASIZE=1500

              CA IDMS DLI Transparency IPSB Section

IPSB SECTION.
    IPSB NAME IS PSB1 OF SUBSCHEMA SUBSCH1
    LANG IS COBOL MAX IOAREA SIZE IS 2000
    MAX SSA SIZE IS 1500 COMPATIBILITY IS NO.
```

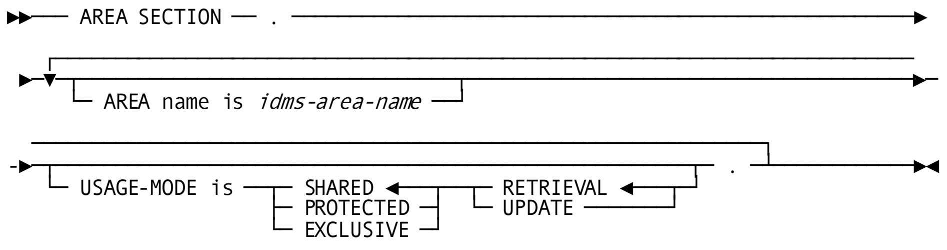
Figure 34. Sample DL/I PSBGEN and IPSB SECTION

AREA SECTION

The AREA SECTION identifies the CA IDMS/DB database areas that are to be readied by the CA IDMS DLI Transparency run-time interface in any usage mode other than shared retrieval (the default). Specify one AREA SECTION statement for each database area that is not to be readied in shared retrieval mode.

Note: Make sure that all database areas to be accessed by the run-time interface are included in the subschema. The run-time interface automatically readies those areas required by this IPSB. Areas included in the subschema but not required by the IPSB are not readied.

Syntax



Parameters

AREA SECTION.

AREA SECTION must be the first entry in the section, followed by as many AREA statements as required. You must include the AREA SECTION clause whether or not the section contains any AREA statements.

AREA name is *idms-area-name*

Identifies the CA IDMS/DB database area to be readied.

Idms-area-name is the 1- to 16-character area name and included in the subschema named in the IPSB SECTION.

USAGE-MODE is

Specifies the usage mode in which the run-time interface is to ready the named area. The usage mode options specify the conditions for readying and accessing the named area.

PROTECTED

Specifies that the named area, once readied by CA IDMS/DB, cannot be readied in update usage mode by other concurrently executing run units.

EXCLUSIVE

Specifies that the named area, once readied by CA IDMS/DB, cannot be accessed by other concurrently executing run units.

RETRIEVAL

Specifies that the named area is to be readied for retrieval only. Other concurrently executing run units can ready the area in any usage mode other than one that is qualified as EXCLUSIVE.

RETRIEVAL is the default.

UPDATE

Specifies that the named area is to be readied for both retrieval and update. Other concurrently executing run units can ready the area in any usage mode other than one that is qualified as EXCLUSIVE or PROTECTED.

Example

```
 AREA SECTION.
      AREA NAME IS IDMSDB-1
           USAGE-MODE IS EXCLUSIVE UPDATE.
      AREA NAME IS SPFAREA1.
      AREA NAME IS SPFAREA2
           USAGE-MODE IS PROTECTED UPDATE.
      AREA NAME IS IDSMDB-2
           USAGE-MODE IS RETRIEVAL.
```

Figure 35. Sample AREA SECTION

RECORD SECTION

The RECORD SECTION names the CA IDMS/DB records to be used either explicitly or implicitly to satisfy DL/I calls, and defines the DL/I fields to be referenced in SSA parameter lists used in DL/I database requests.

The RECORD SECTION consists of RECORD statements and FIELD statements.

The RECORD SECTION draws upon information in the:

- CA IDMS/DB schema
- DL/I PSB

  Since each PSB requires a separate IPSB, the information in one PSBGEN statement is used to complete each RECORD SECTION.
- DL/I DBDs

  The DBDs required are those specified in the PCBs. You should have available all of the DBDs specified in each PCB within the PSB.

If a PCB calls for a logical database or an index database, you also need the DBDs for the associated physical databases or indexed databases, respectively.

When a PCB calls for a HIDAM database or a database with a secondary index(es), you should have available the DBD for the associated index database.

Syntax

```
►►── RECORD SECTION. ────────────────────────────────────────────────►

 ►── RECORD statements. ─────────────────────────────────────────────►

 ►── FIELD statements. ─────────────────────────────────────────────►◄
```

Parameters

RECORD SECTION.

RECORD SECTION must be the first entry in the section followed by RECORD and FIELD statements.

RECORD statements

Following the RECORD SECTION is one RECORD statement for each DL/I segment specified in every PCB in the application program's PSB. The RECORD statement defines the CA IDMS/DB record that corresponds to the DL/I segment.

In addition to these **explicit correspondences**, you must make sure that there are RECORD statements for those records whose corresponding segments are not specified in the PCBs but must be accessed by DL/I to process DL/I calls. These **implicit correspondences** are required for the following types of segments:

- Dependent segments of any segment specified in the PCB if the specified parent segment can be deleted (that is, PROCOPT=A or PROCOPT=D appears in the PCB or SENSEG statement)
- All dependent segments of the preceding dependent segments
- Pointer segments for all target and source segments specified in the PCB
- Source segments for all target segments specified in the PCB
- All segments in the hierarchical path of the destination parent segment in its physical database

FIELD statements

There can be from 0 to 255 FIELD statements following each RECORD statement. The sources for these FIELD statements consist of the appropriate FIELD statements within the relevant DBDs.

The RECORD and FIELD statements are discussed in detail below.

RECORD Statement

A RECORD statement names a CA IDMS/DB record and optionally specifies either the type of CA IDMS/DB ERASE command issued or that a DISCONNECT command was issued. The CA IDMS/DB ERASE or DISCONNECT command is issued in response to a DL/I DLET call for the segment corresponding to the named record.

To determine the RECORD statements required for an IPSB:

1. Locate the PSB that corresponds to the IPSB being coded
2. Locate the PCBs in this PSB
3. If a PCB names a physical DBD (that is, with ACCESS=HDAM, HSAM, HISAM, HIDAM, or INDEX), use the following guidelines:
   - Locate the DBD named in the PCB.
   - Prepare for each DBD a hierarchy diagram showing each segment defined in the DBD.
   - Check off all the segments specified in each PCB within the PSB. Each of these segments will need a corresponding CA IDMS/DB record, which is to be described in a RECORD statement.
   - Check off all those segments in the hierarchy diagrams that meet one of the following conditions:
     - The segment is a dependent segment of any segment that is both specified in the PCB and is subject to deletion.
     - The segment is a source segment associated with a target segment that is specified in the PCB.
     - The segment is a pointer segment associated with a target segment that is a segment specified in the PCB or a dependent of a segment specified in the PCB. The pointer segment for a target segment is located in the associated index DBD.
4. If the PCB names a logical DBD (that is, with ACCESS=LOGICAL), use the following guidelines:
   - Find both the logical DBD and the associated physical DBDs.
   - Note each SEGM statement in the logical DBD with only one SOURCE parameter. In each of these SEGM statements, the SOURCE parameter identifies the segment in the physical database. Identify the corresponding CA IDMS/DB record for each of these physical segments.
   - Locate each SEGM statement that defines a concatenated segment. Identify the real logical child segment and the destination parent segment and locate the names of their corresponding CA IDMS/DB records.

- Prepare hierarchy diagrams of the two associated physical databases. Using the diagram containing the destination parent segment, check off all the segments in the hierarchical path of the destination parent segment. For each of the checked off segments, identify the corresponding CA IDMS/DB record.
- Note if any of the identified segments from the above guidelines can be deleted. If this is the case, note all of the dependents of this segment. (Do not include virtual logical child segments.) For each noted segment, identify the corresponding CA IDMS/DB record.
- Note if any of the segments identified in the above guidelines is a source segment or a target segment of either a HIDAM database or a secondary index. If this is the case, locate the associated index pointer segment, which is defined in a DBD with ACCESS=INDEX. Then, identify the corresponding CA IDMS/DB record for the index pointer segment.
- Make sure there is a RECORD statement for each of the records identified in the above guidelines.

Syntax

```
►►─── RECORD SECTION ── . ────────────────────────────────────────────────►

 ►─── RECORD name is idms-record-name ────────────────────────────────────►

 ►─── LENGTH is ─┬─ dl1-segment-length ───────────────────────────────────►
                 └─ dl1-max-segment-length dl1-min-segment-length ─┘

 ►─┬──────────────────────────────────────────────────────────────────┬──►◄
   └─ DELete by ──┬── ERASE ALL ◄──────── . ──────┘
                  ├── ERASE PERManent ─┤
                  ├── ERASE SELective ─┤
                  └── DISConnect ──────┘
```

Parameters

RECORD name is *idms-record-name*

Identifies the CA IDMS/DB record to be accessed by the CA IDMS DLI Transparency run-time interface.

Idms-record-name must be a 1- to 16-character name that corresponds to a DL/I segment and must be defined in the subschema named in the IPSB SECTION.

LENGTH is

Specifies the length of the DL/I segment to which the *idms-record-name* corresponds.

dl1-segment-length

Specifies the length of the DL/I segment if it is a fixed-length segment.

dl1-max-segment-length dl1-min-segment-length

Specifies the maximum and minimum lengths of the DL/I segment if it is a variable-length segment. See "Determining values for variable length segments" under "Examples" later in this chapter.

DELete by

Specifies the CA IDMS/DB DML command that the interface will issue in response to a DL/I DLET call for the segment corresponding to the named record.

ERASE ALL

Specifies that the named record and all mandatory and optional member record occurrences it owns are to be erased.

All members that are owners of any set occurrences are treated as if they were the object of an ERASE ALL statement.

ERASE ALL is the default.

ERASE PERManent

Specifies that the named record and all mandatory member record occurrences it owns are to be erased from the database. Optional member record occurrences are disconnected.

All erased mandatory members that are owners of set occurrences are treated as if they were the object of an ERASE PERMANENT statement.

Note: For more information about CA IDMS/DBset membership options, see the *CA IDMS Database Administration Guide*

ERASE SELective

Specifies that the named record and all mandatory member record occurrences it owns are to be erased from the database. Optional member record occurrences are erased only if they do not currently participate as members in other set occurrences.

All erased members that are owners of set occurrences are treated as if they were the object of an ERASE SELECTIVE statement.

DISConnect

Specifies that the membership of the named record is cancelled from all sets in which it currently participates as an optional member. The record, however, remains in the database.

Usage

Determining the Value for a Fixed Length Segment

To locate the *dl1-segment-length*, find the SEGM statement defining the segment that corresponds to the named record. Use the entry in the SEGM statement's BYTES clause for *dl1-segment-length*.

Note that if the DL/I segment is a logical child segment, the length of the physical and/or logical parent concatenated key may be required along with the BYTES clause entry when determining the value of *dl1-segment-length*.

Determining Values for Variable Length Segments

To locate *dl1-max-segment-length* and *dl1-min-segment-length* values, find the SEGM statement defining the segment that corresponds to the named record. Use the first entry in the SEGM statement's BYTES clause for *dl1-max-segment-length*; use the second entry in the SEGM statement's BYTES clause for *dl1-min-segment-length*.

Note that if the DL/I segment is a logical child segment, the length of the physical and/or logical parent concatenated key may be required along with the BYTES clause entries when determining the value for *dl1-max-segment-length* and *dl1-min-segment-length*.

Calculating the Length of a Concatenated Key

The length of a concatenated key equals the sum of the lengths of the sequence field, from the sequence field of the named key through the root segment's sequence field.

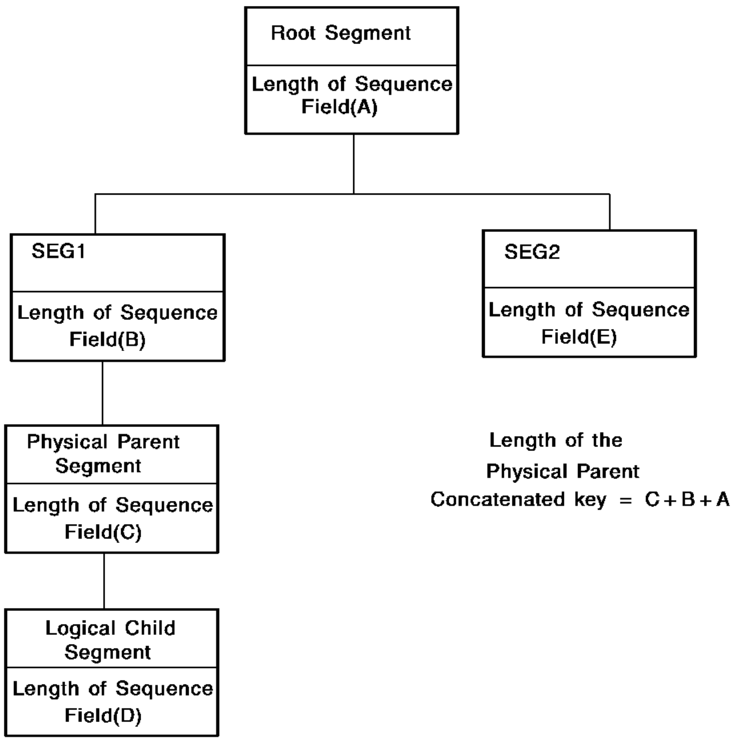

Root Segment
Length of Sequence Field(A)
SEG1
Length of Sequence Field(B)
SEG2
Length of Sequence Field(E)
Physical Parent Segment
Length of Sequence Field(C)
Length of the Physical Parent Concatenated key = C + B + A
Logical Child Segment
Length of Sequence Field(D)

Figure 36. Finding the length of a concatenated key

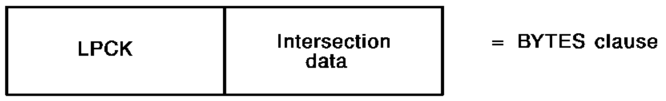
Determining Record Length for Logical Child Equivalent

The examples below show how you can determine the record length for the logical child equivalent.

Refer to "LOGICAL PARENT FIELD Statement" later in this section for details on determining whether the physical parent concatenated key and the logical parent concatenated key are stored virtually or physically.

Example 1

Assume the LPCK is stored virtually and the PPCK is stored physically.



1. Find the LPCK's length. Subtract this key length from the entry(ies) in the logical child's BYTES clause.
2. Find the PPCK's length. Add this key length to the value calculated in step 1 above:

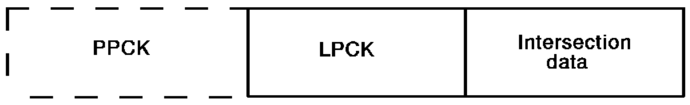
   For fixed-length segments:

   dl/i-segment-length = (BYTES entry - LPCK-length) + PPCK-length

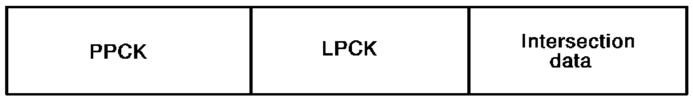
   For variable-length segments:

   dl/1-max-segment-length = (First BYTES entry - LPCK-length) + PPCK-length
   dl/1-min-segment-length = (Second BYTES entry - LPCK-length) + PPCK-length

Example 2



Assume both the PPCK and the LPCK are stored virtually.



Find the LPCK's length. Subtract this key length from the entry(ies) in the logical child's BYTES clause:

For fixed length segments:

dl/i-segment-length = BYTES entry - LPCK-length

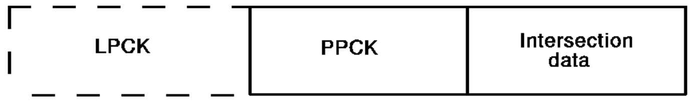
For variable length segments:

dl/1-max-segment-length = First BYTES entry - LPCK-length

dl/1-min-segment-length = Second BYTES entry - LPCK-length

Example 3

Assume that the logical parent concatenated key (LPCK) is stored physically and the physical parent concatenated key (PPCK) is stored virtually.



Use the BYTES parameter value(s) in the logical child's SEGM statement as the value(s) for the LENGTH parameter:

For fixed length segments:

dl/i-segment-length = BYTES entry in logical child's SEGM statement

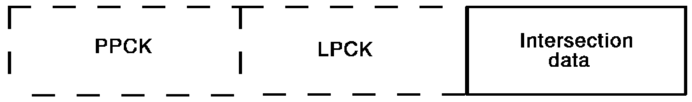
For variable length segments:

dl/1-max-segment-length = First BYTES entry in logical child's SEGM statement

dl/1-min-segment-length = Second BYTES entry in logical child's SEGM statement

Example 4

Assume both the LPCK and the PPCK are stored physically.



Find the PPCK's length. Add this key length to the entry(ies) in the logical child's BYTES clause:

For fixed length segments:

dl/i-segment-length = PPCK-length + BYTES entry in logical child's SEGM statement

For variable length segments:

dl/1-max-segment-length = PPCK-length + first BYTES entry in logical child's SEGM statement

dl/1-min-segment-length = PPCK-length + second BYTES entry in logical child's SEGM statement

FIELD Statement

A FIELD statement defines a DL/I field within the named record and corresponds to the FIELD statement in the DBD. Following each RECORD statement, there must be a FIELD statement for every field listed in the DBD for the segment corresponding to the named record. Some records (that is, those corresponding to the logical child segments) will need additional FIELD statements, as explained below. Up to 255 FIELD statements can follow each RECORD statement. If, however, a named record corresponds to a segment for which no fields are defined in the DBD, the RECORD statement stands alone without any FIELD statements.

Five FIELD Statement Formats

There are five FIELD statement formats available:

- **Sequence** -- Defines DL/I sequence fields
- **Field** -- Defines DL/I search fields other than sequence fields
- **Logical parent** -- Defines logical parent concatenated key fields
- **Physical parent** -- Defines physical parent concatenated key fields
- **Logical sequence** -- Defines logical sequence fields (that is, sequence fields for the virtual logical child segments)

How to Determine the Appropriate FIELD Statement Format

To determine which format of the FIELD statement is appropriate to define a particular DL/I field, first consider the segment equivalent of the record being described in the RECORD statement. Find the SEGM statement defining the segment and determine whether the segment is a root segment, a dependent segment (that is, with only one parent), or a logical child segment (that is, with two parents). After making this determination, apply the appropriate set of rules as follows:

- **Root and dependent segments** -- If the segment is either a root segment or a dependent segment, note its sequence field (if any). Define this sequence field by using the SEQUENCE FIELD statement. This FIELD statement must appear immediately following the appropriate RECORD statement. Next, determine if the segment has search fields (that is, fields defined without a SEQ in the NAME clause of the FIELD statement). If there are search fields, each one must be defined by using the FIELD statement. Each of these FIELD statements must appear under the appropriate RECORD statement.

- **Logical child segment** -- If the segment is a logical child segment, the RECORD statement must be followed by LOGICAL PARENT FIELD and PHYSICAL PARENT FIELD statements to define the logical parent concatenated key field and the physical parent concatenated key field, respectively. Additionally, the logical child segment corresponding to the named record may have a sequence field. If so, define this sequence field with a SEQUENCE FIELD statement following the RECORD statement.

Define Search Fields with Separate FIELD Statement

Define each of the segment's search fields to the IPSB with a separate FIELD statement following the LOGICAL PARENT FIELD and PHYSICAL PARENT FIELD statements that define the logical parent concatenated key and the physical parent concatenated key, respectively. Next, locate the SEGM statement that defines the associated virtual logical child segment. This SEGM statement is generally not located in the same DBD as the SEGM statement that defines the logical child segment.

If the virtual logical child segment has a sequence field, a LOGICAL SEQUENCE FIELD statement is required to define the sequence field under the named record. For each of the remaining search fields for the virtual logical child segment, there must be a FIELD statement. Each of these FIELD statements must appear under the RECORD statement that identifies the record corresponding to the logical child segment.

USAGE Clause

Each of the five formats of the FIELD statement can end with the optional USAGE clause. As with DL/I, this clause is for documentation purposes only. This clause and the five FIELD statement formats are described separately below.

USAGE clause

The USAGE clause defines the data type of the named field. It is used at the *end* of each of the five formats of the FIELD statement and is not repeated for the individual formats of the FIELD statement.

Syntax

```
>─┬─────────────────────────────────────────┬──────────────────────────────────────────────><
  └─ USAGE is ──┬── DISplay ◄──┬─ . ─┘
                ├── BINary ────┤
                └── PACKed ────┘
```

Parameters

USAGE is

Specifies the data type of the named field. To determine the appropriate option, note the FIELD statement in the DBD.

DISplay

Specify if the FIELD statement in the DBD specifies TYPE=C. DISPLAY is the default value.

BINary

Specify if the FIELD statement in the DBD specifies TYPE=F, TYPE=H, or TYPE=X.

PACKed

Specify if the FIELD statement in the DBD specifies TYPE=P.

SEQUENCE FIELD statement

This format of the FIELD statement defines the sequence field for the named record. A sequence field can be defined for:

- Each record corresponding to a root segment
- A dependent segment ordered under its physical parent, including the logical child segment
- A pointer segment

Sequence fields defined for pointer records must comprise the concatenation of the constant, search, and subsequence fields for the pointer segment. (Constant and subsequence fields are described below.)

A field defined as a sequence field can be used as a search field in an SSA.

Syntax

```
►►─┬──────────────────────────────────┬────────────────────────────►
   └ SEQuence FIELD name is dl1-field-name ┘

 ►─┬──────────────────────────────────────┬────────────────────────►
   └ STARTING POSition is starting-position ┘

 ►─┬──────────────────────────┬────────────────────────────────────►◄
   └ LENgth is dl1-field-length ┘
```

Parameters

SEQuence FIELD name is *dl1-field-name*

Specifies the name of the sequence field. *Dl1-field-name* is the entry in the NAME clause of the DL/I FIELD statement defining the sequence field.

STARTING POSition is *starting-position*

Specifies the position in the record in which the sequence field begins. Use the START clause value in the DL/I FIELD statement defining the sequence field.

LENgth is *dl1-field-length*

Specifies the length of the sequence field. Use the BYTES clause entry in the DL/I FIELD statement defining the sequence field.

FIELD statement

This format of the FIELD statement defines the named record's search fields (that is, the search fields other than the sequence fields).

There must be a separate statement to define each search field in each record that corresponds to a segment with search fields.

For a record corresponding to a logical child segment, this format defines the search fields for the logical child segment and for the virtual logical child segment.

DL/I fields whose names begin with /CK or /SX are treated like any other search fields and are defined with this format of the FIELD statement.

Syntax

```
►►─┬──────────────────────────┬────────────────────────────────────►
   └ FIELD name is dl1-field-name ┘

 ►─┬──────────────────────────────────────┬────────────────────────►
   └ STARTING POSition is starting-position ┘

 ►─┬──────────────────────────┬────────────────────────────────────►◄
   └ LENgth is dl1-field-length ┘
```

Parameters

FIELD name is *dl1-field-name*

Names the DL/I field being defined. Use the NAME clause entry in the DL/I FIELD statement that defines the search field for the segment corresponding to the named record.

Ensure that *dl1-field-name* is identical to the field name by which the DL/I application will refer to the field.

STARTING POSition is *starting-position*

Specifies the position in the record in which the search field begins. Use the START parameter value in the DL/I FIELD statement that defines the search field. Omit this field if the name field is a /SX field.

LENgth is *dl1-field-length*

Specifies the length of the search field. Use the BYTES parameter value in the DL/I FIELD statement that defines the search field.

LOGICAL PARENT FIELD statement

This format of the FIELD statement defines the logical parent concatenated key field for a record corresponding to a logical child segment. A logical parent concatenated key is a symbolic pointer to the logical parent.

LOGICAL PARENT FIELD statements and PHYSICAL PARENT FIELD statements (for defining the physical parent concatenated key field) are both required when the named record corresponds to a logical child segment.

Syntax

```
►►─┬─────────────────────────────────────────────────────────────┬──────►
   └ LOGical PARENT CONCATenated KEY FIELD name is dl1-field-name ┘

 ►─┬────────────────────────────────┬───────────────────────────────────►
   └ STORED ──┬── PHYSically ◄──┬──┘
              └── VIRTually ────┘

 ►─┬──────────────────────────────────────┬─────────────────────────────►
   └ STARTING POSition is starting-position ┘

 ►─┬────────────────────────────┬──────────────────────────────────────►◄
   └ LENgth is dl1-field-length ┘
```

Parameters

LOGical PARENT CONCATenated KEY FIELD name is *dl1-field-name*

Specifies the name by which the concatenated key to the logical parent segment is defined to DL/I. Any 1- to 8-character name can be used for the *dl1-field-name*, since this name serves only as a filler.

Ensure that the name selected for *dl1-field-name* is not used to define any other field for the named record.

STORED PHYSically/VIRTually

Specifies whether the logical parent concatenatecd key is stored with the record corresponding to the logical child segment or is built by the CA IDMS DLI Transparency run-time interface.

PHYSically

Specifies that the logical parent concatenated key is stored with the record corresponding to the logical child segment. The use of this option for the segment corresponding to the named record depends on the type of logical relationship defined in the relevant DBDs as follows:

Relationship	What to specify
Unidirectional and bidirectional virtual logical relationships	If PHYSICAL or P is specified on the SEGM statement PARENT parameter defining the real logical child segment, specify PHYSICALLY.
For bidirectional physical logical relationships, the relationship must appear like a bidirectional virtual logical relationship. Choose one logical child segment to represent the real logical child segment and the other to represent the logical virtual child segment. Hence, the parent of the assigned real logical child segment is considered the physical parent segment; the parent of the assigned virtual logical child segment is considered the logical parent segment.	If the entry in the PARENT parameter of the SEGM statement defining the segment assigned as the real logical child segment is PHYSICAL or P, specify PHYSICALLY.

The default for this IPSB clause is PHYSICALLY.

If either of the destination parent concatenated key fields is STORED PHYSICALLY, that field must be the first field in the record.

If both destination parent concatenated key fields are STORED PHYSICALLY (see note below), they must be the first two fields in the record. These however, can be preceded by the halfword-length field if the record is a variable-length record. If PHYSICALLY is specified, the STARTING POSITION clause (see below) must be included in the FIELD statement.

VIRTually

Specifies that the logical parent concatenated key is absent from the record corresponding to the logical child segment and is built by the run-time interface. The use of this option for the segment corresponding to the named record depends on the type of logical relationship defined in the relevant DBDs, as follows:

Relationship	What to specify
Unidirectional and bidirectional virtual logical relationships	Specify VIRTUALLY if VIRTUAL or V is specified in the PARENT parameter of the SEGM statement defining the real logical child segment.
For bidirectional physical logical relationships, the relationship must appear as a bidirectional virtual logical relationship, as described under bidirectional physical logical relationships above.	If the entry in the PARENT parameter of the SEGM statement defining the segment assigned as the real logical child segment is VIRTUAL or V, specify VIRTUALLY. If VIRTUALLY is specified, you must omit the STARTING POSITION clause.

Note: Although DL/I bidirectional virtual relationships permit only the logical parent concatenated key to be stored physically in the logical child, CA IDMS DLI Transparency allows either one or both of the concatenated keys to be stored physically or virtually.

STARTING POSition is *starting-position*

Specifies the position in the record in which the concatenated key field begins, where the record begins in position 1. What you specify on this clause depends upon what you specified on the STORED VIRTUALLY/PHYSICALLY clause:

STORED VIRTUALLY/PHYSICALLY clause	STARTING POSITION clause
If STORED VIRTUALLY is specified for the named field	Don't include it for the named field.
If STORED PHYSICALLY is specified only for the named field (that is, only for the LOGICAL PARENT CONCATENATED KEY field)	START POSITION IS 1.

STORED VIRTUALLY/PHYSICALLY clause	STARTING POSITION clause
If both the named field and the PHYSICAL PARENT CONCATENATED KEY field (PHYSICAL PARENT FIELD statement) are STORED PHYSICALLY	One of the two fields will have a START POSITION of 1. The other field will begin in the next available byte after its complement concatenated key field is stored. For example, assume that the length of the concatenated key for the physical parent is 15 and the STARTING POSITION entered in the IPSB for the PHYSICAL PARENT is 1. Therefore, the LOGICAL PARENT KEY field has a START POSITION of 16.
When both concatenated keys are stored physically and the record is a variable length record	Perform the above calculations and add 2 to the start position to allow for the halfword containing the length of the record.

LENgth is *dl1-field-length*

Specifies the length of the concatenated key for the logical parent.

To determine the entry for this clause, first find the DL/I FIELD statements that define the sequence fields of the logical parent segment and of those segments in the logical parent's hierarchical path to the root segment. Add the BYTES clause entries in these FIELD statements.

Usage

The Length of the Concatenated Key for the Logical Parent

To calculate the length of the concatenated key for the logical parent, assume the DBD has the following entries from the root segment through the sequence field of the logical parent segment:

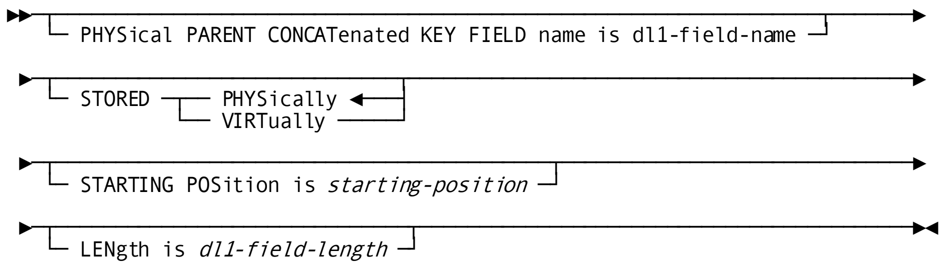
```
SEGM     NAME=SEGRT,PARENT=0,BYTES=31,PTR=TWINBWD
FIELD    NAME=(FIELD1,SEQ,U),BYTES=21,START=,TYPE=C
FIELD    NAME=FIELD2,BYTES=10,START=22
SEGM     NAME=LPSEG,PARENT=SEGRT,BYTES=20,PTR=TWINBWD
FIELD    NAME=(FIELD3,SEQ,U),BYTES=60,START=1,TYPE=C
```

In this example, the sum of the sequence fields (FIELD1 and FIELD3) is 81, which is the value entered in the LENGTH clause of the IPSB FIELD statement. For more details, see Figure 36.

PHYSICAL PARENT FIELD statement

This format of the FIELD statement defines the physical parent concatenated key field for the record corresponding to the logical child segment. A physical parent concatenated key is a symbolic pointer to the logical parent. LOGICAL PARENT FIELD and PHYSICAL PARENT FIELD statements (used to define the logical parent concatenated key field) are both required when the named record corresponds to a logical child segment.

Syntax


PHYSical PARENT CONCATenated KEY FIELD name is dl1-field-name
STORED
PHYSically
VIRTually
STARTING POSition is *starting-position*
LENgth is *dl1-field-length*

Parameters

PHYSical PARENT CONCATenated KEY FIELD name is *dl1-field-name*

Specifies the name by which the concatenated key to the physical parent is defined to CA IDMS/DB. Any 1- to 8-character name can be used for the *dl1-field-name*, since this name serves only as a filler.

Make sure that the name selected for *dl1-field-name* is not used to define any other field for the named record.

STORED PHYSically/VIRTually

Specifies whether the physical parent concatenated key is stored with the record corresponding to the logical child segment or is built by the CA IDMS DLI Transparency run-time interface.

PHYSically

Specifies that the physical parent concatenated key is stored with the record equivalent of the logical child segment.

The default is PHYSICALLY. The following considerations apply to the use of this option:

Relationship	What to specify
When the named record corresponding to the logical child segment is participating in a bidirectional physical logical relationship. In such cases, CA IDMS DLI Transparency requires that the logical relationship be made to appear as a bidirectional virtual logical relationship. As described above (in the STORED PHYSICALLY syntax rules under LOGICAL PARENT FIELD Statement (see page 116)), one of the logical child segments must be treated as the real logical child segment, and the other segment must be assigned as the virtual logical child segment. If PHYSICAL or P is entered in this parameter, specify PHYSICAL in the IPSB clause.	STORED PHYSICALLY.

If either of the destination parent concatenated key fields is stored physically, make that field the first physical field in the record.

If both destination parent concatenated key fields are stored physically (see the discussion of STORED PHYSICALLY/VIRTUALLY under "LOGICAL PARENT FIELD Statement"), they must be the first two physical fields in the record. These fields, however, can be preceded by the halfword-length field if the record is a variable-length record. If PHYSICALLY is specified, the STARTING POSITION clause (see below) must be included in the FIELD statement.

VIRTually

Specifies that the physical parent concatenated key is absent from the record corresponding to the logical child segment and is built by the CA IDMS DLI Transparency run-time interface. The use of this option for the segment corresponding to the named record depends on the type of logical relationship defined in the relevant DBDs, as follows:

Relationship	What to specify
For unidirectional logical relationships and bidirectional virtual logical relationships	VIRTUALLY

Relationship	What to specify
For bidirectional physical logical relationships, CA IDMS DLI Transparency requires that one logical child segment be treated as the real logical child segment, and the other logical child segment be treated as the virtual logical child segment. (See the discussion of STORED PHYSICALLY under LOGICAL PARENT FIELD Statement (see page 116).) If the entry in the PARENT parameter of the SEGM statement defining the segment that is being treated as the virtual logical child segment specifies VIRTUAL or V	VIRTUALLY If VIRTUALLY is specified, the STARTING POSITION clause must be omitted.

STARTING POSition is *starting-position*

Specifies the position in the record in which the concatenated key field begins, where the record begins in position 1. Omit this clause if STORED VIRTUALLY is specified for the named field.

Relationship	What to specify
If STORED PHYSICALLY is specified only for the named field (that is, only for the LOGICAL PARENT CONCATENATED KEY field)	Specify START POSITION IS 1. If both the named field and the LOGICAL PARENT CONCATENATED KEY field are stored physically, one of the fields will have a START POSITION of 1. The other field will begin in the next available byte after its complement concatenated key field is stored. When both concatenated key fields are stored physically and the record is a variable-length record, add 2 to the START POSITION to allow for the halfword containing the length of the record.

LENGTH IS *dl1-field-length*

Specifies the length of the concatenated key to the physical parent.

To determine the entry for this clause, first find the DL/I FIELD statements that define the sequence fields of the physical parent segment and of those segments in the physical parent's hierarchical path to the root segment. Add the BYTES clause entries in these FIELD statements. The result is the entry for *dl1-field-length*.

LOGICAL SEQUENCE FIELD statement

This format of the FIELD statement defines the logical sequence field and its attributes. A logical sequence field must be defined for the named record corresponding to a logical child segment whenever the associated virtual logical child has a sequence field. A field defined as a logical sequence field can be used as a search field in an SSA.

Syntax

```
►►─┬───────────────────────────────────────────────┬──────────────►
   └─ LOGical SEQuence FIELD name is dl1-field-name ─┘

 ►─┬────────────────────────────────────────┬─────────────────────►
   └─ STARTING POSition is starting-position ─┘

 ►─┬─────────────────────────────┬────────────────────────────────►◄
   └─ LENgth is dl1-field-length ─┘
```

Parameters

LOGical SEQuence FIELD name is *dl1-field-name*

Identifies the sequence field of the virtual logical child segment. Use the NAME clause entry in the DL/I FIELD statement defining the sequence field for the virtual logical child segment.

STARTING POSition is *starting-position*

Specifies the position in the record in which the sequence field begins. Use the START clause entry in the DL/I FIELD statement defining the sequence field for the virtual logical child segment.

LENgth IS *dl1-field-length*

Specifies the length of the sequence field. Use the BYTES clause entry in the DL/I FIELD statement defining the sequence field for the virtual logical child segment.

Usage

Sample PSB

This sample PSB calls for DBD1, which is shown in the hierarchy diagram in Figure 38. The DBD that defines DBD1 is shown in Figure 39. Figure 40 shows the resulting RECORD SECTION that is developed.

```
PCB     TYPE=DB,DBDNAME=DBD1,PROCOPT=G,KEYLEN=45,PROCSEQ=INDEX1
SENSEG  NAME=SEGRT1,PARENT=0
SENSEG  NAME=SEG3,PARENT=SEGRT1
SENSEG  NAME=SEG4,PARENT=SEG3
SENSEG  NAME=SEG2,PARENT=SEGRT1
PSBGEN  LANG=COBOL,PSBNAME=PSB1
END
```

Figure 37. Sample PSB

Hierarchy Diagram of DBD1

This hierarchy diagram corresponds to database DBD1. SEGRT1, SEG2, SEG3, and SEG4 are specified in the PSB shown in Figure 37 and, therefore, require RECORD statements to define their equivalent records. SEG5 is indicated by broken lines because it is a virtual logical child segment, which is not a real segment.

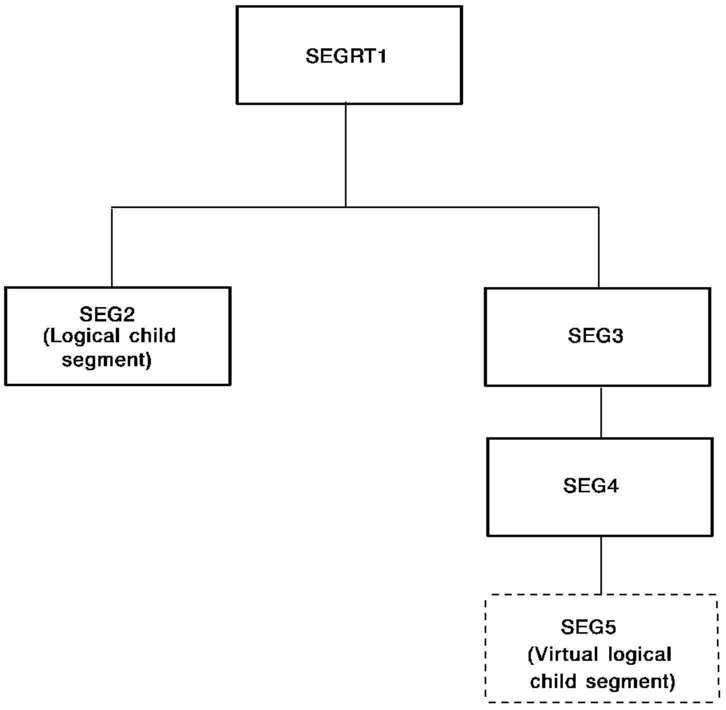

SEGRT1
SEG2 (Logical child segment)
SEG3
SEG4
SEG5 (Virtual logical child segment)

Figure 38. Hierarchy diagram of DBD1

Sample DBDs

DBD1 is the database called for by the PCB shown in Figure 37 DBD2 is the database that contains the logical parent segment of logical child SEG2 and the virtual logical child segment paired with SEG2. Information from the DBDs for both databases is required to complete the RECORD SECTION shown in Figure 40.

```
DBD       NAME=DBD1,ACCESS=HDAM,RMNAME=(DLZHDC30,3,1800,3000)
DATASET   DD1=HDAM1,DEVICE=3350,BLOCK=2048,SCAN=3
SEGM      NAME=SEGRT1,PARENT=0,BYTES=115,POINTER=TWINBWD,RULES=PPV
FIELD     NAME=RT1KEY,SEQ,U,BYTES=11,START=1
FIELD     NAME=FIELD2,BYTES=5,START=1
FIELD     NAME=FIELD3,BYTES=6,START=6
SEGM      NAME=SEG2,PARENT=((SEGRT1),(LPSEGRT,P,DBD2)),
           BYTES=120,POINTER=TWIN6WD),RULES=(PLV)
FIELD     NAME=(KEY2,SEQ,U),BYTES=6,START=1
SEGM      NAME=SEG3,PARENT=SEGRT1,BYTES=10,POINTER=TWIN
FIELD     NAME=(KEY3,SEQ,U),BYTES=3,START=1
FIELD     NAME=FIELD5,BYTES=4,START=4
SEGM      NAME=SEG4,PARENT=SEG3,BYTES=6,POINTER=TWIN
FIELD     NAME=(KEY4,SEQ,U),BYTES=6,START=1
SEGM      NAME=SEG5,PARENT=SEG4,PTR=PAIRED,
           SOURCE=((LCSEG,DATA,DBD3))
FIELD     NAME=(KEY5,SEQ,U),BYTES=21,START=1,TYPE=F
FIELD     NAME=FIELD-5,BYTES=20,START=22,TYPE=F
DBDGEN
FINISH
END


DBD       NAME=DBD2,ACCESS=HDAM
DATASET   DD1=HDAM2,DEVICE=3350,BLOCK=2048,SCAN=3
SEGM      NAME=SEGRT2,PTR=TWINBWD,RULES=LLV
FIELD     NAME=(KEY6,SEQ,U),BYTES=60,START=1
FIELD     NAME=FIELD6,BYTES=15,START=61
FIELD     NAME=FIELD-7,BYTES=75,START=76
LCHILD    NAME=(SEG2,DBD1),PAIR=SEG6,PTR=DBLE
SEGM      NAME=SEG6,PARENT=SEGRT2,PTR=PAIRED
               SOURCE=(SEG2,DATA,DBD1)
FIELD     NAME=(KEY7,SEQ,U),BYTES=21,START=61
FIELD     NAME=FIELD8,BYTES=20,START=22
SEGM      NAME=SEG7 BYTES=200,PARENT=SEG1
FIELD     NAME=(KEY8,SEQ,U)BYTES=99,START=1
FIELD     NAME=FIELD9,BYTES=101,START=100
SEGM      NAME=SEG8,BYTES=100,PARENT=SEG1
FIELD     NAME=(KEY9,SEQ,U),BYTES=15,START=1
FIELD     NAME=FIELD10,BYTES=15,START=51
DBDGEN
FINISH
END
```

Figure 39. Sample DBDs

Sample RECORD SECTION

The information used to define this RECORD SECTION example is based on information in Figure 37 through Figure 39.

SEG5, defined in the first DBD shown in Figure 39, is omitted from this RECORD SECTION example because SEG5 is a virtual logical child segment. However, the fields of a virtual logical child segment are entered under the record corresponding to the logical child when the PSB calls for the associated logical child segment.

Thus, under REC2, which corresponds to SEG2 in DBD1, a logical sequence field and a search field are defined.

These two fields come from the virtual logical child segment (SEG6) located in DBD2, which is defined in the second DBD in Figure 39.

```
RECORD SECTION.
RECORD NAME IS RECRT1 LENGTH IS 115.
  SEQ FIELD NAME IS RT1KEY START POS 1 LENGTH 11.
      FIELD NAME IS FIELD2 START POS 1 LENGTH 5.
      FIELD NAME IS FIELD3 START POS 6 LENGTH 6.
RECORD NAME IS REC2 LENGTH IS 120
LOGICAL PARENT CONCAT KEY FIELD NAME IS FILFLD1
             STORED PHYSICALLY START POS 1 LENGTH 60.
PHYSICAL PARENT CONCAT KEY FIELD NAME IS FILFLD2
            STORED VIRTUALLY LENGTH 11.
  SEQ FIELD NAME IS KEY2 START POS 1 LENGTH 6.
  LOGICAL SEQUENCE FIELD NAME IS KEY7
            START POS 61 LENGTH 21.
      FIELD NAME IS FIELD8 START POS 22 LENGTH 20.
RECORD NAME IS REC3 LENGTH IS 10.
  SEQ FIELD NAME IS KEY3 START POS 1 LENGTH 3.
      FIELD NAME IS FIELD5 START POS 4 LENGTH 4.
RECORD NAME IS REC4 LENGTH IS 6.
  SEQ FIELD NAME IS KEY4 START POS 1 LENGTH 6.
```

Figure 40. Sample RECORD SECTION

INDEX SECTION

The INDEX SECTION provides the information required to relate CA IDMS/DB records and sets to secondary index and HIDAM index structures to be used and/or maintained by the CA IDMS DLI Transparency run-time interface.

The only statement in the INDEX SECTION is the INDEX statement. Each INDEX statement does the following:

- Identifies a HIDAM database or a secondary index
- Identifies the CA IDMS/DB records and sets that correspond to the DL/I segments and segment relationships in that index
- Names the DL/I fields used to build the index
- Names an index suppression exit routine to handle DL/I sparse indexing

Reviewing the INDEX SECTION requires that you identify the HIDAM databases and the secondary indexes that the run-time interface will either use explicitly or maintain implicitly when processing DL/I database requests with this IPSB. An index is used explicitly when one of the following occurs:

- A PCB refers to a DBD with ACCESS=HIDAM.
- A PCB contains a PROCSEQ parameter, which indicates the use of a secondary index to access the root segment.
- One of the SENSEG statements in the PCB has an INDICES parameter.

An index is used implicitly by a PCB when the PCB allows the index target segment or the index source segment to be updated (that is, the PROCOPT parameter has a value of I, R, or D). If in doubt, include the indexes; extra indexes will not affect CA IDMS DLI Transparency processing.

Syntax

```
►►─── INDEX SECTION ── . ──────────────────────────────────────────────►

 ►─┬──────────────────────────────────────┬────────────────────────────►
   └─ INDEX name is indexed-field-name ─┘

 ►─┬───────────────────────────────────────┬───────────────────────────►
   └─ using indexed-set indexed-set-name ─┘

 ►─┬──────────────────────────────────────┬────────────────────────────►
   └─ TARGET record is idms-record-name ─┘

 ►─┬───────────────────────────────────────┬───────────────────────────►
   └─ POINTER record is idms-record-name ─┘

 ►─┬──────────────────────────┬────────────────────────────────────────►
   └─ thru SET idms-set-name ─┘
```

```
►─┬─────────────────────────────────────────────────────────────────►
  └─ SOURCE record is idms-record-name ─┘

►─┬─────────────────────────────────────────────────────────────────►
  └─ CONSTANT is constant ─┘

►─┬─────────────────────────────────────────────────────────────────►
  └─ SEARCH FIELDS are(is) ── (dl1-field-name) ─┘

►─┬─────────────────────────────────────────────────────────────────►
  └─ SUBSEQuence FIELDS are(is) ── (dl1-field-name) ─┘

►─┬─────────────────────────────────────────────────────────────────►
  └─ DUPLicate data FIELDS are(is) ── (dl1-field-name) ─┘

►─┬─────────────────────────────────────────────────────────────────►
  └─ NULL VALue is ─┬─ null-value ─┬─┘
                    ├─ BLANK ──────┤
                    └─ ZERO ───────┘

►─┬─────────────────────────────────────────────────────────────────►◄
  └─ EXIT routine is dl1-exit-routine-name ──── . ──┘
```

Parameters

INDEX SECTION

INDEX SECTION must be the first entry in this section followed by as many INDEX statements as required. The INDEX SECTION sentence must be present even if no INDEX statements are included.

INDEX name is *indexed-field-name*

Names the indexed field by which the index is known. *Indexed-field-name* must be a 1- to 8-character name.

For a HIDAM database, *indexed-field-name* should be the name of the index DBD.

For a secondary index, *indexed-field-name* is the NAME parameter value in the XDFLD statement, which is in the DBD defining the indexed database. For more information about finding the index field names for HIDAM databases and secondary indexes, see DL/I and CA IDMS/DB (see page 21).

using indexed-set *index-set-name*

Identifies the CA IDMS/DB index set through which the DL/I secondary index or HIDAM index structure is implemented.

Index-set-name must be a 1- to 16-character name and must be included in the subschema named in the IPSB SECTION.

TARGET record is *idms-record-name*

Identifies the CA IDMS/DB record that corresponds to the DL/I index target segment in this index.

Idms-record-name must be a 1- to 16-character record name included in the subschema named in the IPSB SECTION and named in the RECORD SECTION.

To identify the target record, first locate in the DBD the SEGM statement that defines the target segment. After identifying the name of the target segment, locate the name of the corresponding record as defined in the CA IDMS/DB subschema in use. For more information about locating the SEGM statement that defines the target segment, see DL/I and CA IDMS/DB (see page 21).

POINTER record is *idms-record-name*

Identifies the CA IDMS/DB record that corresponds to the DL/I index pointer segment in this index.

Idms-record-name must be a 1- to 16-character record name included in the subschema named in the IPSB SECTION and named in the RECORD SECTION.

To identify the pointer record, first locate in the DBD the SEGM statement that defines the pointer segment. After identifying the name of the pointer segment, find the name of the corresponding record as defined in the CA IDMS/DB subschema in use. For more information about locating the SEGM statement that defines the pointer segment, see DL/I and CA IDMS/DB (see page 21).

thru SET *idms-set-name*

Identifies the target pointer set of which the target record is the owner and the pointer record is the member.

Idms-set-name must be a 1- to 16-character name in the subschema and named in the IPSB SECTION. For more information about target pointer sets, see DL/I and CA IDMS/DB (see page 21).

SOURCE record is *idms-record-name*

Identifies the CA IDMS/DB record that corresponds to the DL/I index source segment in this index.

Idms-record-name must be a 1- to 16-character record name in the subschema named in the IPSB SECTION and named in the RECORD SECTION.

The run-time interface uses fields from this record to build the key by which the target record is indexed. To locate the source record, first locate in the DBD the SEGM statement that defines the source segment. After identifying the name of the source segment, locate the name of the corresponding record as defined in the CA IDMS/DB subschema in use. For further information on locating the SEGM statement that defines the source segment, see DL/I and CA IDMS/DB (see page 21).

CONSTANT is *constant*

Specifies a 1-byte field used to identify the index if it is a shared index. The byte value must be enclosed in double quotation marks. (Shared indexes are also known as sparse indexes.)

Constant must be a 1- to 11-character Assembler constant that represents a 1-byte field (typically in character, hexadecimal, or binary format). The following examples illustrate possible values for *constant*:

```
CONSTANT IS "C'A'"
CONSTANT IS "X'02'"
CONSTANT IS "B'00000001"
```

The CONSTANT clauses above specify a 1-byte constant in character, hexadecimal, and binary format. For *constant*, enter the CONST parameter value located in the DL/I XDFLD statement. (This XDFLD statement is found in the DBD defining the indexed database.)

SEARCH FIELDS are (is) *dl1-field-name*

Identifies the DL/I search fields to be taken from the designated source record to build the index key for the target record.

This mandatory clause must name at least one search field and can specify up to five search fields.

Make sure that the search fields identified in this clause are defined in RECORD SECTION FIELD statements.

These FIELD statements are associated with the RECORD statement that names the record designated as the source record. The run-time interface concatenates these fields, uses them to build an index key, and places the key in the designated pointer record. Each *dl1-field-name* must be a 1- to 8-character field name. When entering more than one field name, separate each name by a comma and enclose all the names in parentheses. Enclosing parentheses are optional if only one field name is included.

In a HIDAM database, the sequence field of the root segment is the search field. Therefore, *dl1-field-name* is the NAME parameter value in the DL/I FIELD statement that defines the root segment's sequence field.

For a secondary index, each *dl1-field-name* entry can be found in the SRCH parameter of a XDFLD statement. Each entry corresponds to the name of a DL/I FIELD statement following the SEGM statement that defines the source segment.

SUBSEQuence data FIELDS are (is) *dl1-field-name*

For secondary indexes only, optionally identifies the DL/I subsequence fields to be taken from the designated source record to extend the index key. If specified, you must name at least one subsequence field and can name up to five subsequence fields. The run-time interface concatenates these fields and uses them to extend the index key built from search fields. The subsequence fields identified in this clause must be defined in the RECORD SECTION FIELD statements associated with the RECORD statement that names the record designated as the source record.

Make sure that each of the *dl-field-names* is a 1- to 8-character name. If more than one field name is included, separate the field names with commas and enclose them in parentheses. The enclosing parentheses are optional if only one field name is specified.

To determine an entry for *dl1-field-name*, note the SUBSEQ parameter in the DL/I XDFLD statement. The value in this parameter specifies which of the fields for the index source segment are the subsequence fields. Therefore, although a subsequence field is specified in a SUBSEQ parameter, it is defined in a FIELD statement following the definition of the source segment.

A subsequence field can be a system-related field, in which case its name must begin with /CK or /SX.

DUPLicate data FIELDS are (is) *dl1-field-name*

For secondary indexes only, identifies the DL/I duplicate-data fields to be copied from the designated source record to the pointer record. If specified, this optional clause must name at least one duplicate-data field and can name up to five duplicate-data fields. If named, these fields are concatenated and copied from the source record to the pointer record to permit access to the duplicate data when processing the pointer record independently of the defined index structure. Therefore, data placed in the pointer record has no impact on the key used to create the index. The duplicate-data fields identified in this clause must be defined in RECORD SECTION FIELD statements associated with the RECORD statement that names the record designated as the source record.

Make sure that *dl1-field-name* is a 1- to 8-character name. If more than one field name is included, separate the field names with commas and enclose them in parentheses. The enclosing parentheses are optional if only one field name is included.

To determine an entry for *dl1-field-name*, note the DDATA parameter in the DL/I XDFLD statement. An entry in this parameter specifies which of the fields for the index source segment are the secondary index's duplicate-data fields. Therefore, although a duplicate-data field is specified in a DDATA parameter, it is defined in a FIELD statement following the definition of the source segment.

A duplicate data field can be a system-related field, in which case its name must begin with /CK.

NULL VALue is

Identifies a 1-byte Assembler constant used to suppress the creation of a pointer record during index suppression. The byte value must be enclosed in double quotation marks. Each byte in the named search fields is compared with the NULL VALUE constant.

null-value

Specify a 1- to 8-character Assembler constant that represents a 1-byte field (typically in character, hexadecimal, or binary format). If each byte in the search field equals *null-value*, no pointer record is stored for the associated target record.

BLANK

Specify BLANK for a null value of blanks.

ZERO

Specify ZERO for a null value of binary zeros (low value). The examples below illustrate possible values for *null-value*:

```
NULL VALUE IS "C'A'"
NULL VALUE IS "X'FF'"
NULL VALUE IS "B'00000000'"
```

The NULL VALUE clauses above specify a 1-byte term in character, hexadecimal, and binary format.

To complete this clause, specify the value in the NULLVAL parameter of the XDFLD statement (that is, the XDFLD statement in the DBD that defines the indexed database).

EXIT routine is *dl1-exit-routine-name*

Specifies a user-written exit routine for controlling the creation of selected DL/I secondary index entries.

Dl1-exit-routine-name must match the name specified for the EXTRTN parameter of the XDFLD statement in the indexed DBD. Make sure that you place the named exit routine in an operating-system partitioned data set and that you provide access to it via a CDMSLIB JCL statement.

The CA IDMS DLI Transparency run-time interface loads (invokes) the exit routine when the DL/I application issues an ISRT or REPL call for a CA IDMS/DB record corresponding index source/ to a DL/I index source segment in one or more index relationships.

Usage

Examples of INDEX SECTIONs are shown in the illustrations below along with the resources used to develop these INDEX SECTIONs.

Sample DBDs for a HIDAM Database and Associated Index

The example below shows sample DBDs for a HIDAM database and its associated index database.

As with all HIDAM databases, the target and the source segments are the same segment (the root segment in the HIDAM database), which in this case is SEG1. The pointer segment, SEG2, is the only segment in the index database. Since in a HIDAM database the search field is always the root segment sequence field, the search field in this sample is FIELD1.

```
DBD        NAME=DB1,ACCESS=HIDAM
DATASET    DD1=DBHIDAM,DEVICE=3350,BLOCK=42,RECORD=48,SCAN=1
SEGM       NAME=SEG1,BYTES-31,PTR=H,PARENT=0
FIELD      NAME=(FIELD1,SEQ,U),BYTES=21,START=1
FIELD      NAME=FIELD2,BYTES=10,START=22
LCHILD     NAME=(SEG2,DBINDEX),PTR=INDX
DBDGEN
FINISH
END

DBD        NAME=DBINDEX,ACCESS=INDEX
DATASET    DD1=DBXINDX,DEVICE 3350,BLOCK=44,RECORD=46,SCAN=1
SEGM       NAME=SEG2,BYTES=21
LCHILD     NAME=(SEG1,DB1),INDEX=FIELD1
FIELD      NAME=(FIELD3,SEQ,U),BYTES=21,START=1
DBDGEN
FINISH
END
```

Figure 41. Sample DBDs for a HIDAM database and associated index database

Sample INDEX SECTION Based on a HIDAM Database

The sample below is based on information supplied in the DBDs in Figure 41. The index set IX-SET1 indexes the index pointer segment.

```
INDEX SECTION.
   INDEX NAME IS DBINDEX
   USING INDEXED-SET IX-SET1
   TARGET RECORD IS REC1
   POINTER RECORD IS REC2
   THRU SET REC1-REC2
   SOURCE RECORD IS REC1
   SEARCH FIELD IS FIELD1.
```

Figure 42. Sample INDEX SECTION based on a HIDAM database

DBDs for a Secondary Index and its Associated Index Database

The example below shows the DBDs for a secondary index and its associated index database.

The target segment SEG5 is referenced in the LCHILD statement in the index DBD, while the source segment is referenced in the SEGMENT parameter of the XDFLD statement in the indexed DBD.

The pointer segment, as in all secondary and HIDAM databases, is the only segment in the index database. In this sample, the pointer segment is SEG6.

The search field for the secondary index is referenced in the SRCH parameter of the XDFLD statement in the indexed DBD; the duplicate-data field is referenced in the DDATA parameter of the same XDFLD statement. Both the search field and the DDATA field, however, appear under the SEGM statement defining SEG7.

```
DBD         NAME=DB2,ACCESS=HDAM,RMNAME=(GLDHDC20,5,660,850)
DATASET     DD1=DBHDAM,DEVICE 3350,BLOCK=2048,SCAN=1
SEGM        NAME=SEG5,PARENT=0,BYTES=15
FIELD       NAME=(FIELD5,SEQ,U),BYTES=5,START=1
LCHILD      NAME=(SEG6,DBINDX2),PTR=INDX
XDFLD       NAME=XDFLD1,SEGMENT=SEG7,
             SRCH=FIELD7,DDATA=FIELD6
SEGM        NAME=SEG7,PARENT=SEG5,BYTES=25
FIELD       NAME=(FIELD6,SEQ,U),BYTES=5,START=1
FIELD       NAME=FIELD7,BYTES=20,START=5
DBDGEN
FINISH
END

DBD         NAME=DBINDX2,ACCESS=INDEX
DATASET     DD1=INDX2,DEVICE=3350,BLOCK=23,RECORD=88,SCAN=1
SEGM        NAME=SEG6,PARENT=0,BYTES=25
FIELD       NAME=(FIELD8,SEQ,U),START=1,BYTES=6
LCHILD      NAME=(SEG5,DB2),POINTER=SNGL,INDEX=XDFLD1
DBDGEN
FINISH
END
```

Figure 43. Sample DBDs for a secondary index and associated index database

Sample INDEX SECTION Based on a Secondary Index

This sample is based on information supplied in the DBDs in Figure 43. The CA IDMS/DB records in the sample INDEX SECTION have been assigned the prefix REC, and the segment prefix SEG has been eliminated.

```
 INDEX SECTION.
    INDEX NAME IS XDFLD1
    USING INDEXED-SET IS-SET2
    TARGET RECORD IS REC5
    POINTER RECORD IS REC6
    THRU SET REC5-REC6
    SOURCE RECORD IS REC7
    SEARCH FIELD IS FIELD5
    DUPLICATE DATA FIELD IS FIELD6.
```

Figure 44. Sample INDEX SECTION based on a secondary index

PCB SECTION

The PCB SECTION performs the following:

- Identifies the DL/I segments participating in the hierarchies viewed by a DL/I application
- Names the CA IDMS/DB records that represent these segments
- Defines paths that the DL/I application can follow by naming relevant segments and by defining their relationships to other segments
- Provides a limited amount of DBD information (that is, the DBD names and the access method)

The information included in the PCB SECTION corresponds to the associated DL/I PCBs within a PSB.

The PCB SECTION consists of PCB statements and SEGMENT statements and is formatted as follows:

```
PCB SECTION.
 PCB statement.
 SEGMENT statements.
  ...
PCB statement.
 SEGMENT statements.
  ...
```

The syntax for the PCB statement and the SEGMENT statement are presented separately below.

PCB Statement

A PCB statement is composed of entries from DL/I DBD and PCB statements. The DBD statement is located in the DBD, while the PCB statement is located in the PSB.

Syntax

```
►►─── PCB SECTION ─── . ──────────────────────────────────────────────────►

 ►─┬──────────────────────────────────────────────────────────────────┬───►
   └─ PCB ACCESS METHOD is ──┬─ HDAM ────────────────┬─┘
                             ├─ HIDAM ───────────────┤
                             ├─ HISAM ───────────────┤
                             ├─ INDEX ───────────────┤
                             ├─ SECONDary index ─────┤
                             └─ HSAM ────────────────┘

 ►─┬──────────────────────────────────────────────────────────────────┬───►
   └─ DBDNAME is dbd-name ─┘

 ►─┬──────────────────────────────────────────────────────────────────┬───►
   └─ PROCessing OPTions are(is) ─── dl1-option ─┘

 ►─┬──────────────────────────────────────────────────────────────────┬───►
   └─ POSitioning is ─┬─ SINGLE ◄───┬─┘
                      └─ MULTIPLE ──┘

 ►─┬──────────────────────────────────────────────────────────────────┬──►◄
   └─ PROCessing SEQuence ─┬─ SET is indexed-set-name ──────┬─ . ─┘
                           └─ INDEX is indexed-field-name ──┘
```

Parameters

PCB SECTION.

PCB SECTION must be the first entry in this section, followed by as many PCB statements as required to define all hierarchical views referenced by the DL/I application.

Each PCB statement must be followed by SEGMENT statements to identify the segments that participate in the hierarchical view.

A PCB statement, in conjunction with subsequent SEGMENT statements, represents one DL/I hierarchical view. In addition to presenting the hierarchical view, the SEGMENT statement defines the relationships between the named segment and other DL/I segments, as represented by the corresponding CA IDMS/DB records and set relationships.

PCB ACCESS METHOD is

Specifies the DL/I access method by which the root segment in this database is accessed.

To determine the appropriate entry for this clause:

- First decide if ACCESS METHOD IS SECONDARY INDEX is appropriate (see below). If this entry is inappropriate, locate the DBD specified in the DBDNAME parameter of the DL/I PCB statement.
- Next, locate the DBD statement and use the value in the ACCESS parameter for the PCB ACCESS METHOD clause.

If the PCB specifies a logical database and if the SECONDARY INDEX entry is inappropriate:

- Locate the SEGM statement defining the root segment of the logical database. The SOURCE parameter in this SEGM statement references the source segment (first entry) and the physical database containing the source segment (second entry).
- Locate the DBD that defines this physical database.
- Use the ACCESS parameter value in its DBD statement.

Specific guidelines for the options of the PCB ACCESS METHOD clause follow:

HDAM

Specifies an HDAM access method. If this entry is specified, omit the PROCESSING SEQUENCE clause (see below).

HIDAM

Specifies a HIDAM access method. If this entry is specified, a PROCESSING SEQUENCE INDEX clause (see below) must be included.

HISAM

Specifies a HISAM access method. If this option is specified, a PROCESSING SEQUENCE SET clause (see below) must be included to name the relevant indexed set.

INDEX

Specifies an index access method. If this option is specified, a PROCESSING SEQUENCE SET clause must be included to name the relevant indexed set.

SECONDary index

Specifies a secondary index access method. If this option is specified, a PROCESSING SEQUENCE INDEX clause must be included to name the relevant indexed field.

```
PCB   TYPE=DB,DBDNAME=DBA,PROCOPT=A,KEYLEN=46,PROCSEQ=INDEX1
```

HSAM

Specifies an HSAM access method. If the root segment of the HSAM database is sequenced (that is, a sequenced HSAM), a PROCESSING SEQUENCE SET clause must be included. If the root segment of the HSAM database is unsequenced, omit the PROCESSING SEQUENCE clause. Although the HSAM access method is supported, consider each sequenced HSAM as a HISAM database when defining DL/I databases in the schema (see DL/I and CA IDMS/DB (see page 21)).

DBDNAME IS *dbd-name*

Specifies the name of the DL/I DBD associated with the database view being defined. This name corresponds to the DBD name found in the PCB mask in the application program. Use the name specified in the DBDNAME parameter of the PCB statement.

PROCessing OPTions are (is)

Specifies the DL/I processing options selected for this database view.

Processing options specify whether the DL/I application program can only read the segments in the database view or can both read and update the segments. If updating is allowed, the processing options also specify what kind of updating is permissible. You should include the processing options specified in the associated PROCOPT parameters of the DL/I PCB statement and its associated SENSEG statements.

dl1-option

Acceptable values for *dl1-option* are as follows:

Value	**Explanation**
G	The application program can read the segments.
I	The application program can insert segments.
R	The application program can read and replace segments.
D	The application program can read and delete segments.
A	The application program can read, insert, replace, and delete segments.
P	The application program can issue path calls.

A maximum of four options can be specified for each PCB statement. If more than one processing option is specified, do not separate the option by commas or blanks. See the appropriate DL/I documentation for details on DL/I processing options.

CA IDMS DLI Transparency requires that all processing options be specified for the PCB and does not permit any overrides of global options for an individual segment. To accommodate this requirement, you must enter in the PROCESSING OPTIONS clause the most inclusive DL/I processing option entered in the PSB's PCB statement and its SENSEG statements. If, for example, a PCB statement has PROCOPT=G, and three of the subsequent SENSEG statements have PROCOPT values of I, R, and A, respectively, you would enter the following:

```
PROCESSING OPTION IS A
```

By using the CA IDMS/DB access restrictions, you can restrict the type of access overrides/ for specific records and duplicate DL/I overrides of global processing options.

Note: For more information about CA IDMS/DB access restrictions, see the *CA IDMS Database Administration Guide*.

POSitioning is SINGLE/MULTIPLE

Specifies whether the interface is to maintain single or multiple positioning for this PCB. (Refer to the appropriate DL/I documentation for details on single and multiple positioning.) The default is SINGLE. The entry for the POSITION IS clause is found in the POS parameter of the DL/I PCB statement.

PROCessing SEQuence

The format of the PROCESSING SEQUENCE clause and whether it is included is determined by the access method specified in the PCB ACCESS METHOD clause above.

Omit the PROCESSING SEQUENCE clause if an HDAM access method is specified or if the database is an unsequenced HSAM.

SET is *indexed-set-name*

Include a PROCESSING SEQUENCE clause and specify the SET option if a HISAM or INDEX access method is specified or if the database is an HSAM database with a sequenced root segment. Then, include an *indexed-set-name* parameter.

Indexed-set-name is the 1- to 16-character name of the indexed set having as its member the record equivalent of the root segment of the HISAM, index, or sequenced HSAM database.

INDEX is *indexed-field-name*

Indexed-field-name identifies the index through which the root segment for this hierarchy view can be accessed.

Include the PROCESSING SEQUENCE clause and specify the INDEX option if a HIDAM or SECONDARY INDEX access method is specified. Then, include an *indexed-field-name* parameter.

Indexed-field-name is the 1- to 8-character name of the index. Make sure that the entry in this parameter is defined in the INDEX SECTION.

SEGMENT Statement

Each PCB statement must be followed by a SEGMENT statement for each DL/I segment participating in the DL/I hierarchy. Each SEGMENT statement relates a DL/I segment to a CA IDMS/DB segment to an/ record; the run-time interface uses the CA IDMS/DB record name to represent the segment. The SEGMENT statement also defines relationships between the named segment and other DL/I segments, as represented by the corresponding CA IDMS/DB records and set relationships.

To review SEGMENT statements, you need to locate the relevant PSB and the DBD specified in the DL/I PCB that corresponds to this PCB. If the specified DBD defines a logical DBD, you must also find the accompanying DBDs that define the physical databases and the index databases. Similarly, if the PCB calls for a HIDAM database or for a database with a secondary index, you must locate the DBDs that define the associated index databases. Additionally, you must have a copy of the relevant CA IDMS/DB schema.

There must be a SEGMENT statement for each segment specified in a SENSEG statement in the DL/I PCB. If the processing options A or D have been entered in the PROCESSING OPTION clause of the PCB statement, you may have to enter more SEGMENT statements to identify the dependent segments. The decision on whether additional segments must be identified in a separate SEGMENT statement can be made only after looking at the accompanying DBD and, optionally, a hierarchy diagram of the DBD. Note if any segment defined in the accompanying DBD is a dependent of a segment that is specified in a SENSEG statement. You must define each of these dependent segments with a separate SEGMENT statement if the segment identified in the SEGM statement can be deleted.

SEGMENT statements must appear in hierarchical order. The first SEGMENT statement under a given PCB statement must identify the root segment for the hierarchy. All subsequent SEGMENT statements must be included in the order in which the segments appear in the hierarchy. Similarly, if the DBD specified in the PCB defines a logical database, the SEGMENT statements must appear in the same order as the segments appear in the logical hierarchy. A logical DBD can create inversions from a secondary index and/or from a logical relationship. In such cases, consider the following:

- A secondary index causes an inversion when the target segment of the index is the root segment of the logical database but not of the physical database. In this case, the inversion of the segments is explicitly coded in the logical DBD. By including the SEGMENT statements in the same order as the segments are defined in the logical DBD, you automatically record the inversion. No additional SEGMENT statements are needed other than those coded for the SENSEG statements in the PCB.

- A logical relationship causes an inversion when the PCB references a logical database and you include SEGMENT statements to define the hierarchical path of the destination parent segment in its physical database. In this case, you enter SEGMENT statements to define segments from the destination parent's physical database. The segments being defined in the inversion, however, do not include the dependent segments of the destination parent segment. Even though these dependent segments can require SEGMENT statement entries if they are included in the logical database, they do not participate in the inversion.

The SEGMENT statements must be included to define the segments participating in the logical relationship inversion in reverse hierarchical order. Therefore, the SEGMENT statement defining the destination parent segment must be the first SEGMENT statement in the inversion, and the SEGMENT statement defining the root segment in the destination parent segment's physical database must be the last segment statement in the inversion. Always include the SEGMENT statements for the segments in the logical relationship inversion even if the segments are neither identified in the PCB nor defined in the logical DBD. In such cases, you can assign the segments a status of NOT SENSITIVE. (See the discussion of the USE clause below.)

You must also include SEGMENT statements for each of these segments even if some of them are defined in other SEGMENT statements for the named DBD. If the same segment is included in the logical inversion and is specified in the named DBD, make sure that the segment's name is different each time it is specified in a SEGMENT statement for the named PCB. If the name is the same in both the logical and physical DBD, change the segment name in the SEGMENT statement that is part of the logical inversion. Use instead any name you choose.

Syntax

```
►►── SEGMent name is dl1-segment-name ────────────────────────────►

 ►── RECORD name is idms-record-name ─────────────────────────────►

 ►─┬──────────────────────────────┬───────────────────────────────►
   └─ PARENT is dl1-segment-name ─┘

 ►─┬────────────────────────┬─────────────────────────────────────►
   └─ thru SET idms-set-name ─┘

 ►─┬─┬─ PARENT ◄─┬─┬──────────────┬─┬─────────────────────────────►
   │ └─ CHILD ───┘ └─ is OWNER ─┘ │
   
 ►─┬─┬─ PHYSical ◄─┬─┬──────────────────────────────────────────┬─┬►
     └─ LOGical ───┘ └─ DESTination PARENT is idms-record-name ─┘

 ►─┬────────────────────────┬─────────────────────────────────────►
   └─ thru SET idms-set-name ─┘

 ►─┬──────────────────────────────────────────────────────────────┬►
   └─ INSERT RULES are ─┬─ Logical ◄─┬─,─┬─ Logical ◄─┬─,─┬─ Logical ◄─┬┘
                        ├─ Physical ─┤   ├─ Physical ─┤   ├─ Physical ─┤
                        └─ Virtual ──┘   └─ Virtual ──┘   └─ Virtual ──┘

 ►─┬──────────────────────────────────────────────────────────────┬►
   └─ REPLACE RULES are ─┬─ Logical ◄─┬─,─┬─ Logical ◄─┬─,─┬─ Logical ◄─┬┘
                         ├─ Physical ─┤   ├─ Physical ─┤   ├─ Physical ─┤
                         └─ Virtual ──┘   └─ Virtual ──┘   └─ Virtual ──┘

 ►─┬──────────────────────────────┬───────────────────────────────►─
   └─ ACCESS METHOD is ── HDAM ───┘
 ►─────────────────────────────────────────────────────────────────────────►
   └─┬─ HIDAM PROCessing SEQuence INDEX is indexed-field-name ─┬─┘
     └─ HISAM PROCessing SEQuence SET is indexed-set-name ─────┘

 ►─┬────────────────────────────────────────┬───────────────────────────────►
   └─ SEQuence is by LOGical sequence field ─┘

 ►─┬─────────────────────────────────────────────────────────┬──────────────►◄
   └─ USE is ─┬─ NOT SENsitive ─────────────────────── . ─────┘
              ├─ VIRTual LOGical CHILD (VLC) ─┤
              ├─ KEY ──────────────────────────┤
              ├─ DATA ◄────────────────────────┤
              ├─ KEY,KEY ──────────────────────┤
              ├─ KEY,DATA ─────────────────────┤
              ├─ DATA,KEY ─────────────────────┤
              └─ DATA,DATA ────────────────────┘
```

Parameters

SEGment name is *dl1-segment-name*

Identifies a DL/I segment that participates in the hierarchy being defined. *Dl1-segment-name* must be a 1- to 8-character name. Make sure that each *dl1-segment-name* is used only once within the named DBD.

RECORD name is *idms-record-name*

Identifies the CA IDMS/DB record corresponding to the segment named in the SEGMENT NAME clause. *Idms-record-name* must be the 1- to 16-character name of a record included in the subschema named in the IPSB SECTION and in the RECORD SECTION. For all segments but those in logical databases, the record named here corresponds directly to the segment named in the SEGMENT NAME clause.

When a segment participating in a logical database is named, use the record corresponding to the segment named in the first entry of the SEGM statement's SOURCE parameter. For concatenated segments, which are found only in logical databases, use the record corresponding to the real logical child segment. If the logical child specified in the concatenated segment is the virtual logical child segment, you must first locate the SEGM statement defining the virtual logical child segment to identify the name of the real logical child segment. (For more information, see DL/I and CA IDMS/DB (see page 21).)

PARENT is *dl1-segment-name*

Identifies the parent of the segment named in the SEGMENT NAME clause. The PARENT IS clause must be included for all child segments (that is, segments other than root segments). *Dl1-segment-name* must be a 1-to 8-character segment name and must be the name of a DL/I segment specified in the SEGMENT NAME clause of a preceding SEGMENT statement in the hierarchy.

When entering segments from the database referred to by the PCB, the entry for the PARENT IS clause is the first value in the PARENT parameter of the SEGM statement defining the child segment. Omit this entry, however, when the SEGMENT statement defines a root segment. When entering the SEGMENT statements that define the segments in a logical relationship inversion, the entry for the PARENT IS clause is the name of the SEGMENT defined in the preceding SEGMENT statement. For example, if the destination parent segment is SEGA, and the next segment in the hierarchical path of the destination parent segment in its physical database is SEGB, SEGA is the entry in the SEGMENT statement for SEGB. This entry is correct even though the SEGM statement defining SEGA in its physical DBD shows SEGB as the parent of SEGA.

thru SET *idms-set-name*

Identifies the CA IDMS/DB set that relates the child and parent segments named in the SEGMENT NAME and PARENT IS clauses. The THRU SET clause must be included when the PARENT NAME clause is present. *Idms-set-name* must be a 1- to 16-character set name and must be included in the subschema named in the IPSB SECTION.

PARENT/CHILD is OWNER

Identifies the owner of the parent/child set. The default is PARENT.

PHYSical/LOGical DESTination PARENT is *idms-record-name*

Identifies the CA IDMS/DB record representing the destination parent in a concatenated segment structure. Include this clause only if the record identified in the RECORD NAME clause corresponds to a logical child segment in a logical database.

LOGICAL DESTINATION PARENT must be specified if the logical child named in the DL/I SOURCE parameter of the SEGM statement defining the concatenated segment is the real logical child segment. PHYSICAL DESTINATION PARENT must be specified if the logical child named in the DL/I SOURCE parameter of the SEGM statement defining the concatenated segment is the virtual logical child segment.

Idms-record-name is the CA IDMS/DB record corresponding to the logical child entry in the concatenated segment. This operand must be a 1- to 16-character name, must be included in the subschema named in the IPSB SECTION, and must be named in the RECORD SECTION.

Note: When naming a destination parent, include subsequent SEGMENT statements to define the path back to the root segment in the physical database in which the destination parent participates. All SEGMENT statements included to define this path back to the root segment must specify CHILD IS OWNER for the set that relates the child and parent segments.

thru SET *idms-set-name*

Identifies the CA IDMS/DB set that relates the record equivalents of the logical child segment and destination parent segment. Include this clause only if the DESTINATION PARENT clause is present. *Idms-set-name* name must be a 1- to 16-character set name and must be included in the subschema named in the IPSB SECTION. Always identify the record representing the destination parent segment as the owner of this set.

INSERT RULES are (IS)

Specifies the insert rules to be applied to the physical parent, logical child, and logical parent segments in a concatenated segment structure. This clause can be included only if the SEGMENT NAME clause identifies a concatenated segment. PHYSICAL, LOGICAL, or VIRTUAL must be specified for the physical parent segment, real logical child segment, and logical parent segment, respectively.

Regardless of which parent is used as the destination parent segment, always specify the insert rule for the physical parent first, the insert rule for the logical child second, and the insert rule for the logical parent last. The default insert rule for all three segment types is LOGICAL.

To determine the entry for the INSERT RULES clause, first identify the logical child in the concatenated segment. Trace the logical child back to the DBD that defines its physical database. Locate the SEGM statement that defines the logical child and determine if the segment is the real logical child. If this is the case, locate the RULES parameter in this SEGM statement. The value in the first column of the RULES parameter identifies the insert rule. Using the value in the first column of the RULES parameter, choose the appropriate option for the INSERT RULES clause, as follows:

Physical

Is specified if P is the value in the first column of the RULES parameter.

Logical

Is specified if L is the value in the first column of the RULES parameter.

Virtual

Is specified if V is the value in the first column of the RULES parameter.

If the logical child traced back to the DBD defining the physical database is found to be a virtual logical child segment:

1. Locate the SEGM statement defining the associated real logical child segment (see DL/I and CA IDMS/DB (see page 21)).
2. Then, interpret the RULES parameter in this SEGM statement as described above.
3. Next, locate the RULES parameter in the SEGM statement defining the physical parent segment and in the SEGM statement defining the logical parent segment, and make the appropriate entries in the INSERT RULES clause. Refer to the appropriate DL/I documentation for a description of the insert rules.

REPLACE RULES are (is)

Specifies the replace rules to be applied to the physical parent, logical child, and logical parent segments in a concatenated segment structure. This clause can be included only if the SEGMENT NAME clause names a concatenated segment. Specify the PHYSICAL, LOGICAL, or VIRTUAL option for the physical parent segment, logical child segment, and logical parent segment, respectively.

Regardless of which parent is used as the destination parent segment, always specify the replace rule for the physical parent segment first, for the logical child segment second, and for the logical parent segment last. The default replace rule for all three segment types is LOGICAL.

The last column in the SEGM statement's RULES parameter identifies the replacement rules for the segment. Refer to the appropriate DL/I documentation for a description of the replace rules.

Physical

Is specified if P is the value in the first column of the RULES parameter.

Logical

Is specified if L is the value in the first column of the RULES parameter.

Virtual

Is specified if V is the value in the first column of the RULES parameter.

Note: The run-time interface assumes that the delete rules for the physical parent, logical child, and logical parent are PHYSICAL, VIRTUAL, and LOGICAL, respectively. Refer to the appropriate DL/I documentation for a description of the delete rules.

ACCESS METHOD is

Specifies information about the root segment of the hierarchy that contains the destination parent segment in its physical database, as follows:

- Specifies the root segment's access method in the database containing the destination parent segment.
- Specifies, if applicable, the index through which the root segment is accessed in the database containing the destination parent. This specification is omitted if the root segment is in an HDAM database (see HDAM below).

HDAM

Specifies that the destination parent is in a physical database in which the root segment is accessed through HDAM. Include this option only if the SEGMENT statement identifies the root segment in an HDAM database containing the destination parent.

To determine if this option is appropriate, trace the destination parent, as referenced in the concatenated segment (in the logical database), back to its definition in the physical DBD. If this DBD defines an HDAM database, ACCESS METHOD IS HDAM becomes the appropriate entry when the SEGMENT statement specifying the root segment is entered. If the destination parent in the physical database is the root segment in an HDAM database, include ACCESS METHOD IS HDAM in the SEGMENT statement identifying the concatenated segment and its destination parent. A separate SEGMENT statement is unnecessary for the root segment if the destination parent is the root segment. Omit the PROCESSING SEQUENCE clause with an HDAM specification.

HIDAM PROCessing SEQuence INDEX is *indexed-field-name*

Specifies that the destination parent is in a physical database in which the root segment is accessed through HIDAM. *Indexed-field-name* specifies the field name by which the root segment is indexed. Include the ACCESS METHOD IS HIDAM option only if the SEGMENT statement identifies the root segment in a HIDAM database containing the destination parent.

To determine if this option is appropriate, trace the destination parent, as referenced in the concatenated segment (in the logical database), back to its definition in the physical DBD. If this physical DBD defines a HIDAM database, ACCESS METHOD IS HIDAM becomes the appropriate entry when the SEGMENT statement specifying the root segment is entered. *Indexed-field-name* is the NAME parameter value in the SEQUENCE FIELD statement defining the sequence field of the root segment. This parameter must be a 1- to 8-character name and must be defined in an INDEX statement in the INDEX SECTION.

If the destination parent identified is also the root segment, include this clause in the SEGMENT statement identifying the concatenated segment and its destination parent. A separate SEGMENT statement is unnecessary for the root segment if the destination parent is the root segment.

HISAM PROCessing SEQuence SET is *indexed-set-name*

Specifies that the destination parent is located in a database in which the root segment is accessed through HISAM. *Indexed-set-name* specifies the name of the indexed set that has the record equivalent of the root segment as a member. Include the ACCESS METHOD IS HISAM option only if the SEGMENT statement identifies the root segment in a HISAM database containing the destination parent.

To determine if this option is appropriate, trace the destination parent, as referenced in the concatenated segment (in the logical database), back to its definition in the physical DBD. If this physical DBD defines a HISAM database, ACCESS METHOD IS HISAM becomes the appropriate entry when the SEGMENT statement specifying the root segment is entered. *Indexed-set-name* must be a 1- to 16-character name and must be included in the CA IDMS/DB subschema specified in the IPSB SECTION.

If the destination parent is also the root segment in a HISAM database, include this option in the SEGMENT statement identifying the concatenated segment and its destination parent. A separate SEGMENT statement is unnecessary for the root segment if the destination parent is the root segment.

SEQuence is by LOGical sequence field

Specifies that the logical child, as seen in a concatenated segment, is sequenced under its logical parent segment. If this clause is specified, both of the following conditions must be met:

- The concatenated segment defined in the SEGMENT statement must refer to a physical parent segment as the destination parent.
- The concatenated segment defined in the SEGMENT statement refers to a sequenced virtual logical child segment (that is, the virtual logical child segment in its physical database includes a sequence field). Make sure that the sequence field for the virtual logical child is defined in the RECORD SECTION with a LOGICAL SEQUENCE FIELD statement. (See LOGICAL SEQUENCE FIELD Statement (see page 122).)

USE is

Defines the sensitivity of the segment identified in the SEGMENT NAME clause. The options must be specified as described below. When defining segments other than concatenated segments, select the appropriate option from the first four detailed below. When defining concatenated segments, however, select from the last four options. Note that for concatenated segments, each of the options is a double option, requiring you to specify two options (for example, KEY,KEY or DATA,KEY). For all other segments, one entry must be specified (for example, KEY or DATA).

NOT SENsitive

The named segment is required for CA IDMS DLI Transparency processing but is not to be viewed by the DL/I application. When this option is specified, CA IDMS DLI Transparency allows the segment's corresponding record to be deleted when a DL/I application program calls for deleting any segment in the named segment's hierarchical path. For example, assume that the DL/I application program calls for deleting occurrence B1 from SEGB. Also assume that B1 is the parent of C1 and C2 in SEGC. If SEGC is specified in the SEGMENT statement as USE IS NOT SENSITIVE, CA IDMS DLI Transparency responds to the DL/I deletion call by allowing the deletion of the record equivalents of B1, C1, and C2. USE IS NOT SENSITIVE is appropriate for the named segment if the segment's name is missing from the list of SENSEG statements in the PCB.

VIRTual LOGical CHILD (VLC)

The named segment is available to CA IDMS DLI Transparency processing but is not to be viewed by the DL/I application program. When this option is specified, a DL/I call for deleting this segment's parent (or any segment in its hierarchical path) is honored only if there are no occurrences of this segment under its parent.

For example, assume that SEGC is specified in its SEGMENT statement as USE IS VIRTUAL LOGICAL CHILD, and that SEGB is its parent. The DL/I application program calls for deleting occurrence B1 of segment type SEGB. The record corresponding to B1 is deleted only if B1 has no dependent segments of type SEGC. If B1 has a dependent segment of type SEGC, CA IDMS DLI Transparency notifies the DL/I application that the deletion is not being performed. Normal coding of SEGMENT statements does not require USE IS VIRTUAL LOGICAL CHILD; this option is provided for flexibility.

KEY

The DL/I application views only the key of the named segment. Use this option if either of the following conditions exists:

- The named segment is defined in a logical DBD with a SEGM statement that contains a SOURCE parameter value of KEY or K.
- The SENSEG statement identifying the segment in the PCB has a PROCOPT value of K.

DATA

The named segment is to be viewed in its entirety by the DL/I application. Use this default option if either of the following conditions exists:

- The named segment is defined in a logical DBD with a SEGM statement that contains a SOURCE parameter value of DATA or that uses the DL/I default value of DATA for the SOURCE parameter.
- The SENSEG statement identifying the segment in the PCB either has no PROCOPT value or has any PROCOPT value other than K.

KEY,KEY

For concatenated segments only, the DL/I application program views the concatenated segment. This view is only of the keys of the logical child segment and of the destination parent segment. This option is applicable if KEY is specified in both the logical child portion and the destination parent portion of the SOURCE parameter in the SEGM statement defining the concatenated segment.

KEY,DATA

For concatenated segments only, the DL/I application program views the concatenated segment. This view is of the key of the logical child segment and of the entire destination parent segment. This option is applicable if the SOURCE parameter of the SEGM statement defining the concatenated segment contains KEY in the logical child portion and DATA in the destination parent portion.

DATA,KEY

For concatenated segments only, The DL/I application program views the concatenated segment. This view is of the entire logical child segment and of only the key of the destination parent segment. This option is applicable if the SOURCE parameter of the SEGM statement defining the concatenated segment contains DATA in the logical child portion and KEY in the destination parent portion.

DATA,DATA

For concatenated segments only the DL/I application program views the concatenated segment. This view is of the entire logical child segment and of the entire destination parent segment. This option is applicable if the SOURCE parameter of the SEGM statement defining the concatenated segment contains DATA in both the logical child portion and the destination parent portion.

Usage

An example of a PCB SECTION is shown in the illustrations below, along with the resources that are required to develop this PCB SECTION. The PCB in this PSB calls for a logical database. This logical database and its associated physical databases are diagrammed in the hierarchies shown in Figure 46.

Hierarchy diagrams are often helpful aids in determining which segments are to be specified in the PCB SECTION. To complete a PCB SECTION, however, you must have the applicable DBDs. In this example,

- The applicable DBDs are shown in Figure 46 and Figure 48.
- Figure 47 shows the DBD that defines the logical database
- Figure 48 shows the two DBDs that define the associated physical databases
- Figure 49 shows the data structure diagram for the corresponding CA IDMS/DB database
- The information in Figure 45 through Figure 49 is used to define the sample PCB SECTION shown in Figure 50.

Sample PSB

Figure 45 below shows a sample PSB. Although a PSB can have several PCBs, the PSB shown in this illustration has only one PCB.

```
PCB        TYPE=DB,DBNAME=LOGDB,PROCOPT=G,POS=SINGLE,KEYLEN=12
SENSEG     NAME=LSEGA,PARENT=0,PROCOPT=A
SENSEG     NAME=LSEGB,PARENT=LSEGA,PROCOPT=A
SENSEG     NAME=SEG3,PARENT=LSEGB,PROCOPT=A
SENSEG     NAME=SEG4,PARENT=LSEGB
SENSEG     NAME=SEG8,PARENT=LSEGB
PSBGEN     LANG=COBOL,PSBNAME=PSB1
END
```

Figure 45. Sample PSB

Hierarchies of Sample Databases

These hierarchies correspond to the DBDs in Figure 47 and 4-17. Although SEG7 is not used directly by the application program, it can be affected if SEG6 is deleted.

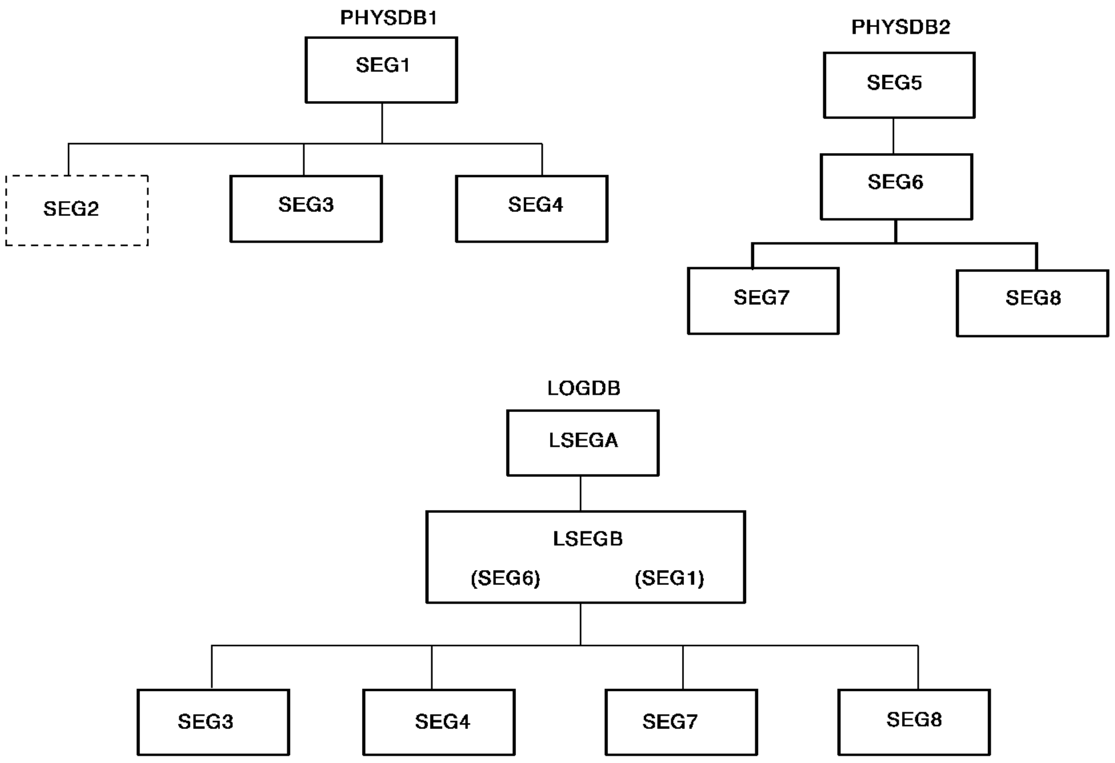


Figure 46. Hierarchies of sample databases

Sample DBD for a Logical Database

LSEGB is the concatenated segment in this example. The SEGM statement for the concatenated segment indicates that SEG6 in PHYSDB2 is the logical child and SEG1 in PHYSDB1 is the destination parent. The DBDs shown in Figure 48 indicate that SEG6 is the real logical child.

```
DBD         NAME=LOGDB,ACCESS=LOGICAL
DATASET     LOGICAL
SEGM        NAME=LSEGA,SOURCE=((SEG5,PHSDB2))
SEGM        NAME=LSEGB,PARENT=LSEGA,
                SOURCE=((SEG6,DATA,PHYSDB2),(SEG1,DATA,PHYSDB1))
SEGM        NAME=SEG3,PARENT=LSEGB,((SEG3,PHYSDB1))
SEGM        NAME=SEG4,PARENT=LSEGB,SOURCE=((SEG4,PHYSDB1))
SEGM        NAME=SEG7,SOURCE=((SEG7,PHYSDB2)),PARENT=LSEGB
SEGM        NAME=SEG8,SOURCE=((SEG8,PHYSDB2)),PARENT=LSEGB
DBDGEN
FINISH
END
```

Figure 47. Sample DBD for a logical database

Sample DBDs for Two Physical Databases

According to these DBDs, SEG2 in PHYSDB1 is the virtual logical child segment, and SEG6 in PHYSDB2 is the real logical child segment.

```
DBD         NAME=PHYSDB1,ACCESS=HDAM
DATASET     DD1=HDAM1,DEVICE=3350,BLOCK=2048,SCAN=3
SEGM        NAME=SEG1,PTR=TWINBWD,RULES=LLV
FIELD       NAME=(FIELD1,SEQ,U),BYTES=60,START=1
FIELD       NAME=FIELD2,BYTES=15,START=61
FIELD       NAME=FIELD3,BYTES=75,START=76
LCHILD      NAME=(SEG6,PHYSDB2),PAIR=SEG2,PTR=DBLE
SEGM        NAME=SEG2,PARENT=SEG1,PTR=PAIRED
                 SOURCE=(SEG6,DATA,PHYSDB2)
FIELD       NAME=(FIELD4,SEQ,U),BYTES=21,START=1
FIELD       NAME=FIELD5,BYTES=20,START=22
SEGM        NAME=SEG3,BYTES=200,PARENT=SEG1
FIELD       NAME=(FIELD6,SEQ,U),BYTES=99,START=1
FIELD       NAME=FIELD7,BYTES=101,START=100
SEGM        NAME=SEG4,BYTES=100,PARENT=SEG1
FIELD       NAME=(FIELD8,SEQ,U),BYTES=15,START=1
FIELD       NAME=FIELD9,BYTES=15,START=51
DBDGEN
FINISH
END


DBD         NAME=PHYSDB2,ACCESS=HDAM,RMNAME=DLZHDC20,7,700,250
DATASET     DDI=HDAM2,DEVICE=3350,BLOCK=2048,SCAN=3
SEGM        NAME=SEG5,BYTES=31,PTR=TWINBWD,RULES=(VLV)
FIELD       NAME=(FIELD9,SEQ,U),BYTES=21,START,TYPE=P
FIELD       NAME=FIELD10,BYTES=10,START=22
SEGM        NAME=SEG6,
                 PARENT=(((SEG5,DBLE),(SEG1,P,PHYSDB1)),
                 BYTES=80,PTR=(LPARNT,TWINBWD),RULES=VVV
FIELD       NAME=(FIELD11,SEQ,U),START=1,BYTES=60
FIELD       NAME=FIELD12,BYTES=20,START=61
SEGM        NAME=SEG7,BYTES=20,,PTR=T
                 PARENT=((SEG6,SNGL))
FIELD       NAME=FIELD13,BYTES=9,START=1
FIELD       NAME=FIELD14,BYTES=11,START=10
SEGM        NAME=SEG8,BYTES=75,PTR=T
                 PARENT=(SEG6,SNGL)
FIELD       NAME=FIELD16,BYTES=50,START=1
FIELD       NAME=FIELD17,BYTES=25,START=51
DBDGEN
FINISH
END
```

Figure 48. Sample DBDs for two physical databases

Sample CA IDMS/DB Data Structure Diagram

The data structure diagram shown in this illustration depicts the CA IDMS/DB schema for the database corresponding to the DBDs shown in Figure 48.

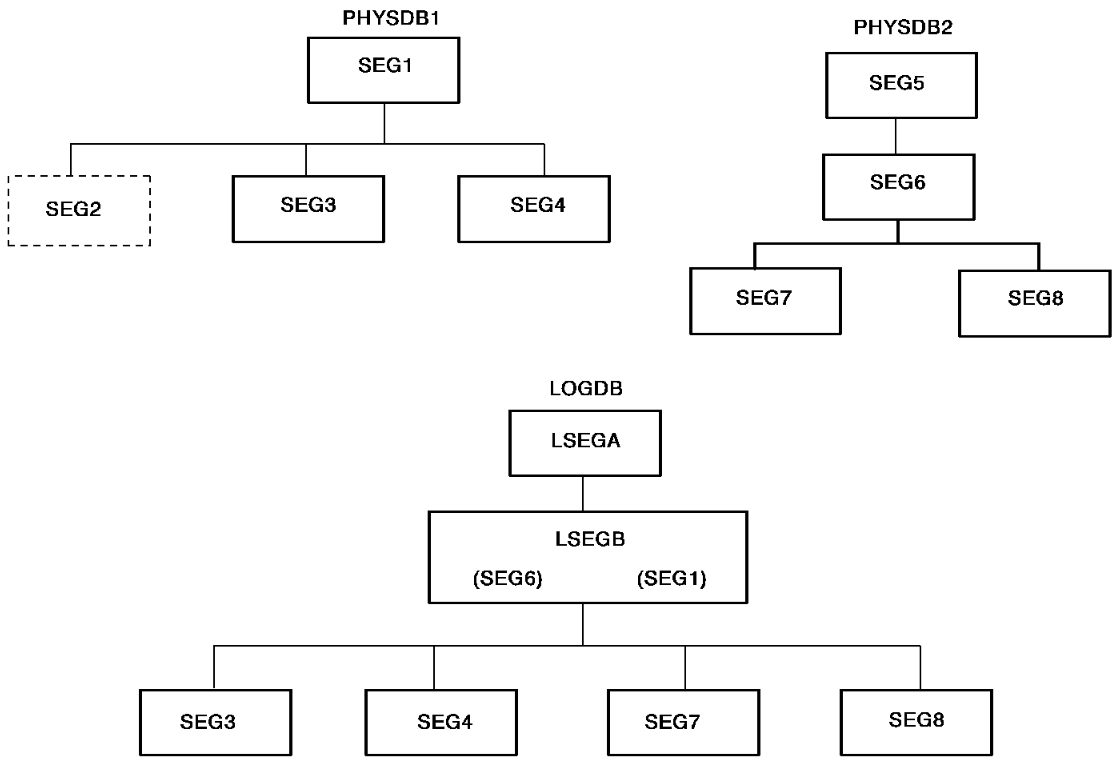


Figure 49. Sample CA IDMS/DB data structure diagram

Sample PCB Section

Figure 45 through Figure 49 are the sources for this sample PCB SECTION.

```
PCB SECTION.
   PCB ACCESS METHOD IS HDAM
     DBDNAME IS LOGDB
     PROCESSING OPTIONS ARE A
     POSITIONING IS SINGLE.
   SEGMENT NAME IS LSEGA RECORD NAME IS REC5
   SEGMENT NAME IS LSEGB RECORD NAME IS REC6
     PARENT IS LSEGA THRU SET REC5-REC6
     LOGICAL DESTINATION PARENT IS REC1
     THRU SET REC1-REC6
     INSERT RULES ARE VIRTUAL,VIRTUAL,LOGICAL
     REPLACE RULES ARE VIRTUAL,VIRTUAL,VIRTUAL
     ACCESS METHOD IS HDAM
     USE IS DATA,DATA.
     SEGMENT NAME IS SEG3 RECORD NAME IS REC3
       PARENT IS LSEGB THRU SET REC1-REC3
       USE IS DATA.
     SEGMENT NAME IS SEG4 RECORD NAME IS REC4
       PARENT IS LSEGB THRU SET REC1-REC4
       USE IS DATA.
    SEGMENT NAME IS SEG7 RECORD NAME IS REC7
       PARENT IS LSEGB THRU SET RC6-REC7
       USE IS NOT SENSITIVE.
     SEGMENT NAME IS SEG8 RECORD NAME IS REC8
       PARENT IS LSEGB THRU SET REC6-REC8
       PARENT IS OWNER USE IS DATA.
```

Figure 50. Sample PCB SECTION

Executing the IPSB Compiler

To execute the IPSB compiler and assemble and link edit the output, use the JCL shown in CA IDMS DLI Transparency JCL (see page 257). The compiler requires as input the IPSB source statements that you have produced via the CA IDMS DLI Transparency Syntax Generator.

Chapter 5: CA IDMS DLI Transparency Run-Time Environment

This section contains the following topics:

About This Chapter

CA IDMS DLI Transparency can run under z/OS or z/VSE in either a batch or CICS environment. CA IDMS supports z/OS V1R10 as well as z/OS 1.1 and above. However, we will always refer to z/OS in this document.

This chapter describes:

- The DL/I run-time environment and the CA IDMS DLI Transparency run-time environment
- The modifications you must make to your system generation parameters for central version (CV) execution in either a batch or CICS environment
- The steps required to run CA IDMS DLI Transparency in either a local mode or CV batch environment
- The steps required to run CA IDMS DLI Transparency in a CICS environment
- Testing the DL/I application in the run-time environment

Note that batch jobs are run in either local mode or under the central version. A CICS environment always operates with the central version.

DL/I and CA IDMS DLI Transparency Run-Time Environments

The DL/I Run-Time Environment

In the DL/I run-time environment:

- A DL/I application issues a call against a DL/I database.
- DL/I controls the program's access to the database by using program specification blocks (PSBs) and Database Definitions (DBDs) that are stored in a run-time library.
- Each PSB contains program communication blocks (PCBs), which define the program's database views.
- After servicing the call, DL/I returns the status information and requested data to the program by way of the appropriate PCB. Note that in a CICS environment, DL/I also uses a user interface block (UIB) to communicate with the program.

Native DL/I Batch and CICS Environments

The diagram below shows the basic DL/I environments for both batch and CICS.

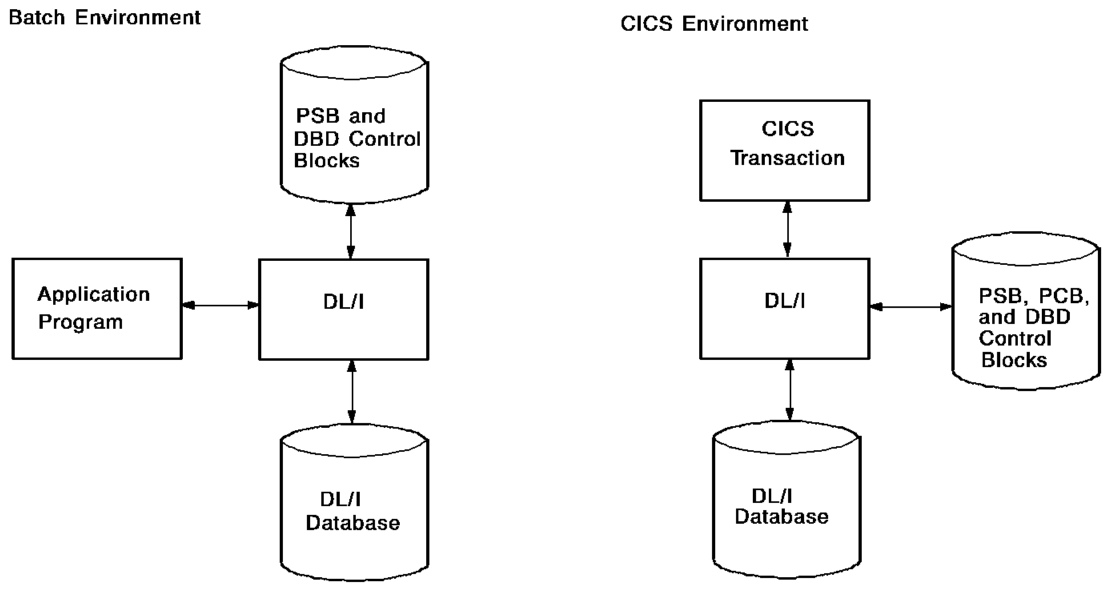


Figure 51. Native DL/I batch and CICS environments

The CA IDMS DLI Transparency Environment

In CA IDMS DLI Transparency, the DL/I database is replaced by a CA IDMS/DB database. DL/I itself is replaced by CA IDMS DLI Transparency and CA IDMS/DB. The DL/I applications remain unchanged.

When a DL/I application issues a call that is addressed to the CA IDMS/DB database that contains the converted DL/I data:

- CA IDMS DLI Transparency converts the call to the corresponding CA IDMS/DB DML call and passes it to CA IDMS/DB, which accesses the database.
- CA IDMS/DB returns the status information and data back to CA IDMS DLI Transparency.
- In CA IDMS DLI Transparency, the PSBs, PCBs, and DBDs are replaced by interface program specification blocks (IPSBs) and subschemas.
- Each IPSB serves as a control block that maps the definition and structure of the DL/I database to the CA IDMS/DB database. Like the PCBs, the subschemas define the program's database views.

CA IDMS DLI Transparency Runtime Components

The diagram below shows the basic CA IDMS DLI Transparency run-time components. The remainder of this section describes how to set up the required CA IDMS DLI Transparency run-time environment for both batch and CICS.

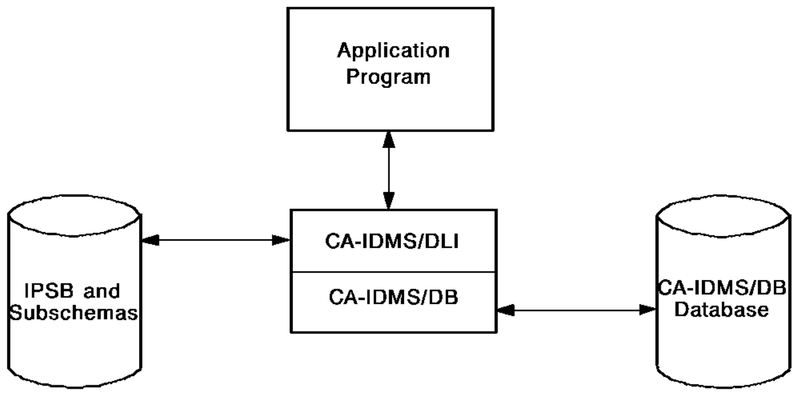


Figure 52. CA IDMS DLI Transparency basic runtime components

Modifying System Generation Parameters

Before generating a CA IDMS system definition, you must set the following system generation parameters on the SYSTEM statement:

- Maximum number of CA IDMS DLI Transparency users
- Program pool size
- Reentrant pool size
- Storage pool size

You must also add certain system generation PROGRAM statements.

Note: For more information about system generation parameters, see the *CA IDMS System Generation Guide*.

These modifications are required only for a batch CV and CICS environment.

Important! Do not make these modifications if you are running CA IDMS DLI Transparency in a local mode environment.

Maximum Number of CA IDMS DLI Transparency Users

On the SYSTEM statement, change the MAXIMUM ERUS parameter to allow for the maximum number of concurrent CA IDMS DLI Transparency users. Note that the MAXIMUM ERUS value must reflect both the number of CA IDMS DLI Transparency users and the maximum number of CA IDMS/DB users, for both batch and CICS.

Program Pool Size

Adjust the program pool size as specified for the PROGRAM POOL parameter on the SYSTEM statement. Use the following formula to calculate the required number of bytes:

```
(ipsb-size * max-num-ipsb) + back-end-size
```

- *Ipsb-size* is the average size for an IPSB. For calculation purposes, you can use 4K as an average IPSB size. If you have large IPSBs, you should adjust the average size accordingly. To determine the actual IPSB sizes, refer to the link maps for the IPSBs.
- *Max-num-ipsb* is the maximum number of nonresident IPSBs.

Reentrant Pool Size

Adjust the reentrant pool size as specified for the REENTRANT POOL parameter on the SYSTEM statement. Use the same formula as for program pool size above.

Storage Pool Size

Adjust the storage pool size as specified for the STORAGE POOL parameter on the SYSTEM statement. Use the following formula to calculate the required number of bytes:

```
4K * maximum erus
```

Maximum erus is the maximum number of concurrent CA IDMS DLI Transparency users. Use the same value calculated for MAXIMUM ERUS above.

Additional PROGRAM Statements

You must include additional system generation PROGRAM statements to define:

- The IDMSDLVC database procedure
- The IDMSDLVD database procedure

You can optionally include PROGRAM statements for IPSBs and subschemas.

IDMSDLVC Database Procedure

Add the following system generation PROGRAM statement to define the IDMSDLVC database procedure. IDMSDLVC is a database procedure for modifying variable-length records.

```
ADD PROGRAM IDMSDLVC
 LANGUAGE IS ASSEMBLER
 REENTRANT
 REUSABLE.
```

IDMSDLVD Database Procedure

Add the following PROGRAM statement to define the IDMSDLVD database procedure. IDMSDLVD is a database procedure for retrieving variable-length records.

```
ADD PROGRAM IDMSDLVD
 LANGUAGE IS ASSEMBLER
 REENTRANT
 REUSABLE.
```

IPSBs and Subschemas

PROGRAM statements can be added for IPSBs and subschemas, but are not required. The PROGRAM statement for an IPSB takes the following form where *ipsb-name* is the name of the IPSB:

```
ADD PROGRAM ipsb-name
  LANGUAGE IS SUBSCHEMA.
```

More information:

CA IDMS DLI Transparency Software Components (see page 241)

Batch Considerations

CA IDMS DLI Transparency batch can run in either a local mode or CV environment.

The diagram below shows the local mode environment.

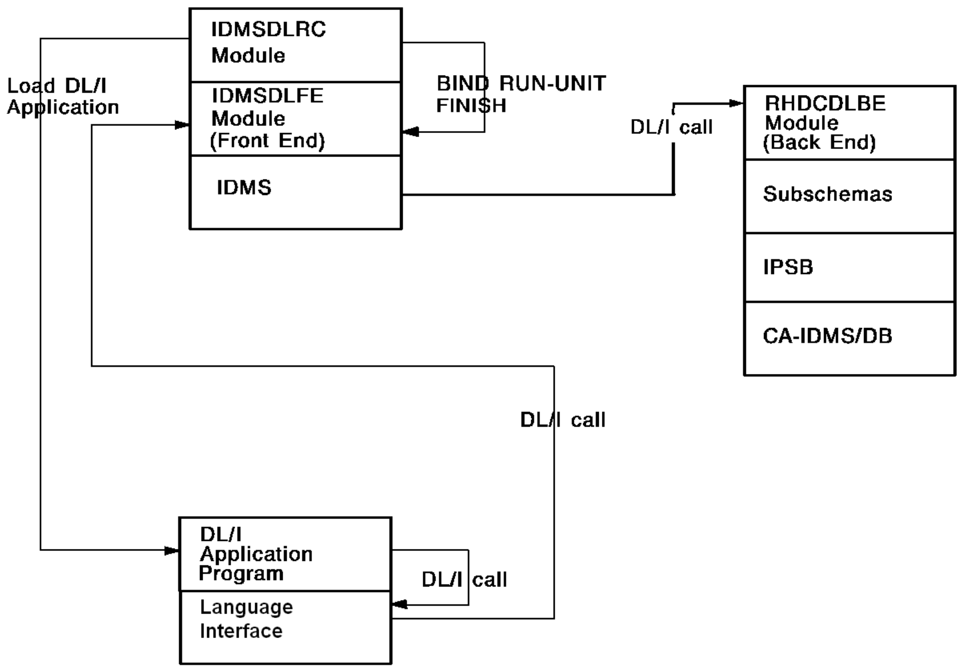


Figure 53. CA IDMS DLI Transparency in a local mode environment

The diagram below shows the batch CV environment.

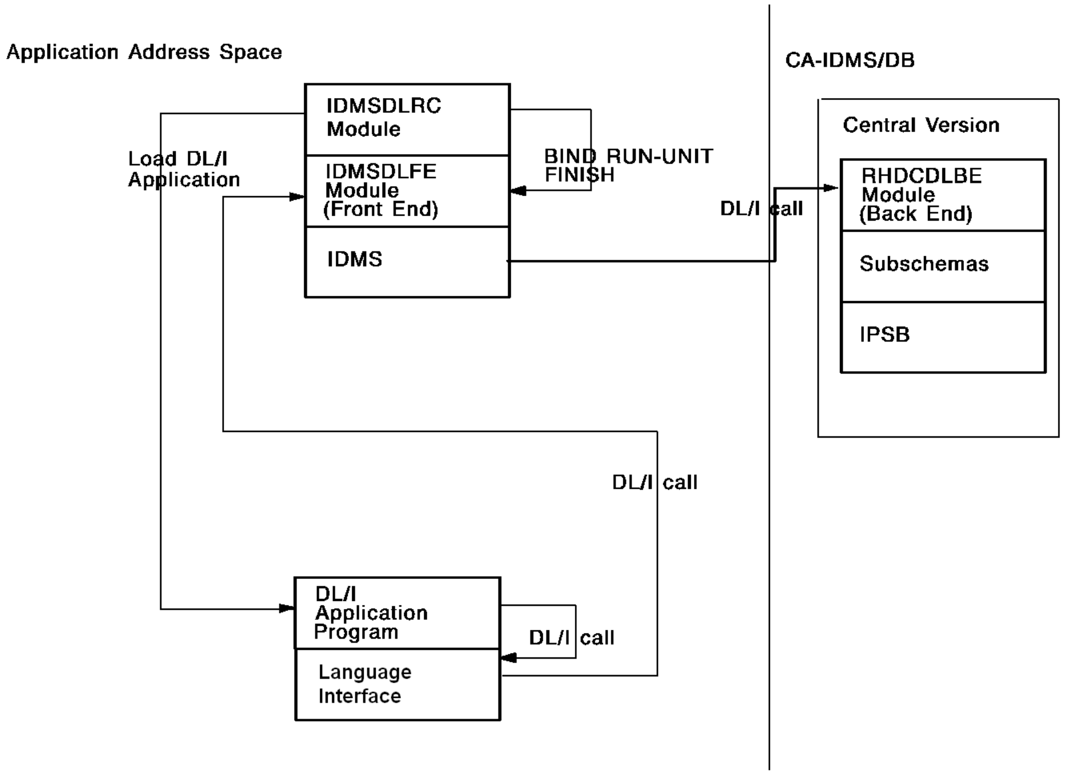


Figure 54. CA IDMS DLI Transparency in a batch CV environment

Steps to Set up Batch Environment

The steps for setting up the CA IDMS DLI Transparency batch environment (local mode or CV) are as follows:

1. Link edit the DL/I applications with the CA IDMS DLI Transparency language interface.
2. Execute the CA IDMS DLI Transparency region controller.

Link Editing Batch DL/I Applications

To prepare your DL/I applications to run in the CA IDMS DLI Transparency batch environment, you must link edit them with the correct CA IDMS DLI Transparency language interface.Module IDMSDLLI should be link edited to call-level DL/I applications, and module IDMSDLHI should be link edited to batch command–level DL/I applications (containing EXEC DLI commands).

To link edit a DL/I application program with the language interface, use the JCL for z/OS and z/VSE provided in Appendix D.

Executing the CA IDMS DLI Transparency Region Controller

The Basic Execute Statement

To run CA IDMS DLI Transparency in a batch environment, you must execute the CA IDMS DLI Transparency region controller (IDMSDLRC). Use the JCL provided in Appendix D.

The basic execute statement (shown for z/OS) is as follows:

```
EXEC  PGM=IDMSDLRC,PARM='DLI,userpgm,ipsb,parms'
```

Parameter List

In the PARM list:

- *Userpgm* is the name of the DL/I batch application.
- *Ipsb* is the name of the IPSB that the application uses when accessing the CA IDMS/DB database.
- *Parms* are additional optional parameters, as follows:
  - **TRACE** -- Traces the call sequence, the I/O areas, the PCBs, and the SSAs. In a central version environment, the trace results are written to the CA IDMS/DB log. In a local mode environment, the trace results are placed in a special dataset called ESCDUMP. If you use TRACE when running in local mode, make sure that you include a DD statement for ESCDUMP. Generally, TRACE is used only for debugging internal problems.
  - **NOSPIE/NOSTAE/NOSTXIT** -- Prevents recursive abends in the case of CA IDMS DLI Transparency abend exit failures.The back-end module (RHDCDLBE) maintains a trace table of activity. If a DL/I application aborts, an abend exit is invoked to format and output the trace information to the CA IDMS/DB log in central version, or to the ESCDUMP DD in local mode. This information is valuable and used by support for diagnostic purposes. Under the central version, if the abend exit also abends, this recursive abend will bring the central version down. These parameters are available in case this situation should ever occur. This parameter should not be routinely specified.

    When running under the central version, only specify one of these options. NOSPIE and NOSTAE are for z/OS only. They turn off SPIES and STAES, respectively. NOSTXIT is for z/VSE only.
  - **DYN** -- Allocates dynamic buffers to the front-end module for use by PL/I programs. In order to use this parameter in a central version environment, you must make sure that the IPSB is available to both the region controller (IDMSDLRC) and the front-end module (IDMSDLFE), as well as to the back-end module (RHDCDLBE).

Modifying Existing DL/I Batch JCL

You can construct the JCL to execute the region controller by modifying the existing JCL for a DL/I batch application. If you do this, make sure you observe the following constraints.

Central Version Environment

- Change the program name to IDMSDLRC.
- Remove any statements that point to DL/I databases. Make sure that you point only to CA IDMS/DB load libraries, and not to IMS or DL/I load libraries.
- Insert a SYSCTL statement.
- Remove all DL/I database definitions.

In a Local Mode Environment

- Change the program name to IDMSDLRC.
- Remove any statements that point to DL/I databases. Make sure that you point only to CA IDMS/DB load libraries and not to IMS or DL/I load libraries.
- Do not include a SYSCTL statement.
- Replace all DL/I file definitions with CA IDMS/DB file definition cards.
- Add journal definition cards. Remember that local mode needs a larger address space than a job accessing the central version. This is because the local address space also includes CA IDMS/DB.

Using Dynamic File Allocation

- There are a number of advantages to utilizing FILE statements in the CA IDMS DMCL to have databases accessed using Dynamic Allocation.

  For more information about utilizing dynamic allocation, refer to the *CA IDMS Database Administration Guide Volume 1*, Chapter 3, Defining Segments, Files, and Areas.

CICS Considerations

You can access CA IDMS/DB from a CICS DL/I application at call level or command-level depending on how your DL/I applications are coded.

If call-level DL/I statements are utilized, DL/I applications can be relinked using the CA IDMS DLI Transparency CICS application interface (IDMSDL1C for z/OS, IDMSDL1V for z/VSE).

If command-level DL/I statements are utilized (EXEC DLI), DL/I applications can be relinked using a CA IDMS DLI Transparency CICS application interface specific to the application programming language and operating system.

For a description of IDMSDL1C, IDMSDL1V, and command-level DL/I application interfaces, see CA IDMS DLI Transparency Software Components (see page 241).

More information:

CA IDMS DLI Transparency Software Components (see page 241)

DL/I CICS Environment

CICS DL/I Environment (z/OS)

As shown in the diagram below, the native DL/I application runs as a CICS transaction.The transaction is linked with the DL/I language interface (DFHDLIAI in z/OS or DLZLI000 in z/VSE) so that it can make DL/I calls. When the transaction starts:

- The language interface loads the address for DFHDLI (or DLZDLI for z/VSE) in the CICS Common Storage Area (CSA)
- DFHDLI (or DLZDLI for z/VSE), in turn, points to the address of the run-time DL/I
- When the transaction issues a DL/I call, the call is passed, via DFHDLIAI and DFHDLI, to DL/I, which services the database request and passes status information and/or data back to the transaction

The diagram below shows the CICS environment for native DL/I under z/OS.

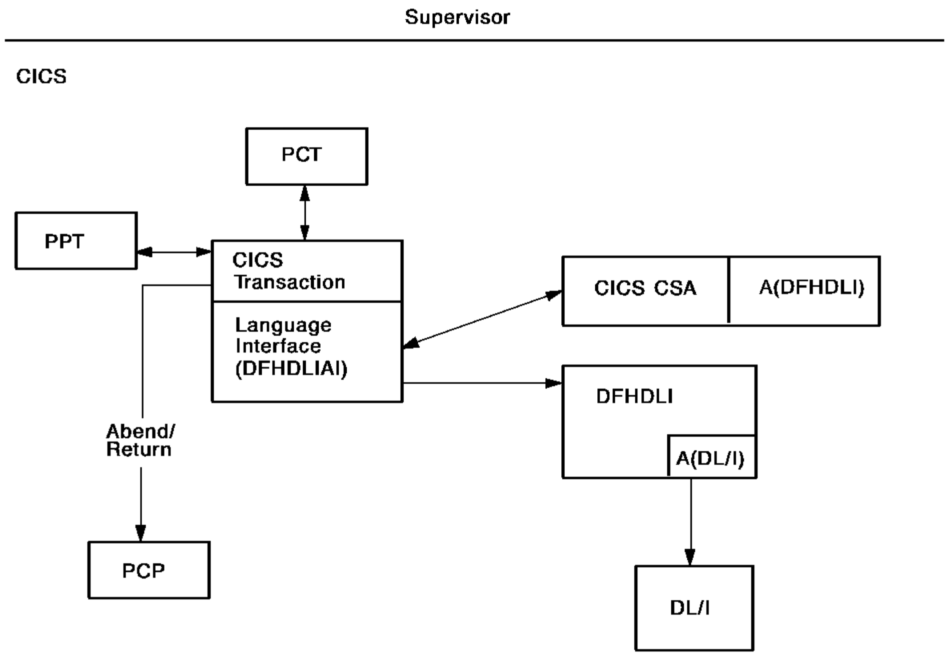


Figure 55. z/OS CICS DL/I environment

CA IDMS DLI Transparency CICS Environment

z/OS CICS Environment (Using Command-Level CICS services)

The diagram below shows the z/OS CICS environment for CA IDMS DLI Transparency using command-level CICS services.

- The CA IDMS DLI Transparency application interface(also referenced as the language interface), intercepts DL/I calls.
- The application interface gets the entry point address of IDMSDLFC (in IDMSINTC) from the CWA and passes control to IDMSDLFC for DL/I parameter processing
- IDMSINTC passes control to the CA IDMS DLI Transparency back-end module, RHDCDLBE, for DL/I call translation into processing against the CA IDMS/DB database

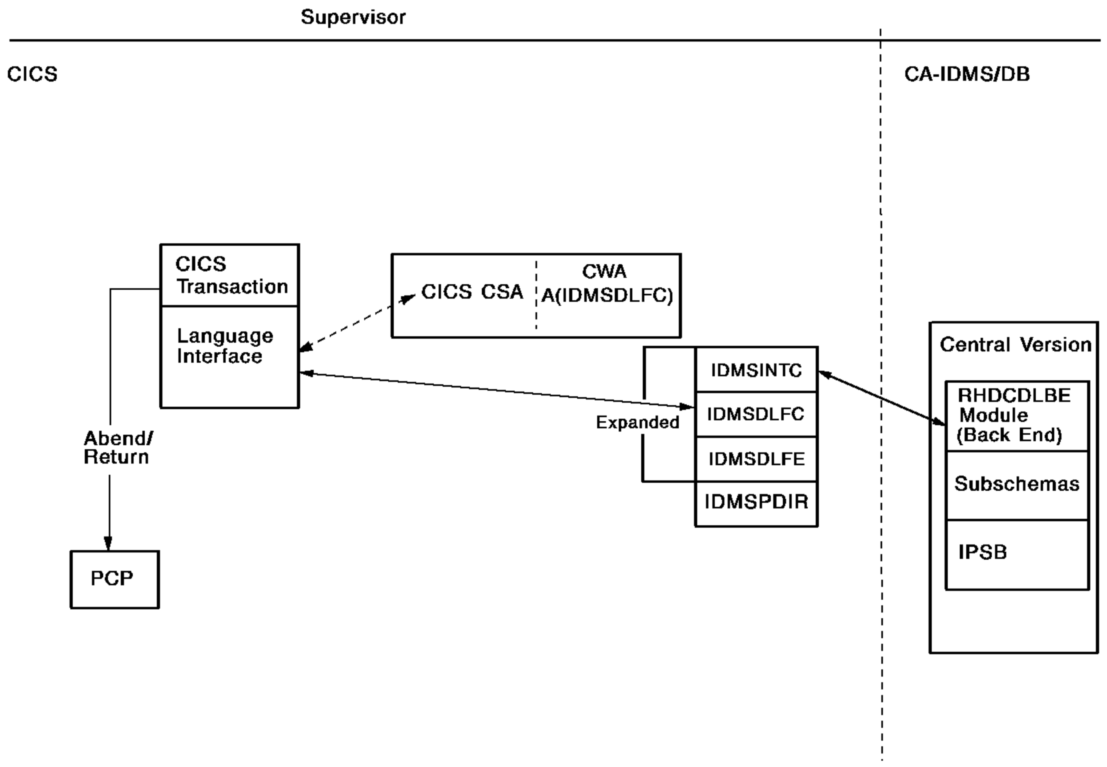


Figure 57. CA IDMS DLI Transparency z/OS CICS environment (command-level only)

Establishing the CA IDMS DLI Transparency CICS Environment

How to Set Up a CA IDMS DLI Transparency CICS Environment

To set up a CICS environment for CA IDMS DLI Transparency, perform the following steps, which are explained in detail in the section directly after this:

1. Assemble the CICSOPTS module with parameter ESCDLI=YES. .
2. If applications utilize EXEC DLI calls, change the HLPI= parameter to YES.
3. Assemble the appropriate language interface module.
4. Link the DL/I application to the language interface module.

Use Appropriate CICS Language Interface

CICS DL/I applications *must be* re-linked with a CA IDMS DLI Transparency application/language interface module. For call-level DL/I usage, IDMSDL1C (z/OS) and IDMSDL1V (Z/VSE) resolve the external references to CBLTDLI, ASMTDLI, or PLITDLI.For EXEC DLI usages, the interface modules are language and operating system specific.For information on assembling language interface modules, see CA IDMS DLI Transparency JCL (see page 257). Note that the interface modules must be assembled with a CWADISP value matching the corresponding CICSOPTS CWADISP value.

Assemble CICSOPTS

Initial Installation

A site-specific CICSOPTS module will be assembled and link edited as part of the installation process. All parameters for CICSOPTS that are required for the DL/I Transparency will be automatically generated by the CAISAG (z/OS) or CAIIJMP (z/VSE) installation utility when you indicate the product to be installed. The site-independent IDMSINTC module will include all modules specifically required to run the DLI/Transparency.

Modifying CICSOPTS

You may need to reassemble CICSOPTS to change some of the original installation options.

z/OS can find the CICSOPTS source in CUSTOM.SRCLIB(CICSOPTS) and the link statements in CUSTOM.LNKLIB(IDMSINTC).

z/VSE clients should edit the CICSOPTS module and relink IDMSINTC. Job control to do this should be taken from the job control that was generated by CAIIJMP for your initial base tape installation.

Note: For more information about the CICSOPTS macro and its parameters, see the *CA IDMS System Operations Guide.*

IDMSINTC is the standard CA IDMS/DB module for running CA IDMS/DB transactions under CICS.CICSOPT parameter ESCDLI=YES specifies that you want to run not only standard CA IDMS/DB, but also CA IDMS DLI Transparency, under CICS. The result of ESCDLI=YES is to expand CICSOPTS, so that it can also serve as the CA IDMS DLI Transparency front end. If applications utilize EXEC DLI statements, HLPI=YES enables support for this DL/I usage. Note that it is possible to link IDMSCINT with a transaction to allow the transaction to make both CA IDMS/DB and DL/I calls.

Prepare to run IDMSINTC in CICS

IDMSINTC itself runs as a transaction under CICS. For a detailed description of how to prepare for this, see the *CA IDMS System Operations Guide*.

Note that IDMSINTC can be executed either automatically at CICS start-up or manually after CICS start-up.

Assemble the language interface

Initial Installation

Language interfaces are automatically generated at installation. The appropriate language interface must be linked with each CICS DL/I application that will access CA IDMS/DB. The value for CWADISP will be set to the same value that was specified in the CICSOPTS assembly by the CAISAG (z/OS) or CAIIJMP (z/VSE) utility.

Modifying Language Interfaces

If you change the CWADISP value used by IDMSINTC, you will need to make the same change to the language interfaces being utilized.

For a description of the language interfaces, see CA IDMS DLI Transparency Software Components (see page 241). For information on assembling CICS language interface modules, see CA IDMS DLI Transparency JCL (see page 257).

z/OS can find the IDMSDL1C source in CUSTOM.SRCLIB and the IDMSSCL1C link statements in the CUSTOM.LNKLIB.

z/VSE clients should make the necessary change to the z/VSE DL/I language interfaces, using the job control that was generated by CAIIJMP as part of your initial base tape installation.

Note: You will need to relink any DL/I application that included the language interfaces.

Testing the DL/I Application

After setting up your CA IDMS DLI Transparency run-time environment, perform the following steps to test a DL/I application:

1. Establish a pilot project using a subset of the DL/I database.
2. Use the CA IDMS DLI Transparency Load Utility (see CA IDMS DLI Transparency Load Utility (see page 171)) to convert database to/ a subset of the DL/I database to a CA IDMS/DB database.
3. Link the DL/I application program with the appropriate language interface.
4. Execute the application against the converted DL/I database.
5. Compare the results of the application's CA IDMS/DB and DL/I executions.

Chapter 6: CA IDMS DLI Transparency Load Utility

This section contains the following topics:

About This Chapter

The CA IDMS DLI Transparency load utility populates an existing CA IDMS/DB database with data unloaded from a DL/I database. This section presents:

- The initial requirements and preparations you need to make
- A description of the process of loading data with the DLI load utility
- Sample code
- A detailed explanation of each step

Using the CA IDMS DLI Transparency Load Utility

The CA IDMS DLI Transparency load utility requires:

- An initialized CA IDMS/DB database and all supporting software necessary to access the database. The supporting software includes usable CA IDMS/DB schema, subschema, and DMCL modules.
- The CA IDMS DLI Transparency run-time interface. The load utility runs under the control of the CA IDMS DLI Transparency region controller. Also, the back-end processor performs special handling of the DL/I data during the load.

- A CA IDMS DLI Transparency interface program specification block (IPSB) load module that accurately describes the DL/I hierarchies involved.
- Unloaded DL/I data in a format compatible with that produced by the IBM DL/I HD unload utility. The load utility accepts data only if it is in this format.
- A working knowledge of CA IDMS/DB, DL/I, and CA IDMS DLI Transparency. Knowledge of CA IDMS DLI Transparency includes familiarity with the CA IDMS DLI Transparency syntax generator, the IPSB compiler, and the run-time interface.

The Database Load Process

The process of loading data with the CA IDMS DLI Transparency load utility can involve up to six steps, as follows:

Step	Process
1. Preload CALC processing	Calculates database pages for CALC records (DL/I root segments). The actual database load (Step 2) can also perform this operation, but it takes longer to do so. Pre-load CALC processing is optional and is provided only to improve loading performance. If preload CALC processing is performed, the resulting data should then be sorted to produce the optimum database loading sequence.
2. Database load	Stores the DL/I data in the prepared CA IDMS/DB database. If the DL/I hierarchies involved do not contain logical relationships, this is the only step required to complete the load process. If logical relationships do exist, you must perform Steps 3 through 6 to resolve the logical child/logical parent relationships. Logical relationships require special treatment for the following reasons: ■ The hierarchical nature of the DL/I data does not ensure that a logical parent will be stored before its logical child. ■ The logical parent concatenated keys are not always present in a logical child input record. If the load utility encounters a logical relationship during the load, it creates logical parent and logical child workfile records and writes them to a separate workfile.
3. Workfile sort/merge	Sorts the workfile produced by Step 2 so that the logical child records appear in proper sequence under their associated logical parent records.

Step	Process
4. Prefix (concatenated key) resolution	Uses the sorted workfile from Step 3 as input. For each logical parent record in the workfile, it generates a correct prefix (concatenated key) for each associated logical child record.
5. Workfile hierarchical sort	Accepts the prefix-resolved workfile from Step 4 as input and sorts the logical child records back into the original hierarchical sequence.
6. Prefix update	Retrieves logical child records already stored in the CA IDMS/DB database (by Step 2 processing). Using the hierarchically sorted workfile from Step 5, it adds the correct prefix (concatenated key) to each logical child database record and connects it to its logical parent record. This step completes the processing for DL/I data that contains logical relationships.

Each of the steps in the database load process is described separately later in this section.

Preparing To Run the Load Utility

Before attempting to execute the load utility, take the following considerations into account.

Preparation of DL/I Data

Unload all DL/I data, including all access methods, by using the DL/I HD unload utility. The CA IDMS DLI Transparency load utility expects the data to be in the format produced by the HD unload utility.

Unload all DL/I HDAM, HIDAM, secondary index, HISAM, and index databases. Index databases have to be unloaded only if the index entries are not created by other record occurrences in the index relationship. See "CA IDMS DLI Transparency Index Maintenance" below.

CA IDMS DLI Transparency Index Maintenance

CA IDMS DLI Transparency creates and updates DL/I index entries for the index relationships defined in the CA IDMS/DB database. In other words, when a source record is inserted, replaced, or deleted in a CA IDMS/DB index relationship, CA IDMS DLI Transparency makes sure that the index relationship's requirements can be met for the insert, replace, or delete call.

You should not input to the load utility any DL/I index entries that would be created by CA IDMS DLI Transparency's index maintenance routines. For example, assume that you have a DL/I index database that is populated whenever a particular root segment is inserted into an associated HDAM database. Since loading of the HDAM database will also populate the index database, there is no need to load the entries in the DL/I index database into the CA IDMS/DB index relationships. CA IDMS DLI Transparency will do this for you.

You can use index suppression exits or null value criteria specifications to support DL/I sparse indexing during the load process. See Index Suppression Exit Support (see page 253) for a discussion of index suppression exits.

Using the CA IDMS DLI Transparency Syntax Generator

It is strongly recommended that you use the syntax generator to produce the source statements for the IPSB load module and the CA IDMS/DB schema, subschema, and DMCL modules. See CA IDMS DLI Transparency Syntax Generator (see page 75) for instructions on creating the special load IPSB and using the GENERATE LOAD IPSB and GENERATE LOAD SCHEMA statements. While you can hand-code the IPSB, use of the syntax generator is more efficient and less time consuming.

Preparation of the IPSB and CA IDMS/DB Load Modules

To produce the CA IDMS/DB modules, input the generated source statements to the appropriate compilers. If you are running the database load in local mode, the subschema and DMCL modules must reside in a library that is accessible by a STEPLIB JCL statement.

To produce the IPSB load module, input the generated IPSB source statements to the IPSB compiler (see IPSB Compiler (see page 93)). Note that the subschema module must be available to compile the IPSB.

The IPSB(s) produced by the syntax generator may not be appropriate for the database load. In this case, you will have to edit the IPSB source to create special load IPSBs (see "Special Load IPSBs" below).

Special Load IPSBs

The IPSB Load Module

The IPSB load module provides the CA IDMS DLI Transparency run-time interface with a description of the DL/I database hierarchies that are referenced in the PCBs (database views) defined for the DL/I application. The IPSB also defines those logical relationships that involve other DL/I databases. Make sure that these logical relationships are correctly defined so that the load utility can find the logical parent records necessary to populate the logical workfile.

Review the IPSB

Specifically, make sure that the IPSB defines:

- Each logically related database
- The physical segments in each database
- The physical path underlying the logical path

Note that logically related databases are defined by way of multiple PCBs within the IPSB. The multiple PCBs are the equivalents of multiple logical DBD descriptions, with full hierarchical definitions, included within the same PSB. Each logically related database is represented by at least one PCB. If the logical relationships do not cross database boundaries, only one PCB that defines the logical relationships is required in the IPSB load module.

In the case of multiple DL/I databases, it is recommended, but not required, that you use one IPSB load module with multiple PCBs.

PROCOPT for Special Load IPSBs

Each PCB included in the IPSB load module must have a PROCOPT of LOAD so the CA IDMS DLI Transparency run-time interface can recognize that the load utility is active. Failure to specify the LOAD PROCOPT can result in load processing errors. If you use the CA IDMS DLI Transparency syntax generator, it will generate this PROCOPT for you automatically. See CA IDMS DLI Transparency Syntax Generator (see page 75) for a description of the GENERATE LOAD IPSB statement.

Availability of the IPSB Load Module

You must make sure that the IPSB load module is available to both the load utility and the CA IDMS DLI Transparency run-time interface. For central version execution, the IPSB load module must be available to the central version and the batch LOAD region.

CA IDMS/DB Schema Requirements

Junction Record Represents Logical Child Segment

For each logical relationship that exists in the DL/I database, the logical child segment must be represented by a CA IDMS/DB junction record. For the junction to exist, there must be two CA IDMS/DB sets. Since there is no assurance that the load process has stored the two set owners (parent records) before it stores the junction (logical child) record, the set with the logical parent as owner must have a set connection option of OPTIONAL MANUAL.

After completing the load process, you can change the set's connection option to MANDATORY AUTOMATIC, if desired.

Bill-of-Materials Relationship Exception

The only exception is the bill-of-materials type of relationship, which requires the junction record to be owned by two different occurrences of the same record type. In this case, the set connection option must remain OPTIONAL MANUAL.

Use the Syntax Generator

It is recommended that you use the CA IDMS DLI Transparency syntax generator to produce a basic load schema with proper set connection options for logical relationships. See CA IDMS DLI Transparency Syntax Generator (see page 75) for a description of the GENERATE LOAD SCHEMA statement.

Multi-Database Logical Relationships

Load Databases Separately

If logical relationships involve more than one database, you must load each database separately (Step 2, under "The Database Load Process", earlier in this section).

Separate Logical Workfiles are Created

Each database load that you perform creates a separate logical workfile. You must make sure that the workfile from each load is available for the workfile sort/merge processing (Step 3). If a required workfile is not available, the prefix resolution processing (Step 4) encounters unresolved logical relationships, and you have to perform the database loads again.

Workfile Space Allocation

Load Utility Generates Separate Workfiles

The load utility generates four separate workfiles:

- The workfile produced by Step 2 (under "The Database Load Process", earlier in this section)
- The sorted workfile produced by Step 3
- The prefix-resolved workfile produced by Step 4
- The hierarchically sorted workfile produced by Step 5

General Considerations for Workfiles

Here are some general considerations for workfiles:

- The workfiles can be allocated to either disk or tape.
- Each workfile is a sequential fixed-block data set with a logical record size of 288 and a block size of 5760.
- If you are using DASD space, you can use the following formula to calculate the number of bytes required for the first workfile produced by Step 2:

  ```
  ((# of logical children) + (# of logical parents)) X 288
  ```
- Be sure to include all potential logical parents as well as all existing logical children. (Refer to "Workfile Usage for HISAM Logical Parents," below.) If you do not know the numbers for logical children and logical parents, you can use the preload CALC processing (Step 1) to get a count of all record occurrences that will appear in the logical workfile.
- Remember that there will be a separate workfile generated for each DL/I database that you load. (See "Multi-Database Logical Relationships," above).
- Space requirements for the second (sorted workfile) are equal to the sum of the space requirements for all the workfiles resulting from the load.
- To calculate the space requirements for the third (prefix-resolved) workfile, use the formula shown above for the first workfile, but specify 0 (zero) for # of logical parents. This workfile contains only the logical child records, but with adjusted prefixes (concatenated keys).
- Space requirements for the fourth (hierarchically sorted) workfile are the same as for the third workfile.

Workfile Usage for HISAM Logical Parents

During the database load, the load utility writes logical parent records that appear in HDAM, HIDAM, and secondary index databases out to the logical workfile. However, the load utility does not write out logical parent records for HISAM databases. Because logical parents from HISAM databases do not appear in the logical workfile, you can reduce its space requirements accordingly. If you use the DASD space requirement formula shown above, you should adjust it so that it does not include any HISAM logical parents.

Preload Sorting

Preload sorting sorts the DL/I input data into database page sequence for more efficient loading. You can preload sort only DL/I data that has been successfully preload CALC processed (Step 1, under "The Database Load Process", earlier in this section).

Diagnostic and Error Messages

The diagnostic and error messages that may be returned by the various steps in the load process are listed in CA IDMS DLI Transparency Messages and Codes (see page 211).

Sample Source Code For Database Load

This section presents sample source code for:

- A DL/I PSB and its associated DBDs
- An IPSB load module
- A CA IDMS/DB schema module

The samples illustrate the process of preparing the necessary modules for use with the CA IDMS DLI Transparency load utility.

The IPSB source code and the CA IDMS/DB source code both derive from the DL/I PSB and DBDs.

The source code examples are also reflected in the sample reports that appear for the various steps in the database load process (described later in the section).

Sample DL/I PSB and DBDs

Figure 58 shows the source for two logically related DL/I databases and a PSB. The DBD descriptions define:

- Two HIDAM physical databases (ITEMDBDP and PARTDBDP)
- Two logical databases (ITEMDBDL and PARTDBDL)
- Two index databases (ITEMDBDI and PARTDBDI)

The PSB references the two logical databases.

The physical databases have root segments named ITEM and PART, respectively. They are logically related using the DETAIL segment.

Source statements for DL/I PSB and DBDs:

```
DL/I ITEM DATABASE PHYSICAL DBD EXAMPLE
        DBD      NAME=ITEMDBDP,ACCESS=HIDAM
        DATASET  DD1=ITEMDB,DEVICE=FBA
        SEGM     NAME=ITEM,PARENT=0,BYTES=150,POINTER=TB,RULES=PPV
        LCHILD   NAME=(ITEMNDX,ITEMDBDI),POINTER=INDX
        FIELD    NAME=(ITEMNO,SEQ),BYTES=7,START=1
        SEGM     NAME=DETAIL,PARENT=((ITEM,SNGL),                    X
              (PART,VIRTUAL,PARTDBDP)),BYTES=150,                    X
              RULES=PVV,POINTER=(TB,LTB)
        FIELD    NAME=(ITMDTAIL,SEQ),BYTES=3,START=19
        DBDGEN
        FINISH
        END

DL/I PARTS DATABASE PHYSICAL DBD EXAMPLE

        DBD      NAME=PARTDBDP,ACCESS=HIDAM
        DATASET  DD1=PARTDB,DEVICE=FBA
        SEGM     NAME=PART,PARENT=0,BYTES=150,POINTER=TB,RULES=PPV
        LCHILD   NAME=(PARTNDX,PARTDBDI),POINTER=INDX
        LCHILD   NAME=(DETAIL,ITEMDBDP),POINTER=SNGL,PAIR=DETAILV
        FIELD    NAME=(PARTNO,SEQ),BYTES=18,START=1
        SEGM     NAME=DETAILV,PARENT=PART,POINTER=PAIRED,            X
              SOURCE=((DETAIL,,ITEMDBDP))
        FIELD    NAME=(ITMDTAIL,SEQ,M),BYTES=3,START=8
        DBDGEN
        FINISH
        END

DL/I ITEM INDEX DBD EXAMPLE

        DBD      NAME=ITEMDBDI,ACCESS=INDEX
        DATASET  DD1=ITEMIX,DEVICE=FBA
        SEGM     NAME=ITEMNDX,PARENT=0,BYTES=7
        LCHILD   NAME=(ITEM,ITEMDBDP),POINTER=SNGL,INDEX=(ITEMNO)
        FIELD    NAME=(ITEMNO,SEQ,U),BYTES=7,START=1
        DBDGEN
        FINISH
        END
```

Figure 58 (Part 1 of 2). Source statements for DL/I PSB and DBDs

```
DL/I PARTS INDEX DBD EXAMPLE

        DBD      NAME=PARTDBDI,ACCESS=INDEX
        DATASET  DD1=PARTIX,DEVICE=FBA
        SEGM     NAME=PARTNDX,PARENT=0,BYTES=18
        LCHILD   NAME=(PART,PARTDBDP),POINTER=SNGL,INDEX=(PARTNO)
        FIELD    NAME=(PARTNO,SEQ,U),BYTES=18,START=1
        DBDGEN
        FINISH
        END

DL/I ITEM DATABASE LOGICAL DBD EXAMPLE

        DBD      NAME=ITEMDBDL,ACCESS=LOGICAL
        DATASET  LOGICAL
        SEGM     NAME=ITEM,PARENT=0,SOURCE=((ITEM,,ITEMDBDP))
        SEGM     NAME=DETAIL,PARENT=ITEM,                           X
              SOURCE=((DETAIL,,ITEMDBDP),(PART,,PARTDBDP))
        DBDGEN
        FINISH
        END

DL/I PARTS DATABASE LOGICAL DBD EXAMPLE

        DBD      NAME=PARTDBDL,ACCESS=LOGICAL
        DATASET  LOGICAL
        SEGM     NAME=PART,PARENT=0,SOURCE=((PART,,PARTDBDP))
        SEGM     NAME=DETAIL,PARENT=PART,                           X
              SOURCE=((DETAIL,,ITEMDBDP),(ITEM,,ITEMDBDP))
        DBDGEN
        FINISH
        END
```

Figure 58 (Part 2 of 2). Source statements for DL/I PSB and DBDs

Sample Load IPSB

GENERATE IPSB Statement

Assuming the DL/I PSB and DBD definitions in illustration 6-1 are assembled and are available to the syntax generator using a CDMSLIB JCL statement, the following GENERATE statement will produce the appropriate IPSB source code for use with the load process:

```
GENERATE LOAD IPSB FOR PSB ITEMPART USING SUBSCHEMA PRODSUBS.
```

This statement instructs the generator to produce an IPSB named ITEMPART and submit it to validity checking for use with the load process.

Figure 59 shows the IPSB source code as it might be produced by the syntax generator using the DL/I DBD and PSB definitions in Figure 58.

Considerations

Here are some points to note about the IPSB source code:

- Each PCB in the DL/I PSB appears as a separate entry in the IPSB's PCB section
- Each PCB entry describes both the physical segments involved and how the physical segments extend into the logical path
- Once the IPSB source is compiled, the resulting IPSB load module can be used to load both of the logically related databases (ITEMDBDL and PARTDBDL). A PCB for each of these DBDs must be included in the IPSB for a successful load.

GENERATE IPSB Statement LOAD Parameter

The use of the LOAD parameter in the GENERATE statement ensures that the resulting IPSB includes all of the DL/I dependencies necessary for a successful load. If a PCB does not identify the physical segment that corresponds to a referenced logical parent, the syntax generator will return an error message and not create the IPSB source.

An example

For example, if the PARTDBDL PCB were not present in the assembled DL/I ITEMPART PSB, the syntax generator would return an error message stating that it could not find the DBD for the logical parent in any PCB. In this case, the missing DBD would be the physical DBD, as referenced by the logical DBDs, ITEMDBDL and PARTDBDL. Providing the PCB for the logical DBD PARTDBDL would satisfy the load process requirements and produce the correct IPSB source.

Generated IPSB source statements:

```
DL/I ITEM DATABASE LOGICAL DBD EXAMPLE


DBD      NAME=ITEMDBDL,ACCESS=LOGICAL
DATASET  LOGICAL
SEGM     NAME=ITEM,PARENT=0,SOURCE=((ITEM,,ITEMDBDP))
SEGM     NAME=DETAIL,PARENT=ITEM,
                     SOURCE=((DETAIL,,ITEMDBD),(PART,,PARTDBDP))
DBDGEN
FINISH
END


DL/I PARTS DATABASE LOGICAL DBD EXAMPLE


DBD      NAME=PARTDBDL,ACCESS=LOGICAL
DATASET  LOGICAL
SEGM     NAME=PART,PARENT=0,SOURCE=((PART,,PARTDBDP))
SEGM     NAME=DETAIL,PARENT=PART,
                     SOURCE=((DETAIL,,ITEMDBDP),(ITEM,,ITEMDBDP))
DBDGEN
FINISH
END


DL/I PSB DESCRIBING ITEM AND PARTS LOGICAL DBDS


PCB      TYPE=DB,DBDNAME=ITEMDBDL,PROCOPT=AP,KEYLEN=28,POS=S
SENSEG   NAME=ITEM,PARENT=0
SENSEG   NAME=DETAIL,PARENT=ITEM
PCB      TYPE=DB,DBDNAME=PARTDBDL,PROCOPT=AP,
              KEYLEN=28,POS=S
SENSEG   NAME=PART,PARENT=0
SENSEG   NAME=DETAIL,PARENT=PART
PSBGEN   LANG=ASM,PSBNAME=ITEMPART
END


IPSB SECTION.
         IPSB NAME IS ITEMPART
         OF SUBSCHEMA PRODSUBS
         LANGUAGE IS ASM.
```

Figure 59 (Part 1 of 4). Generated IPSB source statements

```
AREA SECTION.

         AREA NAME IS ITEMDBDP-REGION
              USAGE-MODE IS EXCLUSIVE UPDATE.
         AREA NAME IS PARTDBDP-REGION
              USAGE-MODE IS EXCLUSIVE UPDATE.

RECORD SECTION.

         RECORD NAME IS ITEM
              LENGTH IS  150.
         SEQUENCE
         FIELD NAME IS ITEMNO
              START POS  1
              LENGTH IS  7.

         RECORD NAME IS DETAIL
              LENGTH IS  157.
         SEQUENCE
         FIELD NAME IS ITMDTAIL
              START POS  26
              LENGTH IS  3.
         LOGICAL PARENT CONCAT KEY
         FIELD NAME IS DETALPCK
              START POS  1
              LENGTH IS  18.
         PHYSICAL PARENT CONCAT KEY
         FIELD NAME IS DETAPPCK
               START POS  19
               LENGTH IS  7.

         RECORD NAME IS PART
               LENGTH IS  150
         SEQUENCE
         FIELD NAME IS PARTNO
               START POS  1
               LENGTH IS  18.

         RECORD NAME IS ITEMNDX
               LENGTH IS  7.
         SEQUENCE
         FIELD NAME IS ITEMNO
               START POS  1
               LENGTH IS  7.
```

Figure 59 (Part 2 of 4). Generated IPSB source statements

```
         RECORD NAME IS PARTNDX
               LENGTH IS  18.
         SEQUENCE
         FIELD NAME IS PARTNO
               START POS  1
               LENGTH IS  18.

INDEX SECTION

         INDEX NAME IS ITEMDBDT
               USING INDEXED-SET IX-ITEMNDX
               TARGET RECORD IS ITEM
               POINTER RECORD IS ITEMNDX
                     THRU SET ITEM-ITEMNDX
               SOURCE RECORD IS ITEM
               SEARCH FIELD (ITEMNO).

         INDEX NAME IS PARTDBDI
               USING INDEXED-SET IX-PARTNDX
               TARGET RECORD IS PART
               POINTER RECORD IS PARTNDX
                     THRU SET PART-PARTNDX
               SOURCE RECORD IS PART
               SEARCH FIELD (PARTNO).

PCB SECTION.

         PCB ACCESS METHOD IS HIDAM
               DBDNAME IS ITEMDBDL
               PROCESSING OPTIONS LOAD
               PROC SEQ INDEX IS ITEMDBDI.

         SEGM NAME IS ITEM
         RECORD  NAME IS ITEM.

         SEGM NAME IS DETAIL
         RECORD  NAME IS DETAIL
               PARENT IS ITEM
                     THRU SET ITEM-DETAIL
               LOGICAL DEST PARENT IS PART
                     THRU SET PART-DETAIL
               INSERT RULES P,P,P
               REPLACE RULES V,V,V
               ACCESS METHOD IS HIDAM
               PROC SEQ INDEX IS PARTDBDI.
```

Figure 59 (Part 3 of 4). Generated IPSB source statements

```
PCB ACCESS METHOD IS HIDAM
      DBDMANE IS PARTDBDL
      PROCESSING OPTIONS LOAD
      PROC SEQ INDEX IS PARTDBDI.

SEGM NAME IS PART
RECORD   NAME IS PART.

SEGM NAME IS DETAIL
RECORD   NAME IS DETAIL
     PARENT  IS PART
           THRU SET PART-DETAIL
     PHYSICAL DEST PARENT IS ITEM
           THRU SET ITEM-DETAIL
     INSERT RULES P,L,P
     REPLACE RULES V,L,V
     ACCESS METHOD IS HIDAM
     PROC SEQ INDEX IS ITEMDBDI
     SEQUENCE BY LOGICAL SEQ FIELD.
```

Figure 59 (Part 4 of 4). Generated IPSB source statements

Sample CA IDMS/DB Schema Module

GENERATE Schema Statement

Just as with the IPSB source code, you can use the syntax generator to make sure that you have a CA IDMS/DB schema module that will support a successful database load. The GENERATE statement in this case takes the following form:

```
GENERATE LOAD SCHEMA NAME IS LOADSCHM FOR DBD ITEMDBDP, PARTDBDP.
```

Note that physical DBD names are all that you need to produce the schema source.

Figure 60 shows the schema source code as it might be produced by the syntax generator using the DL/I physical DBD definitions in Figure 58.

Considerations

Here are some general considerations about the schema source code produced by the syntax generator:

- The generated schema has OPTIONAL MANUAL set connection options for each logical child/logical parent set.
- Generated schema source by itself is not sufficient for a database load. It must be edited to include site-specific standards, optimized database page ranges, and so on.
- If you already have a suitable CA IDMS/DB schema, you can modify this schema without having to create a new load schema. Specifically, you must make sure that the logical child/logical parent set description has OPTIONAL MANUAL connection options. These options are required only during the load process and can be changed to MANDATORY AUTOMATIC after the load. The only exception is in the case of bill-of-materials relationships.
- In this type of relationship the logical parent and physical parent of the child record are different occurrences of the same record type. Bill of materials sets must have OPTIONAL MANUAL connection options.
- The page ranges specified for the CA IDMS/DB database in the schema/subschema must be consistent throughout the load process. A change in the page ranges will invalidate the database pages calculated by the preload CALC processing (Step 1). In this case, you will have to repeat both the CALC and load steps so the logical workfile produced by the load can give correct results for the prefix update step.

Generated Schema source statements:

```
SIGNON
         USAGE MODE IS UPDATE .
SET OPTIONS FOR SESSION
         INPUT     1 THRU     72.


ADD SCHEMA NAME IS LOADSCHM VERSION     1
         MEMO DATE IS 12/22/86
         ASSIGN RECORD IDS FROM     101
         PUBLIC ACCESS IS ALLOWED FOR ALL.


ADD AREA NAME IS PARTDBDP-REGION.


ADD AREA NAME IS ITEMDBDP-REGION.


ADD RECORD NAME IS PART
         RECORD ID IS AUTO
         LOCATION MODE IS CALC
         USING PARTNO
         DUPLICATES ARE NOT ALLOWED
         WITHIN AREA PARTDBDP-REGION.
02       PARTNO            PIC X (18).
02       FILLER            PIC X (132).


ADD RECORD NAME IS ITEM
         RECORD ID IS AUTO
         LOCATION MODE IS CALC
         USING ITEMNO
         DUPLICATES ARE NOT ALLOWED
         WITHIN AREA ITEMDBDP-REGION.
02       ITEMNO            PIC X (7).
02       FILLER            PIC X (143).


ADD RECORD NAME IS DETAIL
         RECORD ID IS AUTO
         LOCATION MODE IS VIA ITEM-DETAIL
         WITHIN AREA ITEMDBDP-REGION.
02       FILLER            PIC X (25).
02       ITMDTAIL          PIC X (3).
02       FILLER            PIC X (129).
```

Figure 60 (Part 1 of 4). Generated Schema source statements

```
ADD RECORD NAME IS PARTNDX
         RECORD ID IS AUTO
         LOCATION MODE IS VIA PART-PARTNDX
         WITHIN AREA PARTDBDP-REGION.
02       PARTNO           PIC X (18).


ADD RECORD NAME IS ITEMNDX
         RECORD ID IS AUTO
         LOCATION MODE IS VIA ITEM-ITEMNDX
         WITHIN AREA ITEMDBDP-REGION.
02       ITEMNO           PIC X (7).


ADD SET NAME IS ITEM-DETAIL
         ORDER IS SORTED
         MODE IS CHAIN LINKED TO PRIOR
         OWNER IS ITEM
              NEXT DBKEY POSITION IS AUTO
              PRIOR DBKEY POSITION IS AUTO
         MEMBER IS DETAIL
              NEXT DBKEY POSITION IS AUTO
              PRIOR DBKEY POSITION IS AUTO
         LINKED TO OWNER
              OWNER DBKEY POSITION IS AUTO
         MANDATORY AUTOMATIC
         ASCENDING KEY IS ITMDTAIL
         DUPLICATES ARE NOT ALLOWED.


ADD SET NAME IS PART-DETAIL
         ORDER IS SORTED
         MODE IS CHAIN LINKED TO PRIOR
         OWNER IS PART
              NEXT DBKEY POSITION IS AUTO
              PRIOR DBKEY POSITION IS AUTO
         MEMBER IS DETAIL
              NEXT DBKEY POSITION IS AUTO
              PRIOR DBKEY POSITION IS AUTO
         LINKED TO OWNER
              OWNER DBKEY POSITION IS AUTO
         OPTIONAL MANUAL
         ASCENDING KEY IS ITMDTAIL
         DUPLICATES ARE LAST.
```

Figure 60 (Part 2 of 4). Generated Schema source statements

```
ADD SET NAME IS PART-PARTNDX
         ORDER IS SORTED
         MODE IS CHAIN LINKED TO PRIOR
         OWNER IS PART
              NEXT DBKEY POSITION IS AUTO
              PRIOR DBKEY POSITION IS AUTO
         MEMBER IS PARTNDX
              NEXT DBKEY POSITION IS AUTO
         LINKED TO OWNER
              OWNER DBKEY POSITION IS AUTO
         MANDATORY AUTOMATIC
         ASCENDING KEY IS PARTNO
         DUPLICATES ARE NOT ALLOWED.


ADD SET NAME IS ITEM-ITEMNDX
         ORDER IS SORTED
         MODE IS CHAIN LINKED TO PRIOR
         OWNER IS ITEM
              NEXT DBKEY POSITION IS AUTO
              PRIOR DBKEY POSITION IS AUTO
         MEMBER IS ITEMNDX
              NEXT DBKEY POSITION IS AUTO
         LINKED TO OWNER
              OWNER DBKEY POSITION IS AUTO
         MANDATORY AUTOMATIC
         ASCENDING KEY IS ITEMNO
         DUPLICATES ARE NOT ALLOWED.


ADD SET NAME IS IX-PARTNDX
         ORDER IS SORTED
         MODE IS INDEX
              BLOCK CONTAINS  50 KEYS
         OWNER IS SYSTEM
         MEMBER IS PARTNDX
              INDEX DBKEY POSITION IS AUTO
         MANDATORY AUTOMATIC
         ASCENDING KEY IS PARTNO
         DUPLICATES ARE NOT ALLOWED.
```

Figure 60 (Part 3 of 4). Generated Schema source statements

```
ADD SET NAME IS IX-ITEMNDX
         ORDER IS SORTED
         MODE IS INDEX
              BLOCK CONTAINS  50 KEYS
         OWNER IS SYSTEM
         MEMBER IS ITEMNDX
              INDEX DBKEY POSITION IS AUTO
         MANDORY AUTOMATIC
         ASCENDING KEY IS ITEMNO
         DUPLICATES ARE NOT ALLOWED.


VALIDATE.
SIGNOFF.
```

Figure 60 (Part 4 of 4). Generated Schema source statements

Step 1: Preload CALC Processing

Preload CALC processing is an optional step that precedes the actual database load. Its intent is to improve the performance of load processing and is especially recommended if:

- There are large amounts of DL/I data.
- There are logical relationships in the DL/I database.
- Space requirements need to be determined for the logical workfile(s) that will be generated by the load (Step 2).

Operation

Preload CALC processing performs the following operations:

1. Accessing the IPSB load module
2. Accessing the subschema module named in the IPSB
3. Reading the DL/I input data
4. Generating database page numbers for the DL/I root segments
5. Updating the DL/I data with the database page numbers and writing it out to the DL/I output file
6. Printing a report on the updated DL/I data

Figure 61 shows the operations performed by preload CALC processing.

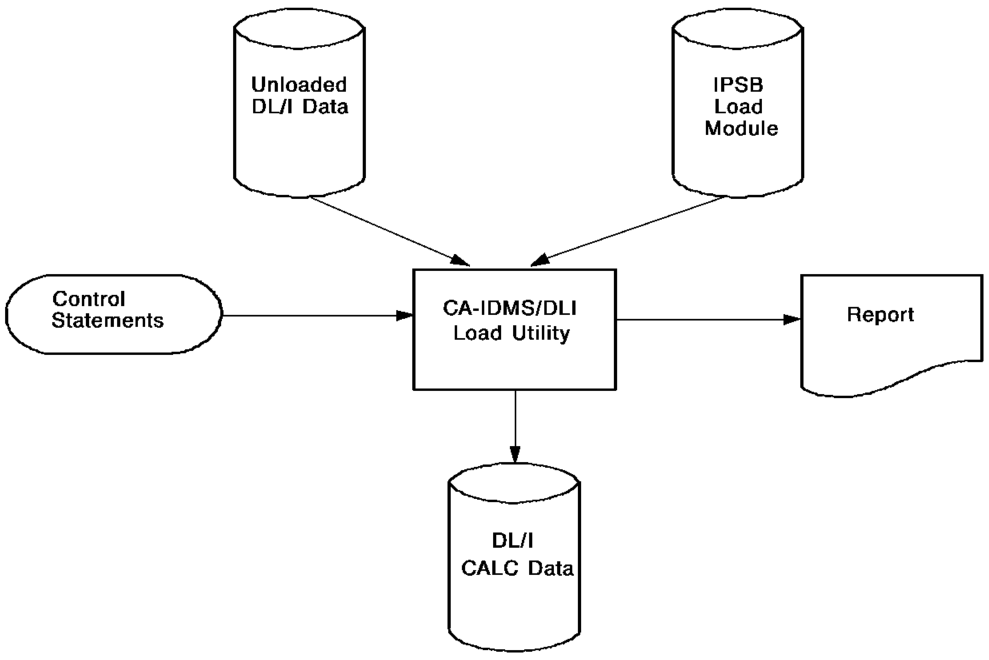


Figure 61. Preload CALC processing

To execute the preload CALC processing step, use the JCL in CA IDMS DLI Transparency JCL (see page 257).

Report

The report produced by the preload CALC processing step lists:

- The DBDNAME for each DL/I database included in the input DL/I data
- The name and level for each DL/I segment, by database
- An indication if a segment is a logical child (LC) or logical parent (LP)
- The number of segment occurrences (records) found, by database
- The number of logical records found, by database

```
*** CA IDMS/DLI TRANSPARENCY DATABASE LOAD

    PROCESS=CALC,IPSB=ITEMPART

    DBDNAME=ITEMDBDL

    SEGMENT COUNT LEVEL RECORD

    ITEM    1086    01  ITEM

LC  DETAIL  3542    02  DETAIL

    TOTAL:  4628  RECORDS READ

            3542  LOGICAL RECORDS
               0  LOGICAL RECORDS WRITTEN

*** CALC PROCESSING COMPLETE
____________________________________________________

*** CA IDMS/DLI TRANSPARENCY DATABASE LOAD

    PROCESS=CALC,IPSB=ITEMPART

    DBDNAME=PARTDBDL

    SEGMENT COUNT LEVEL RECORD

LP  PART     789    01  PART

    TOTAL:   789  RECORDS READ

             789  LOGICAL RECORDS
               0 LOGICAL RECORDS WRITTEN

*** CALC PROCESSING COMPLETE
```

Figure 62. Sample CALC processing report

Preload Sorting (step 1, part 2)

Use Your Own Sort/Merge Utility

To further optimize the CALC-processed data for loading, you can sort it using your own sort/merge facility. As input to the sort/merge facility, supply the DL/I output file produced by the preload CALC processing. The output file will contain the CALC-processed data in sorted form. You can then use the sorted output file as input to the database load (Step 2).

The preload sort is not strictly required, but it should be performed to produce the most effective ordering of the CALC-processed data.

To perform the preload sort, you must use your own sort/merge facility.

What the Preload Sort Does

The preload sort performs the following operations:

1. Accessing the CALC DL/I data produced by the preload CALC processing (Step 1, Part1)
2. Sorting the data so that root segments (CALC records) are in descending database page sequence (the optimum CA IDMS/DB database load order)

To execute the preload sort processing step, use the JCL (Step 1, Part 2) in CA IDMS DLI Transparency JCL (see page 257).

Step 2: Database Load

Using the unloaded DL/I data as input, database load processing invokes the CA IDMS DLI Transparency region controller and populates the CA IDMS/DB database with the unloaded DL/I data. If you have CALC processed and, optionally, sorted the DL/I data, you must input the DL/I file produced as a result of Step 1.

This step completes the database load for DL/I data that does not contain logical relationships. If the DL/I data involves logically related databases, you must continue with Steps 3 through 6.

Operation

Database load processing performs the following operations:

1. Accessing the IPSB load module
2. Reading the DL/I input data
3. Storing all records in the CA IDMS/DB database
4. Extracting all logical child records and writing them out to the logical workfile
5. Extracting all logical parent records and writing them out to the logical workfile
6. Printing a report showing the results of the load

To execute the database load step, use the JCL in CA IDMS DLI Transparency JCL (see page 257).

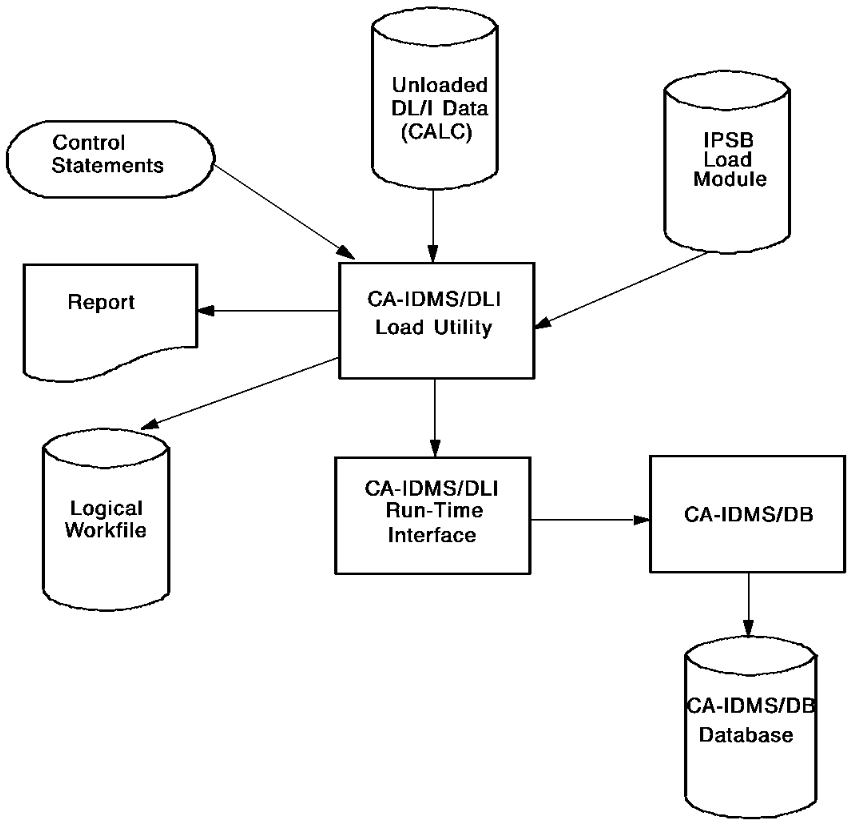


Figure 63. Database load processing

Report

The report produced by the database load step lists:

- The DBDNAME for each DL/I database included in the input DL/I data
- The name and level for each DL/I segment, by database
- An indication if a segment is a logical child (LC) or logical parent (LP)
- The number of segment occurrences (records) loaded, by database
- The number of logical records found, by database
- The number of logical records, by database, written out to the logical workfile

```
*** CA IDMS/DLI TRANSPARENCY DATABASE LOAD

    PROCESS=LOAD

    DBDNAME=ITEMDBDL

    SEGMENT COUNT LEVEL RECORD

    ITEM     1086  01   ITEM

LC  DETAIL   3542  02   DETAIL

    TOTAL:   4628  RECORDS LOADED

             3542  LOGICAL RECORDS
             3542  LOGICAL RECORDS WRITTEN

*** LOAD PROCESSING COMPLETE
------------------------------------------------------


*** CA IDMS/DLI TRANSPARENCY DATABASE LOAD

    PROCESS=LOAD

    DBDNAME=PARTDBDL

    SEGMENT COUNT LEVEL RECORD

LP  PART    789    01   PART

    TOTAL:  789   RECORDS LOADED

            789   LOGICAL RECORDS
            789   LOGICAL RECORDS WRITTEN

*** LOAD PROCESSING COMPLETE
```

Figure 64. Sample database load report

Step 3: Workfile Sort/Merge

The logical workfiles produced by the database load (Step 2) contain the logical child and logical parent records found in the original DL/I data. The workfile sort/merge step sorts the logical child records under their related parents.

If the database load processed multiple DL/I databases, you will have a separate workfile for each database. If this is the case, you must first merge all of the generated workfiles into one workfile. You can then sort this one workfile.

To perform the workfile sort/merge step, you must use your own sort/merge facility.

Operation

The workfile sort/merge performs the following operations:

1. Accessing the workfile(s) resulting from the database load
2. Merging multiple workfiles (from multiple, logically related DL/I databases)
3. Sorting the workfile so that logical child records are sequenced under their logical parents

To execute the workfile sort/merge step, use the JCL in CA IDMS DLI Transparency JCL (see page 257).

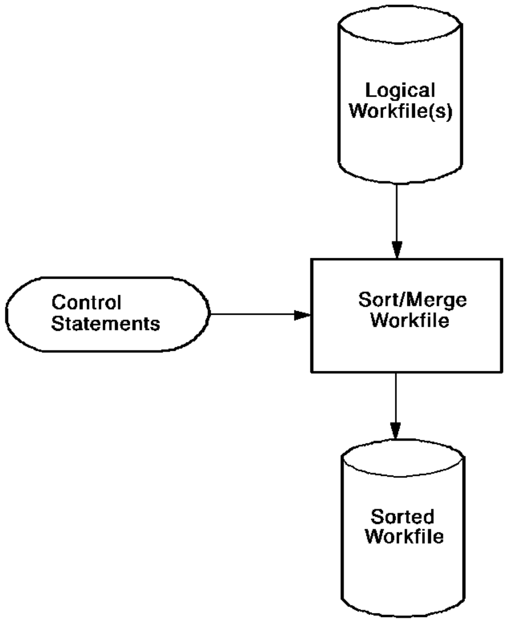


Figure 65. Workfile sort/merge

Step 4: Prefix (Concatenated Key) Resolution

The sorted logical workfile produced by Step 3 contains the logical child and logical parent records from the DL/I logically related databases. The logical child records are sorted correctly under their respective logical parents, but their prefix (nondata) portions do not reflect the parents' concatenated keys. The prefix resolution step updates the logical child records with their parents' concatenated keys so the logical child records can be accessed within their CA IDMS/DB sets.

If the database load processed multiple DL/I databases, you will have a separate workfile for each database. If this is the case, you must first merge all of the generated workfiles into one workfile. You can then sort this one workfile.

Operation

The prefix resolution step performs the following operations:

1. Accessing the IPSB load module
2. Accessing the sorted workfile from Step 3
3. From each logical parent record, generating the correct prefix (concatenated key) for its logical child record
4. Updating the logical child records with the correct prefixes and writing them out to a new workfile
5. Producing a report of the records processed

To execute the prefix resolution step, use the JCL in CA IDMS DLI Transparency JCL (see page 257).

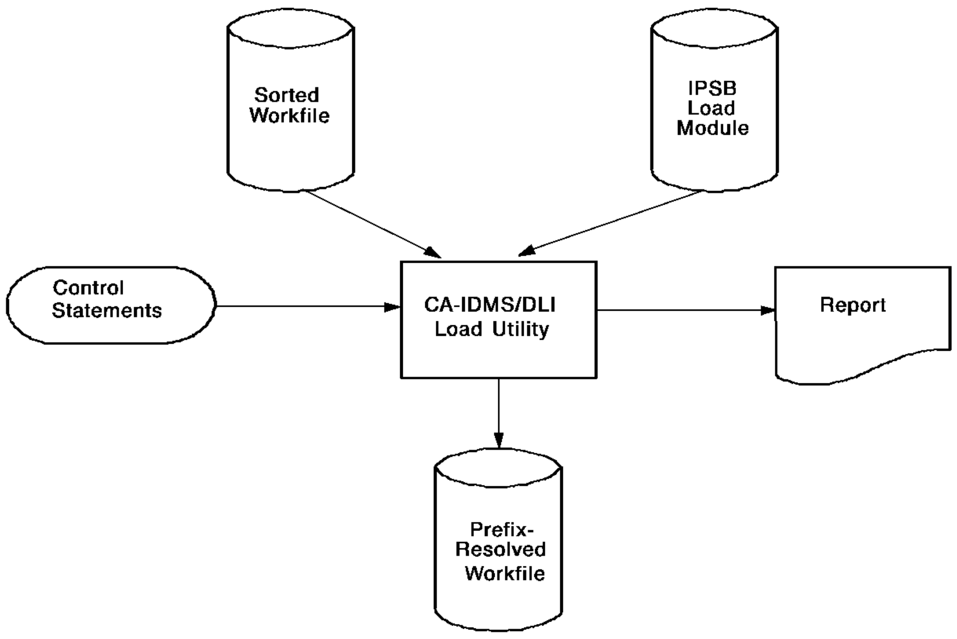


Figure 66. Prefix resolution

Report

The report produced by the prefix resolution step lists:

- The DBDNAME for the DL/I logical child database
- The name and level for each DL/I segment
- An indication if a segment is a logical child (LC) or logical parent (LP)

- The number of logical parent records found
- The number of logical child records found
- The total number of records found in the sorted workfile
- The total number of logical child records updated and written out

```
*** CA IDMS/DLI TRANSPARENCY DATABASE LOAD

    PROCESS=PFXR

    DBDNAME=ITEMDBDL

    SEGMENT COUNT LEVEL RECORD

LP  PARTT     789   01   PART

LC  DETAIL   3542   02   DETAIL

    TOTAL:   4331   RECORDS READ

             3542   LOGICAL RECORDS WRITTEN

*** PFXR PROCESSING COMPLETE
```

Figure 67. Sample prefix resolution report

Step 5: Workfile Hierarchical Sort

The workfile produced by the prefix resolution step (Step 4) contains the logical child records with updated prefixes. The logical child records, though, still remain as sorted by the workfile sort/merge (Step 3). In other words, they are sequenced as they were under their logical parents (even though the logical parents do not appear in the prefix resolution workfile). Before the updated logical child records can be written out to replace the records originally stored in the CA IDMS/DB database (by Step 2), they must be resorted back into the original DL/I hierarchical sequence. The workfile hierarchical sort performs this operation.

To perform the workfile hierarchical sort, you must use your own sort/merge facility.

Operation

The workfile hierarchical sort performs the following operations:

1. Accessing the prefix-resolved workfile from Step 4
2. Sorting the workfile so that the logical child records are sequenced as in the original DL/I hierarchy

To execute the workfile hierarchical sort step, use the JCL in CA IDMS DLI Transparency JCL (see page 257).

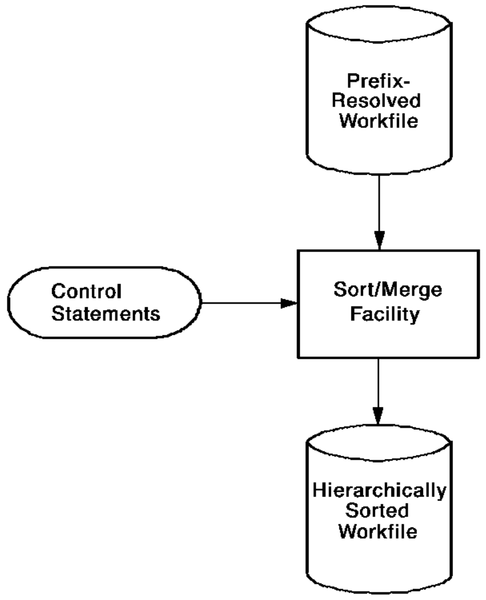


Figure 68. Workfile hierarchical sort

Step 6: Prefix Update

The prefix update step updates the logical child records in the CA IDMS/DB database with the prefixes (concatenated keys) generated by the prefix resolution step (Step 4). For input, it uses the hierarchically sorted workfile from Step 5. After updating the logical child database records with the correct prefixes, it writes them back to the database and connects them to their logical parents within the CA IDMS/DB sets.

This step completes the database load for logically related databases.

Operation

The prefix update step performs the following operations:

1. Accessing the IPSB load module
2. Accessing the hierarchically sorted workfile from Step 5
3. Obtaining the already loaded logical child records from the CA IDMS/DB database
4. Moving the prefix (logical parent concatenated key) from each workfile record into the corresponding database record

5. Writing the updated logical child records back to the database
6. Connecting each logical child database record with its related logical parent database record (that is, establish correct set pointers)
7. Producing a report showing the results of the processing

To execute the prefix update step, use the JCL in CA IDMS DLI Transparency JCL (see page 257).

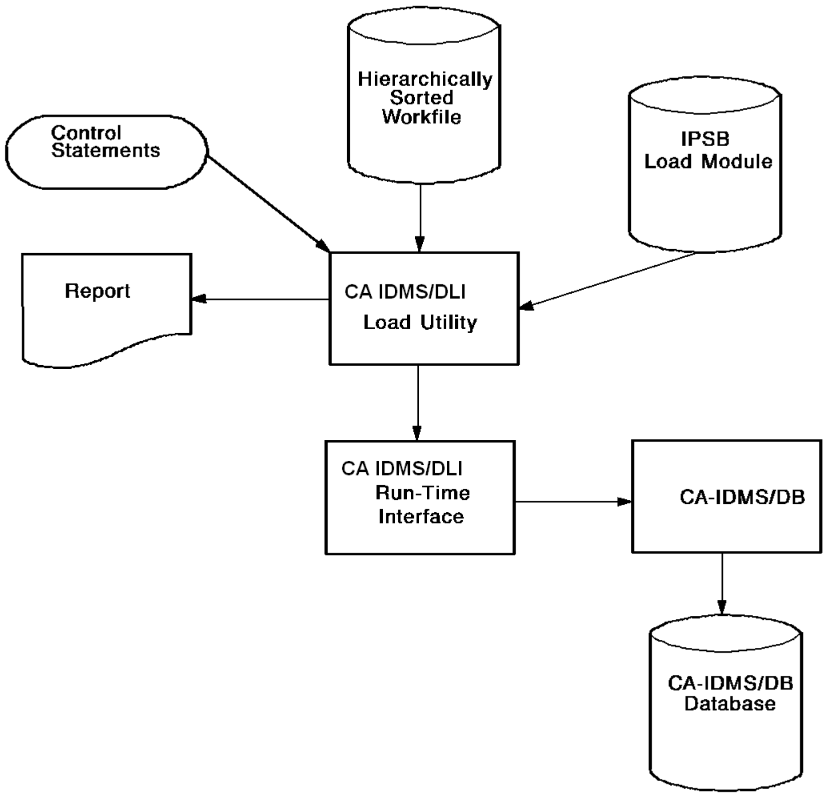


Figure 69. Prefix update

Report

The report produced by the prefix update step lists:

- The DBDNAME for the DL/I logical child database
- The name and level for each DL/I logical child segment
- The number of logical child records found and processed

```
*** CA IDMS/DLI TRANSPARENCY DATABASE LOAD

    PROCESS=PFXU

    DBDNAME=ITEMDBDL

    SEGMENT COUNT LEVEL RECORD

LC  DETAIL  3542   02   DETAIL

    TOTAL:  3542  RECORDS READ

*** PFXU PROCESSING COMPLETE
```

Figure 70. Sample prefix update report

Chapter 7: Using CA IDMS DLI Transparency Within CA IDMS/DB Programs

This section contains the following topics:

About This Chapter

CA IDMS DLI Transparency can be used in the CA IDMS/DB environment. A program written to use CA IDMS/DB must conform to CA IDMS/DB programming standards. All CA IDMS DLI Transparency functions available to batch programs are available to CA IDMS/DB programs. No restrictions are imposed, and no additional or special capabilities are added.

In addition to the conversion considerations for batch programs, using CA IDMS DLI Transparency in the CA IDMS/DB environment requires these considerations:

- Data communications
- Language interface
- Schedule (PCB) call processing
- The CA IDMS DLI Transparency program definition table
- Operational considerations

This chapter discusses each of these issues.

Data Communications

When migrating programs from an IMS-DC environment to CA IDMS/DB, all IMS-DC data communications calls in the programs must be recoded as CA IDMS/DB mapping calls. Any IMS message formatting services (MFS) maps must also be recoded using MAPC.

Note: Any IMS-DB (DL/1) database calls that are not supported by CA IDMS DLI Transparency in batch are also not supported in the CA IDMS/DB environment and must be modified.

Language Interface

CA IDMS DLI Transparency provides a language interface module for use in the CA IDMS/DB environment. CA IDMS DLI Transparency provides a language interface module, IDMSDLIF, for use only in the CA IDMS/DB environment. Programs to be executed under CA IDMS/DB must be link edited with the CA IDMS/DB environment language interface (IDMSDLIF) and must not be link edited with IDMSDLLI, the batch language interface.

Schedule (PCB) Call Processing

When using CA IDMS DLI Transparency in the CA IDMS/DB environment, the schedule (PCB) call processing is performed on behalf of the application program. This is the same as in the CA IDMS DLI Transparency batch environment.

- **In the batch environments**, (either CA IDMS DLI Transparency or native DL/1), the IPSB or PSB name is specified in the region controller's parameters.
- **In the IMS-DC online environment**, the PSB name is associated with a program through the macro specifications used to create a table at IMS system generation.
- **In the CA IDMS/DB CA IDMS DLI Transparency environment**, the method used to associate an IPSB name with an application program is similar (but not identical) to the IMS-DC environment. An application program and an IPSB are associated through a table created prior to the use of the application program, but not necessarily at the time of the CA IDMS/DB system generation. This table is called the CA IDMS DLI Transparency **program definition table**.

The CA IDMS DLI Transparency Program Definition Table

How the Program Definition Table is Created

The CA IDMS DLI Transparency program definition table is created from user-supplied input to the CA IDMS DLI Transparency program definition table compiler (**IDMSDLTG**). This compiler produces assembler source output which is then assembled and link edited into a CDMSLIB load library (z/OS) or core-image library (z/VSE).

The CA IDMS DLI Transparency program definition table load module (z/OS) or phase (z/VSE) must always have the name **DLPDTAB**. Each application program that is to have an IPSB automatically scheduled must have an entry in the table. The information in each entry is the same as in a region controller's parameter list, but the format is different.

The CA IDMS DLI Transparency program definition table can be thought of as an extension to the CA IDMS/DB program definition table. Before any program can be added to the CA IDMS DLI Transparency program definition table, it must already be in the CA IDMS/DB program definition table. (For this to be true, you must have defined the program to the CA IDMS/DB system with a system generation ADD PROGRAM statement.)

Syntax

```
►►─── MODify PROgram program-name ─┬──────────────────────────┬──────────►
                                   └─ version is(=) nnnn ─────┘

 ►─── IPSB name is(=) ipsb-name ─────────────────────────────────────────►

 ►──┬───────────────┬────────────────────────────────────────────────────►
    ├─ TRACE ───────┤
    └─ NOTRACE ◄ ───┘

 ►──┬───────────────┬────────────────────────────────────────────────────►
    ├─ STAE ◄ ──────┤
    └─ NOSTAE ──────┘

 ►──┬─────────────────────────────┬──────────────────────────────────────►
    └─ NODENAME is(=) nodename ───┘

 ►──┬─────────────────────────┬──────────────────────────────────────────►
    └─ DBNAME is(=) dbname ───┘

 ►──┬─────────────────────────────┬──────────────────────────────────────►
    └─ DICTNODE is(=) dictnode ───┘

 ►──┬─────────────────────────────┬─────────────────────────────────────►◄
    └─ DICTNAME is(=) dictname ───┘
```

Parameters

program-name

Identifies the name of the application program (already defined to the system through a system generation ADD PROGRAM statement) to be modified to use CA IDMS DLI Transparency.

nnnn

Identifies the 1- to 4-digit version number that further qualifies the program.

ipsb-name

Identifies the name of the IPSB to be automatically scheduled for the program.

TRACE/NOTRACE

Indicates whether or not CA IDMS DLI Transparency will build and maintain an internal trace table for aid in debugging. NOTRACE is the default.

STAE/NOSTAE

Indicates whether or not CA IDMS DLI Transparency will trap program abnormal terminations and produce formatted information for aid in debugging. NOSTAE is the default.

NODENAME IS *nodename*

Specifies the nodename that will be used to bind the CA IDMS DLI Transparency run unit.

DBNAME IS *dbname*

Specifies the dbname that will be used to bind the CA IDMS DLI Transparency run unit.

DICTNODE IS *nodename*

Specifies the nodename for the dictionary that will be used to bind the CA IDMS DLI Transparency run unit.

DICTNAME IS *dictname*

Specifies the dictname that will be used to bind the CA IDMS DLI Transparency run unit.

The JCL necessary to execute the CA IDMS DLI Transparency program definition table compiler (IDMSDLTG) and to assemble and link edit the DLPDTAB output is shown below:

PROGRAM DEFINTION TABLE COMPILER

```
//DL   EXEC   PGM=IDMSDLTG
//STEPLIB DD  DSN=idms.loadlib,DISP=SHR
//SYSLST DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSPCH DD  DSN=&&SYSPCH,UNIT=disk,SPACE=(4000,(100,50))
//               DCB=(RECFM=FB,LRECL=80,BLKSIZE=4000),DISP=(NEW,PASS)
//SYSIPT DD  *
pdt input statements
/*
//ASM  EXEC  PGM=ASMA90
//SYSPRINT DD  SYSOUT=A
//SYSLIB DD  DSN=yourHLQ.CAGJMAC,DISP=SHR
//SYSUT1 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT2 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT3 DD UNIT=disk,SPACE=(cyl,(2,2))
//SYSPUNCH DD DSN=&&PDTB,UNIT=disk,DISP=(NEW,PASS),
//    SPACE=(80,(400,40))
//SYSIN  DD  DSN=&&SYSPCH,DISP=(OLD,DELETE)
//LINK      EXEC  PGM=HEWL
//SYSPRINT      SYSOUT=A
//SYSLIN DD  DSN=&&PDTB,DISP=(OLD,DELETE)
//SYSUT1 DD  UNIT=disk,SPACE=(trk,(20,5))
//SYSLMOD DD  DSN=idms.loadlib(DLPDTAB),DISP=SHR
```

idms.loadlib	data set name of the CA IDMS/DB load library containing the subschema description and IDMSDLTG

cyl,(2,2)	space to be allocated in bytes per cylinders
disk	symbolic device type for the disk file
&&PDTB	temporary data set containing the output from the assembly step
yourHLQ.CAGJMAC	data set name of the macro library
&&SYSPCH	temporary data set containing the output from program definition table compiler (IDMSDLTG)
trk,(20,5)	space to be allocated in bytes per tracks
4000,(100,50)	space to be allocated in bytes per blocks
80,(400,40)	space to be allocated in bytes per blocks
DLPDTAB	required link edit module name in the SYSLMOD statement.

Operational Considerations

System Definition and Initialization

IDMSDLTI

Before any CA IDMS/DB application program can use CA IDMS DLI Transparency, the CA IDMS DLI Transparency environment within CA IDMS/DB must be initialized. This is done using the initialization program called **IDMSDLTI**.

System Generation Statements Defining IDMSDLTI

The system generation must contain an ADD PROGRAM statement to define IDMSDLTI:

```
ADD PROGRAM IDMSDLTI LANGUAGE IS ASSEMBLER REENTRANT REUSABLE.
```

The system generation must also contain an ADD TASK statement to define a task code that invokes IDMSDLTI:

```
ADD TASK IDMSDLTI INVOKES IDMSDLTI.
```

No CA IDMS/DB programs may use CA IDMS DLI Transparency before IDMSDLTI has been run. It is recommended that the system definition also contain an ADD AUTOTASK statement to automatically run IDMSDLTI immediately after CA IDMS/DB has come up.

```
ADD AUTOTASK IDMSDLTI INVOKED AT STARTUP PREEMPT.
```

Note that the PREEMPT option is included on the autotask definition. This is recommended so that no application programs that use CA IDMS DLI Transparency start before CA IDMS DLI Transparency initialization is completed.

System Execution

The automatic scheduling of an IPSB associated with an application program (as defined in the CA IDMS DLI Transparency program definition table) is performed whenever the application program is linked to, either by the CA IDMS/DB system itself or from another application program.

- If an application program named in the CA IDMS DLI Transparency program definition table (DLPDTAB) is also associated with a CA IDMS/DB task code, then entering that task code in response to an ENTER NEXT TASK CODE message causes automatic scheduling of the IPSB before CA IDMS/DB passes control to the application program.
- The automatic scheduling is done during the linking process (that is, after the program issuing the LINK command gives up control but before the target program receives control) if an application program that is *not* named in the DLPDTAB links to an application program that *is* named in the DLPDTAB.

All application programs receiving control from a region controller (following the automatic scheduling) must be set up to receive the scheduled PCBs. This is the same as for CA IDMS DLI Transparency batch, IMS-DC, and IMS-DB.

Linking to lower level programs

An application program that receives control following the automatic scheduling may link (DC LINK) to lower level programs.

- If one or more scheduled PCBs are passed as parameters to the lower level program, the lower level program may issue DL1 calls using the passed PCBs.
- If a program is linked to as a lower level program, it must not be named in the DLPDTAB, since naming an application program in the DLPDTAB causes automatic scheduling to be performed. Automatic scheduling must not be performed on these lower level programs.

Termination processing

Automatic termination (TERM call) processing is performed for all application programs that have had an automatic scheduling call done. The termination processing is done at the time when the application program that had the automatic scheduling issues a DC RETURN. If the application program or any lower level programs it links to abnormally terminates (that is, the task thread is interrupted), the CA IDMS DLI Transparency run unit is abnormally terminated as well and any changes to the database are rolled back.

Appendix A: CA IDMS DLI Transparency Messages and Codes

This section contains the following topics:

What This Appendix is About

CA IDMS DLI Transparency issues codes and messages to report errors encountered during processing. This appendix contains codes and messages returned by:

- The run-time interface
- The Syntax Generator
- The IPSB compiler
- The Load Utility
- The IPSB decompiler

Run-Time Messages and Codes

At run time, errors can cause CA IDMS DLI Transparency to terminate processing or to return specific DL/I status codes to the DL/I application program. When CA IDMS DLI Transparency terminates processing, it issues abend codes that are unique to CA IDMS DLI Transparency. When DL/I status codes are returned to the program, however, they are directly related to CA IDMS/DB error-status codes. Presented below are the run-time abend codes, the DL/I status codes and their equivalent CA IDMS/DB run-time error-status codes, and the DL/I status codes determined by the CA IDMS DLI Transparency run-time interface.

Run-Time Abend Codes

At run time, specific conditions cause CA IDMS DLI Transparency to terminate processing. If CA IDMS DLI Transparency encounters one or more of these conditions, the system returns an abend code number. The following is a list of these codes and their meanings:

Abend Code	Code
0063	Invalid request. The CA IDMS DLI Transparency run-time system determined the request was not a BIND, FINISH, or SEND/RECEIVE call.
2163	Loaded IPSB has invalid format.
2166	Unsuccessful ready of area during PCB call processing.
2463	Loaded IPSB has invalid format.
2466	Unsuccessful ready of area during PCB call processing.
2469	The request-unit PROGRAM-ID was found to be invalid.
2472	Storage not available for CA IDMS DLI Transparency run-time work area.
2474	Unsuccessful load of IPSB by CA IDMS DLI Transparency run-time system.
2499	An error was detected during DLET processing. A ROLLBACK has been issued for this transaction.
3301	System internal error. A nonzero request unit status was returned while attempting to process a DL/I service call.
3302	System internal error. IDMSDLFE has been called with an invalid parameter list or invalid parameters.
3303	A nonzero request unit status was returned while attempting to process a DL/I database call.
3304	A nonzero return code resulted from an attempt to acquire storage.
3305	A nonzero request unit status was returned after attempting a BIND REQUEST UNIT.
3306	A nonzero request unit status code was returned after attempting a DL/I PCB schedule call.
3307	A DL/I database call was attempted with more than 15 segment search arguments.
3308	System internal error. An error condition was detected while building a buffer parameter list.

Abend Code	Code
3310	A nonzero request unit status was returned after attempting a DL/I term call.
3311	A nonzero return code was returned while attempting to free storage.
3312	System internal error. The PCB address list was found to be invalid after a PCB schedule call.
3313	Load of DL/I application program failed or BLDL failed (z/OS only).

DL/I Status Codes and Equivalent CA IDMS/DB Codes

CA IDMS/DB Error Codes

CA IDMS/DB error-status codes are relatively specific in error condition descriptions when compared to the somewhat general approach reflected by the DL/I error-status codes. This difference causes a number of CA IDMS/DB error-status codes to be roughly equivalent to a single DL/I status code. While this situation may hamper problem determination, it is the result of an attempt to simulate the DL/I system as closely as possible.

Some CA IDMS/DB Error Codes Have No DL/I Equivalent

Additionally, there are some error situations that can occur in CA IDMS/DB, for which there is no DL/I equivalent. In this case, a two-character DL/I-type error-status code has been assigned and is documented in the following cross-reference. The CA IDMS/DB conditions for which the DL/I-type codes have been assigned will most likely never appear, unless the CA IDMS DLI Transparency run-time system has detected an extremely unusual situation.

DL/I Status Codes Table

The table below presents DL/I status codes. One or more of these codes is returned to a DL/I application by the CA IDMS DLI Transparency run-time system should an error condition be detected by CA IDMS/DB or the CA IDMS DLI Transparency run-time interface.

The DL/I status-code table also includes the error descriptions and, where applicable, the corresponding CA IDMS/DB error-status codes, call types, and minor codes.

Note: For more information, see the *CA IDMS Messages and Codes Guide*.

DL/I Status	Error Description	CA IDMS/DB Information		
		Error/Status Code	Minor Code	Call Type
	No error	0000		
A0	Write error		76	
AB	Segment I/O area was required for a database command, but was not specified (EXEC DLI)			
AC	Segment name in segment search argument not in hierarchy			
AD	Invalid function. Either a SCHEDULE or TERM call was issued in BATCH, or a LOAD command was issued (EXEC DLI)			
AH	Segment selection required, but not specified for a command that requires at least one segment name to be specified (EXEC DLI)			
AI	Area not readied or READY failed	0301		FIND/ OBTAIN
	Area not readied or READY failed	1201		STORE
	Areas other than area of object record occurrence must be readied in correct usage mode	0221		ERASE
	Areas other than area of object record occurrence must be readied in correct usage mode	0721		CONNECT
	Areas other than area of object record occurrence must be readied in correct usage mode	0821		MODIFY

DL/I Status	Error Description	CA IDMS/DB Information		
		Error/Status Code	Minor Code	Call Type
	Areas other than area of object record occurrence must be readied in correct usage mode	1121		DISCONNECT
	Areas other than area of object record occurrence must be readied in correct usage mode	1221		STORE
	Database or journal file will not ready properly		70	
	Database page read not requested		65	
	Dynamic load of module failed		74	
	Page range for area being readied or page requested, not found in DMCL	0971		READY
	Subschema invoked does not match object tables	1467		BIND
AJ	Concatenated segment in path call, not at lowest level			
	Invalid segment search argument			
AK	Invalid segment search argument field name			
AM	Areas readied with incorrect usage mode	0209		ERASE
	Areas readied with incorrect usage mode	0709		CONNECT
	Areas readied with incorrect usage mode	0809		MODIFY
	Areas readied with incorrect usage mode	1109		DISCONNECT
	Areas readied with incorrect usage mode	1209		STORE
	No current record of run unit	0813		MODIFY
	PCB not sensitive to particular function (see PROCOPTS)			
	Record name is defined as mandatory automatic member of set name	0714		CONNECT

DL/I Status	Error Description	CA IDMS/DB Information		
		Error/Status Code	Minor Code	Call Type
	Record name not defined as optional member of set name	1115		DISCONNECT
	Statement format conflicts with location mode	0331		FIND/ OBTAIN
AO	Read error		75	
AT	Not enough space in run-time I/O area			
B1	Run unit not bound to DBMS		69	
B2	Run unit not bound or bound twice		77	
B3	Area wait deadlock has occurred		78	
BA	Db-key inconsistent with area in which specified record is stored	0302		FIND/ OBTAIN
BB	Db-key not in range of db-keys defined for stored record	1202		STORE
BC	No currency established for record name, set name, or area name	0306		FIND/ OBTAIN
BD	No currency established for record name, set name, or area name	0706		CONNECT
BE	No currency established for record name, set name, or area name	0806		MODIFY
BF	No currency established for record name, set name, or area name	1106		DISCONNECT
BG	No db-key for record to be stored	1212		STORE
BH	No current record of run unit	0313		FIND/ OBTAIN
BI	Record name already member of set name	0716		CONNECT
BJ	Current record not same type as record name	0220		ERASE
BK	Current record not same type as record name	0820		MODIFY
BL	Record name not currently member of set name	1122		DISCONNECT
BM	Invalid area name used	0323		FIND/ OBTAIN
BN	No current of set name established	0725		CONNECT

DL/I Status	Error Description	CA IDMS/DB Information		
		Error/Status Code	Minor Code	Call Type
BO	Areas included in subschema currently ready	0928		READY
BP	CALC values in user work area and current record not equal	0332		FIND/ OBTAIN
BQ	Record type inconsistent with set name	0206		ERASE
	Record type inconsistent with set name	0306		FIND/ OBTAIN
BR	No record with specified db-key	1261		STORE
BS	Area not available for update	0966		READY
BT	Page range for area being readied or page requested, not found in DMCL	0371		FIND/ OBTAIN
BU	Record not bound		18	
BV	Db-key KEEP deadlock		29	
BW	Record occurrence not correct type		62	
BX	Invalid parameter list		63	
BY	CALC data item not described properly		64	
BZ	CICS interface not requested		68	
CA	Unsupported command received by run-time system			
CD	Attempted privacy breach, or invalid use of ERASE	0210		ERASE
	Attempted privacy breach, or invalid use of ERASE	0310		FIND/ OBTAIN
	Attempted privacy breach, or invalid use of ERASE	0710		CONNECT
	Attempted privacy breach, or invalid use of ERASE	0810		MODIFY
	Attempted privacy breach, or invalid use of ERASE	0910		READY
	Attempted privacy breach, or invalid use of ERASE	1110		DISCONNECT
	Attempted privacy breach, or invalid use of ERASE	1210		STORE

DL/I Status	**Error Description**	**CA IDMS/DB Information**		
		Error/Status Code	**Minor Code**	**Call Type**
DA	Sensitive field has been changed (REPL, DLET)			
DJ	Invalid command sequence for DLET. DLET call not preceded by HOLD TYPE call, or REPL call			
DX	No current of set name established	0225		ERASE
DX	Record occurrence is owner of nonempty set occurrence	0230		ERASE
DX	Segment to be deleted has nondeleted, dependent segments			
DX	Segment to be deleted participates in an inversion			
GB	End of database condition			
GB	End of set, area, index	0307		FIND/ OBTAIN
GD	Segment search argument(s) required for call			
GE	Not found condition			
	Record or index entry not found	0326		FIND/ OBTAIN
GP	Error in parentage			
II	Operation would have violated DUPLICATES NOT ALLOWED	1205		STORE
	Segment already exists (DUPLICATES NOT ALLOWED)			
IX	Insert rule violated			
	No current of set name established	1225		STORE
NI	Operation would have violated DUPLICATES NOT ALLOWED	0705		CONNECT
	Operation would have violated DUPLICATES NOT ALLOWED	0805		MODIFY
NX	Error loading user-supplied index suppression exit			

DL/I Status	**Error Description**	**CA IDMS/DB Information**		
		Error/Status Code	**Minor Code**	**Call Type**
RX	Invalid command sequence for REPL. REPL call not preceded by HOLD TYPE call, or REPL call			
	No current of set name established	0825		MODIFY
	Violated REPLACE rule			
TI	Error in PATH INSERT data transfer specification. Data transfer must be specified for all segments between the first parent segment requesting data transfer, and the object segment (EXEC DLI)			
TO	Error in PATH REPLACE. Segment usage in path replace does not match those segments retrieved in the last GET command (EXEC DLI)			
TP	Invalid PCB INDEX. An invalid PCB number has been specified. The scheduled PSB has no PCB satisfying the request (EXEC DLI)			
V1	Invalid length for variable-length record	0855		MODIFY
	Invalid length for variable-length record	1255		STORE
V2	SEGLENGTH is required but not specified, or is zero or negative (EXEC DLI)			
V3	FIELDLENGTH is required but not specified, or is zero or negative (EXEC DLI)			
V4	Invalid SEGLENGTH specified for a variable length segment (EXEC DLI)			
V5	OFFSET is greater than SEGLENGTH, or is zero or negative. This applies to segments having a logical relationship (EXEC DLI)			
V6	No KEYLENGTH specified, but is required (EXEC DLI)			
X1	Invalid record name or set name	0208		ERASE

DL/I Status	Error Description	CA IDMS/DB Information		
		Error/Status Code	Minor Code	Call Type
	Invalid record name or set name	0308		FIND/ OBTAIN
	Invalid record name or set name	0708		CONNECT
	Invalid record name or set name	1108		DISCONNECT
	Invalid record name or set name	1208		STORE
	Invalid record name or set name	1408		BIND
X2	No space in area for record to be stored	1211		STORE
X3	All required set type relationships not defined	0233		ERASE
	All required set type relationships not defined	0833		MODIFY
	All required set type relationships not defined	1233		STORE
X4	Insufficient memory for COMPRESS/DECOMPRESS		56	
XX	Error in obtaining storage			
	Insufficient memory for load or storage allocation	1472		BIND
	Insufficient memory for load or storage allocation		72	

Non-Run-Time Messages and Codes

This section lists the messages that can be returned by the CA IDMS DLI Transparency non-run-time components:

- Syntax generator
- IPSB compiler
- Load utility

Message Format

The format of the non-run-time messages is as follows:

message-number message-severity-level message-text

The message items have the following meanings:

- *Message-number* indicates the message number.
- *Message-severity-level* can be one of the following:
  - **W** (Warning) –– Alerts you to potential problems; processing continues.
  - **E** (Error) –– Indicates a nonfatal error; processing continues.
  - **F** (Fatal) –– Indicates a fatal error; the component terminates processing.

Note: Load utility and IPSB decompiler messages do not include a severity level.

- *Message-text* is the message issued in response to the error.

Messages Listed by Message Number

The messages are listed in numerical order by message number. For each message, an explanation is provided as well as an indication of the component that issues it.

- (Syntax generator)
- (IPSB compiler)
- (Load utility)
- (IPSB decompiler)

Error code	Message
220001	**'EJECT' NOT ALONE ON CARD. TOKEN ASSUMED.** EJECT must appear as the only entry on the card unless it is to be used as other that a compiler directive. **Severity:** W
220002	**INVALID 'SPACE' COMMAND PARAMETER.** SPACE must be followed by a blank and, optionally, a 1-digit number greater than 0 indicating the number of lines to be spaced. **Severity:** W
220003	**'SPACE' NOT ALONE ON CARD. TOKEN ASSUMED.** SPACE must appear as the only entry on the card unless it is to be used as other than a compiler directive. **Severity:** W
220004	**SEQUENCE ERROR. RUN ABORTED.** Input was out of sequence. **Severity:** F

Error code	Message
220005	**STRING EXCEEDS MAXIMUM OR AVAILABLE LENGTH.** The specified string exceeds the maximum allowable length for this parameter. **Severity:** E
220006	**HEX STRING EXCEEDS MAXIMUM OR AVAILABLE LENGTH.** The specified hex string exceeds the maximum allowable length for this parameter. **Severity:** E
220007	**HEX STRING CONTAINS INVALID CHARACTERS.** The specified hex string contains invalid hexadecimal characters. **Severity:** E
220008	**Ictl-parameter INVALID ICTL PARAMETER SPECIFICATION.** The ICTL parameter was incorrectly specified. Check the syntax. (IPSB compiler) **Severity:** E
220009	**Octl-parameter INVALID OCTL PARAMETER SPECIFICATION.** The OCTL parameter was incorrectly specified. Check the syntax. (IPSB compiler) **Severity:** E
220010	**Iseq-parameter INVALID ISEQ PARAMETER SPECIFICATION.** The ISEQ parameter was incorrectly specified. Check the syntax. (IPSB compiler) **Severity:** E
220011	**Core-size-parameter INVALID COMPILER TABLE SIZE SPECIFICATION.** The core-size parameter must be a 1- to 6-digit number optionally followed by a K. There must be at least one space between the number and the K. (IPSB compiler) **Severity:** F
220012	**Parameter UNEXPECTED END OF FILE (PERIOD MISSING).** Invalid syntax has been encountered. Check for missing periods. (IPSB compiler) **Severity:** E

Error code	Message
220013	**Keyword UNKNOWN KEYWORD FOR STATEMENT TYPE.** The keyword encountered is not valid for the current statement type. (IPSB compiler) **Severity:** E
220014	**UNEXPECTED END OF FILE SEARCHING FOR STATEMENT.** End of file occurred before sufficient control input was found. **Severity:** E
220015	**Keyword INVALID STATEMENT TYPE. SKIPPING TO NEXT PERIOD.** The statement encountered is not valid for the current section. (IPSB compiler) **Severity:** E
220016	**Ipsb-name MISSING OR INVALID IPSB NAME.** The IPSB name must be a 1-to 8-character alphanumeric string. (IPSB compiler) **Severity:** E
220017	**Subschema-name MISSING OR INVALID SUBSCHEMA NAME.** The subschema name must be a 1- to 8-character alphanumeric string. (IPSB compiler) **Severity:** F
220018	**Language-parameter INVALID PROGRAM LANGUAGE SPECIFICATION.** Language must be COBOL, PL/I, or Assembler. (IPSB compiler) **Severity:** E
220019	**Subschema-name SUBSCHEMA NOT FOUND IN LOAD LIBRARY.** The subschema named could not be found. (IPSB compiler) **Severity:** F
220020	**Subschema-name ERROR LOADING SUBSCHEMA MODULE.** The subschema named could not be loaded. (IPSB compiler) **Severity:** F
220021	**INSUFFICIENT STORAGE FOR COMPILATION.** The IPSB compiler has run out of an internal work space. Contact technical support. (IPSB compiler) **Severity:** F

Error code	Message
220022	**Subschema-name LOADED SUBSCHEMA MODULE INVALID.** The subschema named was used to load a module, but the module is not a valid subschema. (IPSB compiler) **Severity:** F
220023	**Parameter-name DIAGNOSTIC TABLE SIZE EXCEEDED. TOO MANY ERRORS.** Too many errors have occurred, causing an overflow of the table. Correct the previous errors. (IPSB compiler) **Severity:** F
220025	**Keyword INVALID KEYWORD, SKIPPING TO NEXT PERIOD.** The keyword encountered is not valid. Compilation resumes with the next statement. (IPSB compiler) **Severity:** E
220026	**MISSING KEYWORD, SKIPPING TO NEXT PERIOD.** A required keyword is missing. compilation resumes with the next statement.
220027	**Parameter INVALID, PARAMETER TOO LONG.** The character string specified is greater in length than the maximum allowed for this parameter. (IPSB compiler) **Severity:** E
220031	**Area-name INVALID AREA NAME.** The character string as specified is not a valid area name. (IPSB compiler) **Severity:** E
220032	**Area-name AREA NOT DEFINED IN SUBSCHEMA.** All areas used in an IPSB must be defined in the subschema previously specified in the IPSB statement. (IPSB compiler) **Severity:** E
220033	**Usage-mode INVALID USAGE-MODE, SHARED RETRIEVAL ASSUMED.** The usage mode as specified is incorrect. Check the syntax. (IPSB compiler) **Severity:** W
220034	**Area-name AREA HAS BEEN PREVIOUSLY SPECIFIED.** An area name can be used in only one AREA statement. (IPSB compiler) **Severity:** E

Error code	Message
220035	**Record-name INVALID RECORD NAME.** A record name must be a 1- to 16-character alphanumeric string. (IPSB compiler) **Severity:** E
220036	**Option INVALID DELETE OPTION, ERASE ALL ASSUMED.** The DELETE option was specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** W
220037	**Field-name INVALID FIELD NAME.** A field name must be a 1- to 8-character alphanumeric string. (IPSB compiler) **Severity:** E
220038	**Stored-option INVALID, STORED PHYSICALLY ASSUMED.** The stored option was specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** W
220039	**Position INVALID/MISSING STARTING POSITION.** The starting position must be a 1- to 5-digit number, greater than 1, and less than the *record-length* - 1. (IPSB compiler) **Severity:** E
220040	**Length INVALID/MISSING LENGTH SPECIFICATION.** The length must be a 1- to 5-digit number that indicates the length of the field. (IPSB compiler) **Severity:** E
220041	**Usage INVALID USAGE, DISPLAY ASSUMED.** The usage has been specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** W
220042	**Record-name RECORD HAS BEEN PREVIOUSLY SPECIFIED.** A record name can appear in only one RECORD statement. (IPSB compiler) **Severity:** E
220043	**Record-name RECORD NOT DEFINED IN SUBSCHEMA.** The record named must be defined in the subschema previously specified in the IPSB statement. (IPSB compiler) **Severity:** E

Error code	Message
220044	**Field-name FIELD HAS BEEN PREVIOUSLY SPECIFIED.** A field name must be unique within any one record definition. (IPSB compiler) **Severity:** E
220045	**Start-position STARTING POSITION INVALID IF STORED VIRTUALLY.** Starting position must not be specified if the field is stored virtually. This clause is ignored. (IPSB compiler) **Severity:** W
220046	**Record-name PREVIOUS RECORD HAS ONLY ONE CONCATENATED KEY.** The record preceding the current record has only one destination parent concatenated key defined. This can cause abnormal termination at run time. (IPSB compiler) **Severity:** E
220047	**Key-name LOGICAL CONCATENATED KEY PREVIOUSLY DEFINED.** Only one logical destination parent concatenated key field can be defined within any one record. (IPSB compiler) **Severity:** E
220048	**Key-name PHYSICAL CONCATENATED KEY PREVIOUSLY DEFINED.** Only one physical destination parent concatenated key field can be defined within any one record. (IPSB compiler) **Severity:** E
220049	**Parameter LENGTH OR STARTING POSITION INVALID FOR /SX.** Length and/or starting position must not be specified for /SX fields. (IPSB compiler) **Severity:** W
220050	**Field-name INVALID INDEXED FIELD NAME.** An indexed field name must be a 1- to 8-character alphanumeric string. (IPSB compiler) **Severity:** E
220052	**Record-name INVALID/MISSING TARGET RECORD.** If present, the target record as specified is not a valid record name. (IPSB compiler) **Severity:** E

Error code	Message
220053	**Record-name INVALID/MISSING SOURCE RECORD.** If present, the source record as specified is not a valid record name. (IPSB compiler) **Severity:** E
220054	**Record-name INVALID/MISSING POINTER RECORD.** If present, the pointer record as specified is not a valid record name. (IPSB compiler) **Severity:** E
220055	**Constant INVALID SHARED INDEX CONSTANT.** The shared index constant must be a 1-byte, self-defining Assembler constant enclosed in double quotes. (IPSB compiler) **Severity:** E
220056	**Field-name MISSING SEARCH FIELD(S).** At least one field must be specified as a search field. (IPSB compiler) **Severity:** E
220057	**Field-name INVALID FIELD(S) SPECIFICATION.** A field name has been specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** E
220058	**Value INVALID NULL INDEX VALUE.** The null index value must be a 1-byte, self-defining Assembler constant enclosed in double quotes, or BLANK or ZERO. (IPSB compiler) **Severity:** E
220059	**Index-name INDEX HAS BEEN PREVIOUSLY SPECIFIED.** An index name can be used in only one INDEX statement. (IPSB compiler) **Severity:** E
220061	**Record-name RECORD NOT DEFINED IN RECORD SECTION.** The record named must be defined by a RECORD statement in the RECORD SECTION. (IPSB compiler) **Severity:** E
220062	**Field-name FIELD NOT DEFINED IN SOURCE RECORD.** The field named must be defined within the source record definition in the RECORD SECTION. (IPSB compiler) **Severity:** E

Error code	Message
220063	**Record-name NO SEQUENCE FIELD DEFINED FOR POINTER RECORD.** All pointer records must have a sequence field defined corresponding to the sort-key field of the indexed set of which it is a member. (IPSB compiler) **Severity:** E
220064	**Length LENGTH SPECIFIED GREATER THAN SUBSCHEMA LENGTH.** The record length specified must not be greater than the length as defined in the subschema. (IPSB compiler) **Severity:** E
220065	**Length LENGTH SPECIFIED LESS THAN CONTROL LENGTH.** The minimum record length specified is less than the control length of the record. It is rounded up to the control length. (IPSB compiler) **Severity:** W
220066	**Length INVALID/MISSING LENGTH SPECIFICATION.** The length must be a 1- to 5-digit number that indicates the length of the segment this record represents. (IPSB compiler) **Severity:** E
220067	**Record-name RECORD IS NOT VARIABLE LENGTH IN SCHEMA.** A maximum and minimum length has been specified, but the record is not of variable length. (IPSB compiler) **Severity:** E
220069	**Clause MISSING PARENT CLAUSE.** A parent segment must be specified for all but root segments. (IPSB compiler) **Severity:** E
220070	**Access-method INVALID ACCESS METHOD.** The access method has been specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** E
220071	**Dbd-name INVALID/MISSING DBDNAME.** The DBD name must be a 1- to 8-character alphanumeric string. (IPSB compiler) **Severity:** E

Error code	Message
220072	**Option INVALID/MISSING PROCESSING OPTIONS.** Processing options have been specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** E
220073	**INVALID KEY FEEDBACK LENGTH.** The value specified for the key feedback length must be numeric. (IPSB compiler) **Severity:** E
220074	**Option INVALID POSITIONING.** Positioning has been specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** E
220077	**Segment-name INVALID SEGMENT NAME.** A segment name must be a 1- to 8-character alphanumeric string. (IPSB compiler) **Severity:** E
220078	**Set-name INVALID SET NAME.** A set name must be a 1- to 16-character alphanumeric string. (IPSB compiler) **Severity:** E
220079	**Record-name INVALID RECORD NAME.** A record name must be a 1- to 16-character alphanumeric string. (IPSB compiler) **Severity:** E
220080	**Use-option INVALID USE OPTION.** The USE option has been specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** E
220082	**Option PROCESSING SEQUENCE MUST BE SPECIFIED.** Processing sequence must be specified for all access methods except HDAM. (IPSB compiler) **Severity:** E
220083	**Option PROCESSING SEQUENCE MUST NOT BE SPECIFIED.** Processing sequence must not be specified if the access method is HDAM. (IPSB compiler) **Severity:** E

Error code	Message
220085	**Index-name INDEX NOT DEFINED IN INDEX SECTION.** The indexed field named must be defined by an INDEX statement in the INDEX SECTION. (IPSB compiler) **Severity:** E
220086	**Segment-name SEGMENT HAS BEEN PREVIOUSLY DEFINED.** A segment name can be used only once within any one PCB. (IPSB compiler) **Severity:** E
220087	**Record-name RECORD NOT DEFINED IN RECORD SECTION.** The record named must be defined by a RECORD statement in the RECORD SECTION. (IPSB compiler) **Severity:** E
220088	**Segment-name SEGMENT NOT PREVIOUSLY DEFINED.** The segment named must be previously defined by a SEGMENT statement within the same PCB. (IPSB compiler) **Severity:** E
220489	**Parent-name PARENT MUST NOT BE SPECIFIED ON ROOT SEGMENTS.** Remove the PARENT clause from this SEGMENT statement. (IPSB compiler) **Severity:** E
220090	**Set-name SET NOT DEFINED IN SUBSCHEMA.** The set named must be defined in the subschema previously specified in the IPSB statement. (IPSB compiler) **Severity:** E
220091	**Set-name INVALID USE OF SET.** Processing sequence set can be specified only if the access method is HISAM or INDEX. (IPSB compiler) **Severity:** E
220092	**INVALID MEMBER OF SET.** The IDMS record is not a valid member of the set specified. (IPSB compiler) **Severity:** E
220093	**INVALID OWNER OF SET.** The IDMS record is not a valid owner of the set specified. (IPSB compiler) **Severity:** E

Error code	Message
220094	**Index-name INVALID USE OF INDEX.** Processing sequence index can be specified only if the access method is HIDAM or secondary index. (IPSB compiler) **Severity:** E
220095	**Segment-name INVALID INVERSION OF SEGMENTS.** A segment appears in the inversion that is not in the hierarchic path of the destination parent in its physical database. (IPSB compiler) **Severity:** E
220096	**Rule INVALID RULE SPECIFIED.** An insert or replace rule has been specified incorrectly. Check the syntax. (IPSB compiler) **Severity:** E
220097	**Parent-name LOGICAL PARENT CONCATENATED KEY IS UNDEFINED.** If a logical destination parent is specified, its concatenated key must be defined within the record definition of the logical child. (IPSB compiler) **Severity:** E
220098	**Parent-name PHYSICAL PARENT CONCATENATED KEY IS UNDEFINED.** If a physical destination parent is specified, its concatenated key must be defined within the record definition of the logical child. (IPSB compiler) **Severity:** E
220100	**set-name INVALID INDEXED SET NAME.** The name specified for an indexed set must be a 1- to 16-character name. (IPSB compiler) **Severity:** E
220101	**set-name INDEXED SET NOT DEFINED IN SUBSCHEMA.** The specified indexed set must be defined in the subschema. (IPSB compiler) **Severity:** E
220102	**subschema-name SUBSCHEMA DOES NOT CONTAIN INDEXED SETS.** 'INDEXED-SET' was specified in the IPSB SECTION, but no indexed sets were found in the subschema. (IPSB compiler) **Severity:** E

Error code	Message
220103	**INVALID EXIT ROUTINE NAME** The index suppression exit routine name must be a 1- to 8-character name. (IPSB compiler) **Severity:** E
221001	**'EJECT' NOT ALONE ON CARD. TOKEN ASSUMED.** EJECT must appear as the only entry on the card unless it is to be used as other than a compiler directive. (Syntax Generator) **Severity:** W
221002	**INVALID 'SPACE' COMMAND PARAMETER.** SPACE must be followed by a blank and, optionally, a 1-digit number greater than 0 indicating the number of lines to be spaced. (Syntax Generator) **Severity:** W
221003	**SPACE NOT ALONE ON CARD. TOKEN ASSUMED.** SPACE must appear as the only entry on the card unless it is to be used as other than a compiler directive. (IPSB compiler) **Severity:** W
221004	**SEQUENCE ERROR. RUN ABORTED.** Input was out of sequence. (Syntax Generator) **Severity:** F
221005	**STRING EXCEEDS MAXIMUM OR AVAILABLE LENGTH.** The character string specified is greater in length than the maximum allowed for this parameter. (Syntax Generator) **Severity:** E
221006	**HEX STRING EXCEEDS MAXIMUM OR AVAILABLE LENGTH.** The hexadecimal string specified is greater in length than the maximum allowed for this parameter. (Syntax Generator) **Severity:** E
221007	**HEX STRING CONTAINS INVALID CHARACTERS.** Other than valid characters appear in the hexadecimal string specified. (Syntax Generator) **Severity:** E
221009	**Dbd-name - DBD NOT FOUND IN LIBRARY.** The DBD name listed prior to this message was not in the library designated by the STEPLIB JCL statement. (Syntax Generator) **Severity:** E

Error code	Message
221010	**Dbd-name - DBD NOT LOADED - ERROR.** An error has occurred during load processing for the specified DBD. (Syntax Generator) **Severity:** E
221011	**Dbd-name - DBD NOT LOCATED.** A DBD was not located in the already loaded chain. This is a system internal error. (Syntax Generator) **Severity:** E
221012	**UNEXPECTED END OF FILE PROCESSING IPSB STATEMENT.** End of file occurred before sufficient control input was found. (Syntax Generator) **Severity:** F
221013	**Keyword UNKNOWN KEYWORD FOR STATEMENT TYPE.** The keyword encountered is not valid for the current statement type. (Syntax Generator) **Severity:** E
221014	**PRIMARY INDEX NOT FOUND.** The LCHILD statement for the named index was not found. (Syntax Generator) **Severity:** E
221015	**Keyword INVALID STATEMENT TYPE. SKIPPING TO NEXT PERIOD.** The statement encountered is not a valid statement type. (Syntax Generator) **Severity:** E
221016	**Segment-name - SEGMENT NOT FOUND IN DBD.** The named segment was not found in the appropriate DBD. (Syntax Generator) **Severity:** E
221017	**Segment-name - SEGMENT PHYSICAL OWNER NOT FOUND.** An attempt to establish a path to the physical owner of a destination parent segment was unsuccessful. (Syntax Generator) **Severity:** E

Error code	Message
221018	**Segment-name - SEGMENT HAS NO FIELDS - REQUIRED.** A logical child segment has been encountered that has no fields defined for it. (Syntax Generator) **Severity:** E
221019	**Segment-name - SEGMENT PARENT NOT FOUND.** While determining the length of a logical child concatenated key, the root segment could not be found. The probable cause is an incorrectly defined path. (Syntax Generator) **Severity:** E
221020	**Segment-name - SEGMENT SEQUENCE FIELD REQUIRED.** The sequence field for an index pointer record was not found. (Syntax Generator) **Severity:** E
221021	**Psb-name - PSB NOT FOUND IN LIBRARY.** The named PSB was not found in any library accessible through a STEPLIB JCL statement. (Syntax Generator) **Severity:** F
221022	**Psb-name - PSB NOT LOADED - ERROR.** An error has occurred during load processing for the specified PSB. The PSB named could not be loaded. (Syntax Generator)
221023	**GENERATION TERMINATED - TOO MANY ERRORS.** Too many errors have occurred for this processing run. (Syntax Generator) **Severity:** F
221024	**Dbd-name DBD NOT VALID FOR USE WITH CA IDMS DLI Transparency.** A loaded DBD has been found to contain an invalid format. The probable cause is that the IBM version of the DBD was loaded. Use the CA IDMS DLI Transparency assembled DBD. (Syntax Generator) **Severity:** F
221025	**Psb-name PSB NOT VALID FOR USE WITH CA IDMS/DLI Transparency.** A loaded PSB has been found to contain an invalid format. Use the CA IDMS DLI Transparency assembled PSB. (Syntax Generator) **Severity:** F

Error code	Message
221026	**Dbd-name DBD FOR LOGICAL PARENT NOT FOUND IN ANY PCB.** During generation of a load IPSB, the named DBD was referenced as a logical parent DBD, but no PCB in the load PSB defined the logical parent as a physical segment. (Syntax Generator) **Severity:** F
221500	**DATABASE CAPACITY EXCEEDED.** Database capacity is not sufficient to load all necessary records. Reallocate the database files with more space. Issued by Step 2. (Load Utility)
221501	**SEGMENT=segment-name NOT IN IPSB.** The named segment has been found in the input file, or workfile, but is not defined in any PCB within the IPSB. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221503	**NO LOGICAL RELATIONSHIPS.** Database load Processing (Step 2) encountered no logical relationships. Steps 3 through 6 are not required to complete the database load. (Load Utility)
221506	**IPSB=ipsb-name NOT FOUND.** The named IPSB could not be loaded. Make sure that the correct IPSB resides in the data set(s) referenced by CDMSLIB. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221508	**INITIAL DATABASE LOAD COMPLETE.** Database load processing (Step 2) has been successfully completed. (Load Utility)
221509	**PREFIX RESOLUTION COMPLETE.** Prefix resolution processing (Step 4) has been successfully completed. (Load Utility) **Severity:** F
221510	**PREFIX UDPDATE COMPLETE.** Prefix update processing (Step 6) has been successfully completed. (Load Utility)
221511	**PCB DBDNAME=dbdname NOT FOUND.** Prefix resolution or prefix update processing failed to find a DBD in the IPSB that matches the named DBD. The named DBD was referenced in the logical workfile produced by the database load (Step 2). Issued by Steps 4 and 6. (Load Utility)

Error code	Message
221512	**INVALID INPUT CONTROL FORMAT.** Invalid processing control statements have been encountered. Rerun the step in question with correctly formatted control specifications. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221513	**PROCESSING TERMINATED-ERROR(S).** A fatal error condition was detected. This message is usually preceded by a message indicating the specific error condition. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221514	**SEGM=segment-name - NO LOGICAL PARENT.** A logical child record in the workfile has no corresponding logical parent record in the workfile. This message may be the result of an incomplete Step 3 sort, or it may indicate that multiple logical workfiles from Step 2 were not merged prior to the Step 3 sort. Issued by Steps 4 and 6. (Load Utility)
221516	**PARAMETER 'IPSB' REQUIRED.** The control format for a processing step is incomplete. Specify the IPSB name required for processing. Issued by Steps 2, 4, and 6. (Load Utility)
221517	**INVALID IPSB PROCOPTS - 'LOAD' REQUIRED.** Use of the Load Utility within the CA IDMS DLI Transparency Run-Time Interface requires that each PCB in the IPSB be specified with PROCOPT=LOAD. Issued by Steps 2, 4, and 6. (Load Utility)
221519	**DBDNAME=dbdname NOT IN IPSB.** The named DBD was not found in the IPSB. Use the same IPSB as you used in Step 2 processing. Issued by Steps 4 and 6. (Load Utility)
221521	**RELATED WORKFILES MISSING.** This message usually appears after messages 221514 and 221518, to indicate the probable error cause. Issued by Steps 4 and 6. (Load Utility)
221522	**NO FURTHER PROCESSING REQUIRED.** Database Load Processing (Step 2) has been successfully completed. No logical relationships were found, and no additional processing is necessary. (Load Utility)
221523	**STATUS=code RETURNED-SEGMENT= segment-name.** An unexpected DL/I status code has been returned to the Load Utility. This message usually indicates a fatal error. Issued by Steps 2 and 6. (Load Utility)

Error code	Message
221524	**CHECK IPSB FOR PROBLEM(S).** An error has been detected that may be related to an IPSB specification. Issued by Steps 2 and 6. (Load Utility)
221525	**UNEXPECTED END OF FILE-SYSIPT.** End of file was encountered before sufficient process control information was found. Issued by Steps 1, 4, and 6. (Load Utility)
221526	**INVALID IPSB FORMAT.** The IPSB that was loaded does not have a valid format. Issued by Steps 1, 4, and 6. (Load Utility) **Severity:** F
221527	**PARAMETER 'PROCESS=' REQUIRED.** The JCL for the step did not include the PROCESS control parameter. PROCESS= is required. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221528	**INSUFFICIENT STORAGE.** More main storage is required for successful processing. Increase the storage specification, and rerun the processing step in question. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221530	**CALC PROCESSING COMPLETE.** The Pre-Load CALC Processing (Step 1) has been successfully completed. (Load Utility) **Severity:** F
221531	**I/O ERROR ON FILE=SYS999.** An I/O error has been detected during file processing. Determine the nature of the cause, and re-run the processing step. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221532	**ERROR OPENING FILE=SYS999.** An attempt to open a required file has not been successful. Check for missing file definitions, or conflicts in file definitions. Issued by Steps 1, 2, 4, and 6. (Load Utility)
221542	**LOAD OF SUBSCHEMA=subschema-name FAILED.** The subschema specified in the IPSB was not available for CALC processing. Make sure that subschema is accessible through a STEPLIB JCL statement. Issued by Steps 1 and 2. (Load Utility)

Error code	Message
221543	**AREA=area-name NOT IN SUBSCHEMA.** The specified area name was found in the load IPSB, but was not found in the subschema. Make sure that the subschema contains all required area names. Issued by Steps 1 and 2. (Load Utility)
223902	**ipsb-name COMPILE DATE: mm/dd/yy TIME: HHmmsshh** Issued during IPSB validation, this indicates the original IPSB compilation date/time. Date is in month/day/year format. Time is in hours/minutes/seconds/hundreth seconds. (IPSB decompiler)
223902	**subschema COMPILE DATE: mm/dd/yy TIME: HHmmsshh** Issued during IPSB validation, this indicates the corresponding subschema compilation date/time. Date and time formats are as indicated above. (IPSB decompiler)
223903	**REQUESTED MODULE IS NOT AN IPSB.** IPSB validation has determined that the loaded module does not contain a format similar to that of a CA IDMS DLI Transparency IPSB. Due to the environment associated with CA IDMS DLI Transparency, this module may be a native DL/I PSB. (IPSB decompiler)
223904	**REQUESTED IPSB RELEASE LEVEL NOT SUPPORTED.** IPSB validation has found that the requested IPSB loaded for decompilation is for a release level of CA IDMS DLI Transparency that is not supported by the IPSB decompiler. (IPSB decompiler)
223905	**ERROR IN LOAD OF IPSB=ipsbname** Issued when an error has occurred during an attempt to access the specified IPSB load module. (IPSB decompiler)
223906	**ERROR OPENING FILE=SYSxxx** Produced when a request to open a specific file has been unsuccessful, probably due to missing or conflicting file definitions. *SYSxxx* can include SYSIPT, SYSLST, OR SYSPCH. (IPSB decompiler)

Error code	Message
223907	**ERROR ON FILE=SYSxxx FUNC=xxxx STAT=xxxx**. Produced when a request to close, or read/write to/from a specific file has resulted in an error condition. Here, SYS*xxx* can be the file names SYSIPT, SYSLST, or SYSPCH. The FUNC= and STAT= operands of the message relate to the processing functions and resulting statuses that are common to the I/O utility module IDMSUTIO (IDMSUTIO is used for all I/O functions for the Decompiler). (IPSB decompiler)
223908	**UNEXPECTED END OF FILE - SYSIPT** If no valid IPSB-directive control statement is encountered before end-of-file occurs while reading the SYSIPT input file, this message is issued, and decompilation terminates. (IPSB decompiler)
223909	**CA IDMS DLI Transparency IPSB DECOMPILATION TERMINATED-ERROR(S).** If an error occurs during SYSIPT processing, IPSB loading, IPSB identity and release level validation, or SYSLST or SYSPCH processing, this message is issued as an indication of the final status of the current processing run. (IPSB decompiler)
223910	**CA IDMS DLI Transparency IPSB DECOMPILATION COMPLETE** Issued when IPSB decompilation process has completed without encountering any problem situations. This is the final message issued by the decompiler after a successful run. (IPSB decompiler)

Appendix B: CA IDMS DLI Transparency Software Components

This section contains the following topics:

About This Appendix

CA IDMS DLI Transparency has four major software components:

- The syntax generator
- The IPSB compiler
- The runtime interface
- The load utility

Each component is described in this section.

The Syntax Generator

Input to the Syntax Generator

The syntax generator consists of the IDMSDLPG module. For input, it accepts a DL/I PSB and the DBDs referenced by the PCBs included in the PSB. The source code for the PSB and DBDs must be assembled using the CA-supplied macros before inputting them to the syntax generator.

In addition to the assembled PSB and DBDs, the syntax generator requires user-supplied input statements. The input statements direct the generator to produce source statements for an IPSB load module and any of the following CA IDMS/DB entities:

- Schema
- DMCL
- Subschema

Output from the Syntax Generator

When executed, the syntax generator reads in and extracts the DL/I definitions reflected in the assembled PSB and DBDs. Based on the DL/I definitions, the generator creates corresponding source statements for the IPSB load module and the requested CA IDMS/DB modules.

Review the Source Statements

The user must review the IPSB and CA IDMS/DB source statements to make sure that they reflect the dependencies that are present, either explicitly or implicitly, in the DL/I definitions. For example, does every logical child segment have its physical parent segment defined? If an IPSB is to be used with the load utility, there are special load utility considerations that the user must include in the IPSB source.

After reviewing and, if necessary, modifying the source statements, the user must input them to the appropriate compiler to produce the required load module. Operation of the IPSB compiler is described below. Operation of the CA IDMS/DB schema and subschema compilers and guidelines for creating a DMCL module are described in the *CA IDMS Database Administration Guide*.

The IPSB Compiler

What the IPSB Compiler Does

The IPSB compiler, consisting of the IDMSDLMG module, accepts user-supplied input statements and subschema tables as input. Compiler output consists of the IPSBs and a listing of any diagnostic messages. The resulting IPSBs are known as fixed IPSBs. At runtime, when CA IDMS DLI Transparency processes a DL/I application, the back end of the runtime interface loads the relevant fixed IPSB. The fixed IPSB then serves the back end as the source for creating a variable IPSB. A variable IPSB keeps track of the DL/I and CA IDMS/DB information during CA IDMS DLI Transparency processing.

IPSB Must Be in Load (Core-Image) Library at Runtime

The fixed IPSB must be available in a load (core-image) library at runtime. Hence, the user must run the IPSB compiler to create the appropriate IPSBs before using the runtime interface. Once an IPSB is compiled, assembled, and link edited, however, it is available for use during all subsequent executions of the DL/I application program (assuming no changes are made to the DL/I applications).

Runtime Interface

The runtime interface:

- Accepts retrieval and update requests from the DL/I application programs
- The interface then processes the requests into appropriate CA IDMS/DB requests and sends them to CA IDMS/DB
- After CA IDMS/DB processes the requests, the runtime interface accepts the retrieved data and status information from CA IDMS/DB for placement in a format acceptable to the DL/I application program
- To accomplish these functions, the runtime interface consists of special-purpose components, a front end, and a back end.

The special-purpose components, the front end, and the back end are discussed in the remainder of this appendix.

Special-Purpose Components

The CA IDMS DLI Transparency special-purpose components consist of the following modules and database procedures:

- **IDMSDLRC module** –– IDMSDLRC is the module used in place of the DL/I region controller.
- **IDMSDLLI module** –– IDMSDLLI (the CA IDMS/DLI Transparency language interface used in place of native DL/I language interfaces) is for batch call-level DL/I application programs only.
- **IDMSDLHI module** –– IDMSDLHI is the CA IDMS DLI Transparency language interface used for batch command-level DL/I (EXEC DLI) applications.
- **IDMSDL1C module** –– IDMSDL1C is the language interface used with CICS call-level DL/I applications in z/OS.
- **IDMSDL1V module** –– IDMSDL1V is the CA IDMS DLI Transparency language interface used for CICS call-level DL/I applications in z/VSE.
- **IDMSDLHC module** –– IDMSDLHC is the language interface used for CICS COBOL command-level DL/I (EXEC DLI) applications in z/OS.
- **IDMSDLCV module** –– IDMSDLCV is the language interface used for CICS COBOL command-level DL/I (EXEC DLI) applications in z/VSE.
- **IDMSDLHP module** –– IDMSDLHP is the language interface used for CICS PL/I command-level DL/I (EXEC DLI) applications in z/OS.
- **IDMSDLPV module** –– IDMSDLPV is the language interface used for CICS PL/I command-level DL/I (EXEC DLI) applications in z/VSE.
- **IDMSDLHA module** –– IDMSDLHA is the language interface used for CICS Assembler command-level DL/I (EXEC DLI) applications in z/OS.
- **IDMSDLAV module** –– IDMSDLAV is the CA IDMS DLI Transparency CICS Assembler command-level DL/I (EXEC DLI) applications in z/VSE.
- **IDMSDLVC database procedure** –– IDMSDLVC is a system-provided database procedure for modifying variable-length records.
- **IDMSDLVD database procedure** –– IDMSDLVD is a system-provided database procedure for retrieving variable-length records.

Each of the above components is discussed below. Additionally, diagrams are provided to illustrate the relationship among the components at runtime in both a batch and CICS environment.

Runtime Components in a Batch Environment

In a batch environment, CA IDMS DLI Transparency processing of a DL/I application program:

- Begins in the IDMSDLRC module. The IDMSDLRC module's functions include
  - Issuing a call to the front-end (IDMSDLFE) module
  - Loading the DL/I application program and passing control to the DL/I application program
- From the DL/I application program, call-level DL/I calls are passed to the language interface, IDMSDLLI. EXEC DLI type commands are passed to the command-level language interface IDMSDLHI.
- The language interface transfers control to the IDMSDLFE module
- IDMSDLFE issues, as appropriate, a BIND RUN-UNIT or a FINISH, or sends the DL/I call to RHDCDLBE
- RHDCDLBE then converts the DL/I call to the appropriate CA IDMS/DB request

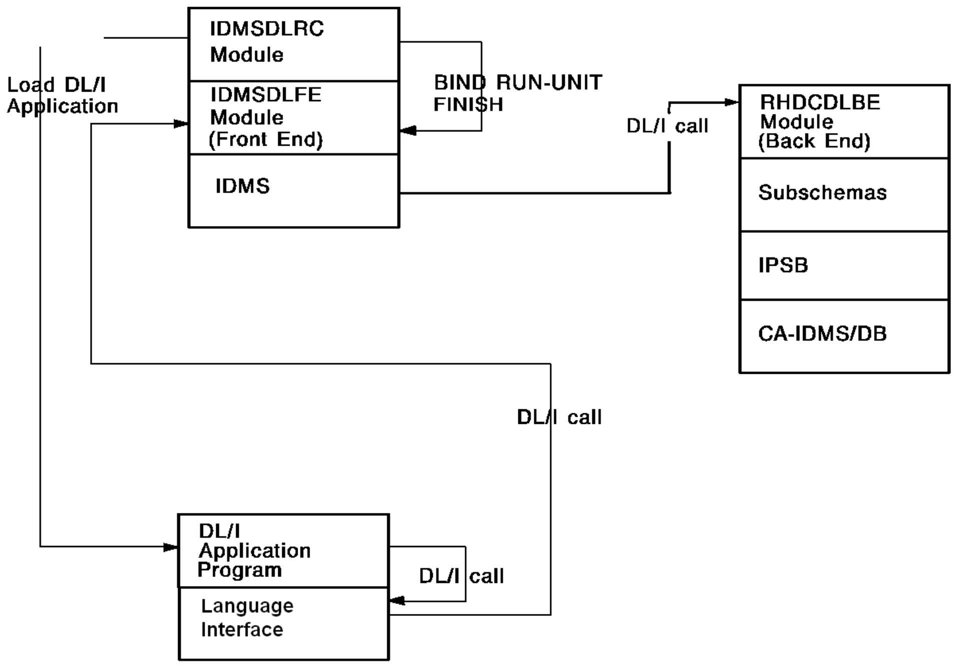


Figure 71. CA IDMS DLI Transparency runtime components in a batch environment

Components in a CICS Runtime Environment

- The DL/I application program issues a DL/I call through the language interface. The language interface locates the address of the IDMSDLFC module in the CICS common workarea (CWA) and passes control to IDMSDLFC.

- IDMSDLFC is part of the IDMSINTC module created for CA IDMS DLI Transparency (see Section 5, "CA IDMS DLI Transparency Runtime Environment") and is the CA IDMS DLI Transparency's equivalent of native DL/I's online nucleus. At runtime, IDMSDLFC validates the call and control is passed to the IDMSDLFE module (the front end).
- IDMSDLFE issues a BIND RUN-UNIT or FINISH, or sends the DL/I call information to RHDCDLBE (the back end).
- RHDCDLBE converts the DL/I call to the appropriate CA IDMS/DB request.

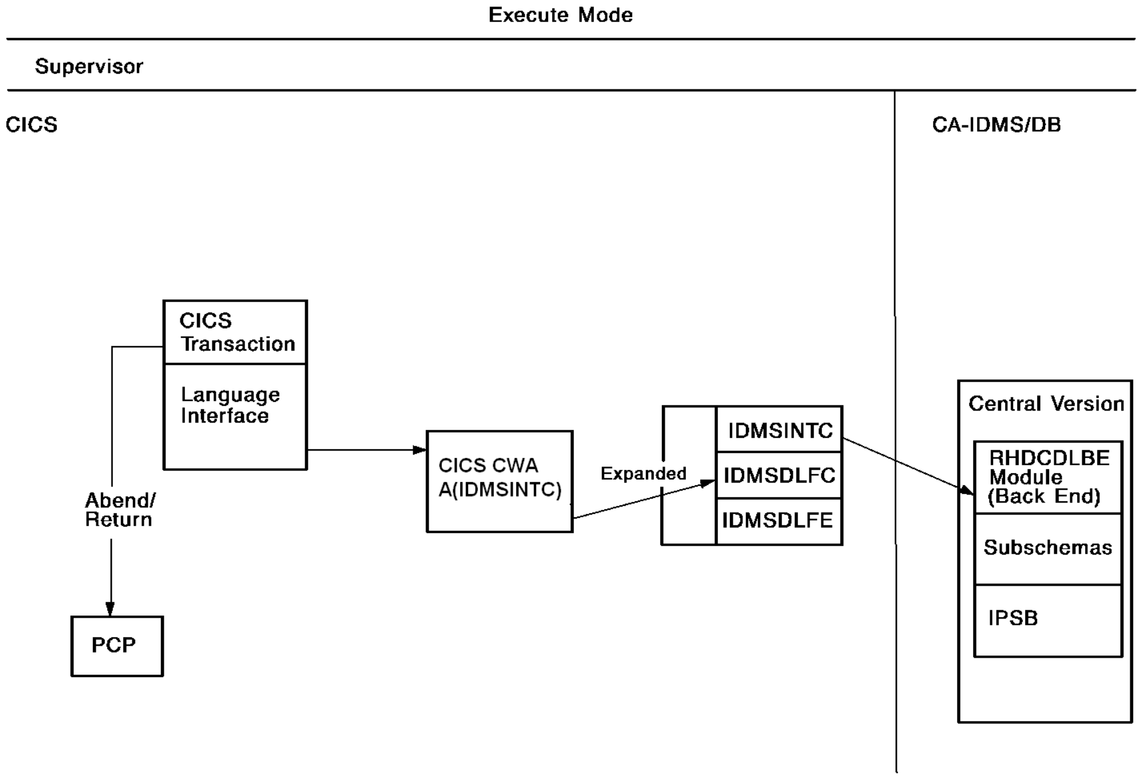


Figure 72. CA IDMS DLI Transparency components in a CICS environment at runtime

IDMSDLRC module

The IDMSDLRC module is the replacement for the DL/I region controller. In DL/I, the operating system executes a region controller and the region controller loads and passes control to the DL/I application program. IDMSDLRC performs the following functions:

- Accepts from the JCL the user-specified parameters. In z/OS ,these parameters are specified in the JCL in the PARM clause of the EXEC statement; in z/VSE, they are specified in the JCL in the SYSIPT file. The parameters identify the DL/I application program to be processed and the IPSB that is to be accessed at runtime.
- Issues a call to the front end (IDMSDLFE), requesting the front end to issue a BIND RUN-UNIT. Along with this call, IDMSDLRC provides the front end with the name of the IPSB to be used at run time.
- Receives the addresses of the PCBs used by the DL/I application program.
- Loads and passes control to the DL/I application program. As IDMSDLRC passes control, it provides the DL/I application with the PCB parameter list.
- Issues a termination call to IDMSDLFE after the DL/I application has executed. This call requests the front end to issue a FINISH.

IDMSDLLI module

The IDMSDLLI module is used for batch DL/I application programs only. This module replaces the following DL/I language interfaces:

- Native DL/I COBOL language interface (CBLTDLI)
- Native DL/I PL/I language interface (PLITDLI)
- Native DL/I Assembler language interface (ASMTDLI)

At runtime, the CA IDMS DLI Transparency user link edits IDMSDLLI to each DL/I application program to be processed by CA IDMS DLI Transparency. When link edited to the DL/I application, the IDMSDLLI performs the following functions:

- Receives control on a DL/I call from the DL/I application program
- Reformats the call parameter list and sends the list to the front end
- Passes control to the front end, which establishes, controls, and terminates communication with the back end

  Note: In XA environments, if the COBOL DYNAMIC link-edit option is used, and the DL/I application program does not run in XA mode, relink module IDMSDLLI with RMODE=24.

IDMSDL1C module

IDMSDL1C is for use only under z/OS CICS for call-level DL/I applications. This module replaces the DL/I application interface resolving the entry points CBLTDLI, ASMTDLI, and PLITDLI.

IDMSDL1V module

IDMSDL1V is for use only under z/VSE CICS for call-level DL/I applications. This module replaces the DL/I application interface resolving the entry points CBLTDLI, ASMTDLI, and PLITDLI.

IDMSDLHI module

IDMSDLHI is for use with batch command-level (EXEC DLI) COBOL and PL/I programs in z/OS. This module replaces modules DFSLICBL, DFSLIPLI. IDMSDLHI must be ordered first in the link edit with the application program.

IDMSDLHC module

IDMSDLHC is for use with CICS command-level (EXEC DLI) COBOL programs in z/OS. This module replaces module DFHECI. IDMSDLHC must be ordered first in the link edit with the application program.

IDMSDLCV module

IDMSDLCV is for use with CICS command-level (EXEC DLI) COBOL programs in z/VSE. This module replaces module DFHECI. IDMSDLCV must be ordered first in the link edit with the application program.

IDMSDLHP module

IDMSDLHP is for use with CICS command-level (EXEC DLI) PL/I programs in z/OS. This module replaces module DFHEPI. IDMSDLHP must be ordered first in the link edit with the application program.

IDMSDLPV module

IDMSDLPV is for use with CICS command-level (EXEC DLI) PL/I programs in z/VSE. This module replaces module DFHPL1I. IDMSDLPV must be ordered first in the link edit with the application program.

IDMSDLHA module

IDMSDLHA is for use with CICS command-level (EXEC DLI) Assembler programs in z/OS. This module replaces module DFHEAI. IDMSDLHA must be ordered first in the link edit with the application program.

IDMSDLAV module

IDMSDLAV is for use with CICS command-level (EXEC DLI) Assembler programs in z/VSE. This module replaces module DFHEAI. IDMSDLAV must be ordered first in the link edit with the application program.

IDMSDLVC database procedure

IDMSDLVC is a database procedure provided with CA IDMS DLI Transparency for modifying variable-length records that correspond to variable-length segments. Before a variable-length record is modified, IDMSDLVC is called to maintain the length of the CA IDMS/DB variable-length record. IDMSDLVC is specified in a CALL sentence as part of the RECORD DESCRIPTION in the schema (see the *CA IDMS Database Administration Guide*).

IDMSDLVD database procedure

IDMSDLVD is a database procedure provided with CA IDMS DLI Transparency for retrieving variable-length records that correspond to variable-length segments. Before a variable-length record is retrieved, IDMSDLVD is called to maintain the length of the CA IDMS/DB variable-length record. IDMSDLVD is specified in a CALL sentence as part of the RECORD DESCRIPTION in the schema (see the *CA IDMS Database Administration Guide*).

CA IDMS DLI Transparency Front End

The CA IDMS DLI Transparency front-end components consist of the IDMSDLFE module and, if CA IDMS DLI Transparency is used under CICS, the IDMSDLFC module.

IDMSDLFE module

The IDMSDLFE module establishes, controls, and terminates communication with the back end (the RHDCDLBE module). When IDMSDLFE receives an initialization call from the IDMSDLRC module in a batch environment or from IDMSDLFC under CICS (see below), it performs the following functions:

- Acquires work area
- Issues a BIND RUN-UNIT to the back end (RHDCDLBE)
- Issues a call to RHDCDLBE for PCB information

Once the initialization functions are complete, IDMSDLFE accepts DL/I calls from the language interface and performs the following functions:

- Sends the DL/I calls to the back end.
- Accepts the retrieved data and status information from the back end.
- Receives from the back end the updated PCB control blocks, which are used to return retrieved data and status information to the DL/I application. When the DL/I application finishes executing, the front end receives a termination call from the region controller and performs the following:
  - Issues a FINISH to the back end
  - Frees storage

IDMSDLFC module

The IDMSDLFC module is a component in the reassembled IDMSINTC macro (see CA IDMS DLI Transparency Run-Time Environment (see page 155)). Used only under CICS, IDMSDLFC is initialized by a special signon transaction and performs the following functions:

- Linking a CICS DL/I application with IDMSDL1C or any other CA IDMS DL/I Transparency language interface establishes the intent to use CA IDMS DLI Transparency (see CA IDMS DLI Transparency Run-Time Environment (see page 155)).
- Receives control on a DL/I call from the DL/I application program.
- Reformats the call parameter list and sends the list to the front end.
- Passes control to the front end, which establishes, controls, and terminates communication with the back end.

CA IDMS DLI Transparency Back End

The back end consists of the RHDCDLBE module The back end processing is initiated by a BIND RUN-UNIT issued by IDMSDLFE. The back end performs the following functions during initiation of the run unit:

- Loads the appropriate fixed IPSB
- Acquires storage
- Acquires PCB information from the IPSB and then uses the information to build the PCBs

Once the run unit is initiated, the back end performs the following functions for it:

- Issues appropriate CA IDMS/DB calls to service DL/I requests
- Accepts from CA IDMS/DB retrieved data and/or status information
- Sends retrieved data and/or status information to the front end

After the DL/I application program has executed, RHDCDLBE receives a FINISH from either the front end (in batch processing) or IDMSINTC (in CICS processing) and terminates processing.

The Load Utility

The load utility consists of the IDMSDLLD module. It accepts data unloaded from a DL/I database (via IBM's HD unload utility) and stores it in a CA IDMS/DB database. The CA IDMS/DB database must be prepared and initialized before running the load utility.

To execute, the load utility also requires:

- **An IPSB load module.** The IPSB translates the DL/I segment and data structure definitions to equivalent CA IDMS/DB record and set definitions. The load utility uses the DL/I-to-CA IDMS/DB equivalencies when storing the data in the CA IDMS/DB database. The IPSB definition must reflect the special considerations for a load IPSB (IPSB used with the load utility).
- **CA IDMS/DB schema, subschema, and DMCL modules**. The CA IDMS/DB modules constitute the runtime environment for the CA IDMS/DB database.

The process of loading the DL/I data can involve up to six steps. If the DL/I data does not include logical relationships, the only step required is the actual database load (Step 2). The steps in the load process are:

1. **Preload CALC processing** — Calculates CA IDMS/DB preload database pages for DL/I root segments to speed up the actual load (Step 2). Included in this step is a sort of the preload CALC data.
2. **Database load** — Stores the DL/I data in the CA IDMS/DB database. If logical relationships are found, the load utility writes the logical child records and their related logical parents to a workfile for additional processing. If the DL/I data comes from multiple databases (DBDs), a separate workfile is produced for each source database.
3. **Workfile sort/merge** — Merges multiple workfiles from Step 2 and sorts the resulting file to arrange logical child records under their logical parents.
4. **Prefix (concatenated key) resolution** — Processes the sorted workfile and generates correct prefixes (concatenated keys) for the logical child records.

5. **Workfile hierarchical sort** –– Sorts the workfile with resolved prefixes so that the logical child records are in their original DBD hierarchical sequences.
6. **Prefix update** –– Updates the logical child records in the CA IDMS/DB database with the generated prefixes. The prefixes are needed to establish the CA IDMS/DB set pointers for the logical child (member) sets and their logical parent (owner) sets.

Only Steps 1, 2, 4, and 6 invoke the IDMSDLLD module. Steps 3 and 5 (the sorts) take place outside of the load utility and CA IDMS DLI Transparency. They require use of the user's native sort/merge facility.

The IDMSDLLD Steps 1, 2, 4, and 6 produce reports that show the results of the processing and a count of the records involved.

Appendix C: Index Suppression Exit Support

This section contains the following topics:

About This Appendix

This appendix describes how to use the index suppression exit.

Index Suppression Exit Support

Use Your Own Index Suppression Exit Routine

CA IDMS DLI Transparency allows you to write your own index suppression exit routines for use with DL/I sparse indexes. If you have a DL/I secondary index, you can specify the exit routine so that it receives control immediately before the pointer records are stored in the secondary index. The exit routine can then indicate to CA IDMS DLI Transparency whether to process or ignore the store request.

How to Define and Exit Routine

To define an exit routine to CA IDMS DLI Transparency, specify the name of the routine for the EXIT ROUTINE parameter on the INDEX statement in the IPSB INDEX SECTION (described in IPSB Compiler (see page 93)). The name of the routine must match the name specified for the EXTRTN parameter on the XDFLD statement in the DL/I DBD definition. Note that the syntax generator will generate a corresponding EXIT ROUTINE in the IPSB source for each EXTRTN parameter it finds in the DL/I DBD definitions.

Run Time Operation

When the Exit Routine is Invoked

At program run time, the exit routine comes into play when the DL/I application issues an ISRT (insert) or REPL (replace) call for a CA IDMS/DB record that has been defined as an index source record in the INDEX SECTION of the active IPSB. When CA IDMS DLI Transparency encounters the ISRT or REPL call, it attempts to load the exit routine. To make sure the exit routine is available to CA IDMS DLI Transparency, you must place it in an operating system partitioned data set that can be accessed via a CDMSLIB JCL statement. An unsuccessful load of the routine will result in a PCB error status of NX.

ISRT Call

For an ISRT call, CA IDMS DLI Transparency determines whether the record to be stored participates in an index relationship as the index source record. If CA IDMS DLI Transparency finds such a relationship, it builds a suitable index pointer record. After checking for null value criteria, CA IDMS DLI Transparency calls the exit routine specified in the IPSB and passes control to it. It is the responsibility of the routine to determine whether the index pointer record should be stored or suppressed. The routine indicates its decision via a return code in register 15.

REPL Call

For a REPL call, the same process occurs as for an ISRT. The only difference is that prior to storing or suppressing an index pointer record CA IDMS DLI Transparency removes all existing index pointer records from the secondary index.

Interface

CA IDMS DLI Transparency expects an index exit routine to perform standard assembler linkage and provides a save area in register 13 for this purpose. Upon entry, the exit routine must save the contents of register 13. Upon return, it must restore the contents of registers 1 through 14. Under no circumstances should the routine alter data addressed by the registers at entry.

CA IDMS DLI Transparency initializes the registers to the following values:

- **Register 2** - Address of the index pointer record
- **Register 3** - Address of the index exit PARMS DSECT (described in figure 73 available further below)
- **Register 4** - Address of the index source record
- **Register 13** - Address of the save area
- **Register 14** - Return address in CA IDMS DLI Transparency
- **Register 15** - Address of the index exit entry point

The exit routine controls CA IDMS DLI Transparency's action by the return code it places in register 15, as follows:

- **4** —– Suppresses the index pointer record
- **0** —– Stores the index pointer record as part of the secondary index relationship

Figure 73 shows the format of the index exit PARMS DSECT (NDXXITDS DSECT), as passed to the exit routine.

```
                       NDXXITDS DSECT
  Offset    Field Name        Type/           Description
                                    Length
       0    NDXRECNM     DS    CL8       Index pointer record name
       8    NDXFLDNM     DS    CL8       Index definition field name
      16    NDXXITNM     DS    CL8       Index exit name
      24    NDXXITEP     DS    A          Index exit entry point
```

Figure 73. Index Exit PARMS DSECT

Appendix D: CA IDMS DLI Transparency JCL

This section contains the following topics:

About This Chapter

This appendix presents all of the JCL required for:

- The syntax generator
- The IPSB compiler
- The run-time interface
- The load utility

Note: z/VSE JCL is presented using UPSI. z/VSE users can optionally use a SYSCTL statement or utilize a SYSIDMS parameter at runtime. In some cases, having SYSIDMS parameters in inline JCL (SYSIPT) may produce undesirable results due to application parameter usage. In such cases, SYSIDMS should be implemented as a DATASET. Otherwise, SYSIDMS parameters should be placed before the DL/I SYSIPT parameter information. For more information about all SYSIDMS parameters, see the *CA IDMS Common Facilities Guide*.

Syntax Generator JCL

Assemble a PSB

The JCL to assemble a PSB for use when generating IPSB source statements is shown below:

PSB (z/OS)

```
//JOB
//ASM  EXEC PGM=ASMA90
//SYSPRINT DD SYSOUT=A
//SYSLIB DD DSN=yourHLQ.CAGJSRC,DISP=SHR
//SYSUT1 DD DSN=&&SYSUT1,UNIT=disk,SPACE=(1700,(600,100))
//SYSUT2 DD DSN=&&SYSUT2,UNIT=disk,SPACE=(1700,(300,50))
//SYSUT3 DD DSN=&&SYSUT3,UNIT=disk,SPACE=(1700,(30,50))
//SYSPUNCH  DD DUMMY
//SYSGO  DD DSN=&&OBJSET,UNIT=SYSDA,SPACE=(80,(200,50)),
    DISP=(MOD,PASS)
//SYSIN  DD  *
Insert PSB source code here.
/*
//SYSIN  DD  *
//LINK   EXEC PGM=HEWL
//SYSPRINT  DD  SYSOUT=A
//SYSUT1 DD DSN=&&SYSUT1,UNIT=disk,SPACE=(1024,(50,20))
//SYSLMOD   DD  DISP=SHR,DSN=user.loadlib(psbname)
//SYSLIN DD DSN=&&OBJSET,DISP=(OLD,DELETE)
//       DD  DDNAME=SYSIN
//
```

yourHLQ.CAGJSRC	data set name of the CA IDMS/DB source library
disk	symbolic device type for a disk file
psbname	member name of the PSB
user.loadlib	data set name of the load library that is to contain the resulting assembled PSB

PSB (z/VSE)

```
// JOB
// LIBDEF *, SEARCH=idms.library
// LIBDEF *, CATALOG=user.library
// OPTION CATAL
   PHASE psbname,*
// EXEC  ASSEMBLY
insert PSB source code here
/*
// EXEC  LNKEDT
```

idms.library	name of the CA IDMS/DB source library
user.library	name of the library that is to contain the resulting assembled PSB
psbname	name of the PSB source statements

Assemble DBDs

The JCL to assemble DBDs for use when generating IPSB source statements is shown below:

DBD (z/OS)

```
//JOB
//ASM       EXEC  PGM=ASMA90
//SYSPRINT  DD  SYSOUT=A
//SYSLIB    DD DSN=yourHLQ.CAGJMAC,DISP=SHR
//SYSUT1    DD DSN=&&SYSUT1,UNIT=disk,SPACE=(1700,(600,100))
//SYSUT2    DD DSN=&&SYSUT2,UNIT=disk,SPACE=(1700,(300,50))
//SYSUT3    DD DSN=&&SYSUT3,UNIT=disk,SPACE=(1700,(30,50))
//SYSPUNCH  DD DUMMY
//SYSGO     DD DSN=&&OBJSET,UNIT=SYSDA,SPACE=(80,(200,50)),DISP=(MOD,PASS)
//SYSIN     DD  *
DBD source code
/*
//LINK      EXEC  PGM=HEWL
//SYSPRINT  DD  SYSOUT=A
//SYSUT1 DD DSN=&&SYSUT1,UNIT=disk,SPACE=(1024,(50,20))
//SYSLMOD   DD  DISP=SHR,DSN=user.loadlib(dbdname)
//SYSLIN DD DSN=&&OBJSET,DISP=(OLD,DELETE)
//
```

yourHLQ.CAGJMAC	data set name of the CA IDMS/DB macro library

dbdname	member name of the DBD
disk	symbolic device type for a disk file
user.loadlib	data set name of the load library that is to contain the resulting assembled DBD

DBD (z/VSE)

```
// JOB
// LIBDEF *,SEARCH=idms.library'
// LIBDEF *,CATALOG=user.library
// OPTION CATAL
   PHASE dbdname,*
// EXEC  ASSEMBLY
insert DBD source code here
/*
// EXEC  LNKEDT
```

idms.library	name of the CA IDMS/DB source library
user.library	name of the library that is to contain the resulting assembled DBD
dbdname	name of the DBD source statements

Execute the Syntax Generator

The JCL to execute the syntax generator is shown below:

SYNTAX GENERATOR (z/OS)

```
//JOB
//IPSBGEN   EXEC  PGM=IDMSDLPG
//STEPLIB   DD DISP=SHR,DSN=idms.loadlib
//        DD DISP=SHR,DSN=user.loadlib
//SYSLST DD  SYSOUT=A
//SYSPCH DD DSN=user.syntax,DISP=(NEW,CATLG),SPACE=(TRK,5),
    DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB)
//SYSIPT DD  *
compiler-directive statements
generator statements
/*
```

idms.loadlib	data set name of the CA IDMS/DB load library
user.loadlib	data set name of the load library that contains the assembled PSB and DBDs

user.syntax	data set name for the file that is to contain the resulting source statements

SYNTAX GENERATOR (z/VSE)

```
// JOB
// DLBL IDMSPCH,'idms.user.syntax'
// EXTENT SYS016,nnnnnn
// LIBDEF *,SEARCH=user.library
// EXEC IDMSDLPG
compiler-directive statements
generator statements
/*
/&
```

idms.user.syntax	name of the source library that is to contain the generated SCHEMA, SUBSCHEMA, DMCL or ISPSB source statements
nnnnnn	volume serial identifier
user.library	name of the library that contains the assembled DBDs/PSBs

IPSB Compiler JCL

The JCL necessary to execute the IPSB compiler to assemble and link edit the output is shown below:

IPSB COMPILER (z/OS)

```
//DLMG   EXEC  PGM=IDMSDLMG
//STEPLIB DD  DSN=idms.loadlib,DISP=SHR
//SYSLST DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSPCH DD  DSN=&&SYSPCH,UNIT=disk,SPACE=(4000,(100,50))
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=4000),DISP=(NEW,PASS)
//SYSIPT DD  *
ipsb input statements
/*
//ASM  EXEC  PGM=ASMA90
//SYSPRINT DD  SYSOUT=A
//SYSLIB DD  DSN=yourHLQ.CAGJMAC,DISP=SHR
//SYSUT1 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT2 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT3 DD UNIT=disk,SPACE=(cyl,(2,2))
//SYSPUNCH DD DSN=&&IPSB,UNIT=disk,DISP=(NEW,PASS),
//    SPACE=(80,(400,40))
//SYSIN  DD  DSN=&&SYSPCH,DISP=(OLD,DELETE)
//LINK      EXEC  PGM=HEWL
//SYSPRINT      SYSOUT=A
//SYSLIN DD  DSN=&&IPSB,DISP=(OLD,DELETE)
//SYSUT1 DD  UNIT=disk,SPACE=(trk,(20,5))
//SYSLMOD DD  DSN=idms.loadlib(ipsb),DISP=SHR
```

idms.loadlib	data set name of the CA IDMS/DB load library containing the subschema description and IDMSDLMG
cyl,(2,2)	space to be allocated in bytes per cylinders
disk	symbolic device type for the disk file
&&IPSB	temporary data set containing the output from the assembly step
yourHLQ.CAGJMAC	data set name of the macro library
&&SYSPCH	temporary data set containing the output from IPSB compiler (IDMSDLMG)
trk,(20,5)	space to be allocated in bytes per tracks
4000,(100,50)	space to be allocated in bytes per blocks
80,(400,40)	space to be allocated in bytes per blocks

IPSB COMPILER (z/VSE)

```
// JOB
// LIBDEF *,SEARCH=idms.library
// LIBDEF *,CATALOG=ipsb.library
// DLBL IJSYSPH,'===.compiler.output',0
// EXTENT SYSPCH,nnnnnn,1,,ssss,llll
// ASSGN SYSPCH,DISK,VOL=nnnnnn,SHR
// EXEC IDMSDLMG
insert IPSB source statements here
/*
CLOSE SYSPCH,PUNCH
/*
// DLBL  IJSYSIN,'===.compiler.output',0
// EXTENT SYSIPT,nnnnnn,1,,ssss,llll
   ASSGN  SYSIPT,DISK,PERM,VOL=TECHD1,SHR
// OPTION DECK,NOEDECK,LIST,NORLD,NOXREF
// EXEC  ASSEMBLY
/*
CLOSE SYSIPT,SYSRDR
CLOSE SYSPCH,OOD
/*
DLBL IJSYSIN,'===.assembler.output',0:
// EXTENT SYSIPT,nnnnnn,1,,ssss,llll
   ASSGN SYSIPT,DISK,VOL=nnnnnn,SHR
// OPTION CATAL
   PHASE ipsbname,*
   INCLUDE
// EXEC LNKEDT
/*
   CLOSE SYSIPT,SYSRDR
/*
```

idms.library	name of the library
ipsb.library	name of the library that is to contain the compiled IPSB modules
ijsysin	file name of the input file to the linkage editor
ijsyspch	file name of the output file
llll	number of tracks required for the disk file
nnnnnn	volume serial number of the disk unit
ssss	relative starting track of the disk file
sysipt	logical-unit assignment of the input file to the linkage editor
syspch	logical-unit assignment of the output file

ipsbname	name of the IPSB runtime module

The JCL necessary to execute the CA IDMS DLI Transparency program definition table compiler (IDMSDLTG) and to assemble and link edit the DLPDTAB output is shown below:

PROGRAM DEFINITION TABLE COMPILER (z/OS)

```
//DLTG  EXEC  PGM=IDMSDLTG
//STEPLIB DD  DSN=idms.loadlib,DISP=SHR
//SYSLST DD  SYSOUT=A,DCB=BLKSIZE=133
//SYSPCH DD  DSN=&&SYSPCH,UNIT=disk,SPACE=(4000,(100,50))
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=4000),DISP=(NEW,PASS)
//SYSIPT DD  *
pdt input statements
/*
//ASM  EXEC  PGM=ASMA90
//SYSPRINT DD  SYSOUT=A
//SYSLIB DD  DSN=yourHLQ.CAGJMAC,DISP=SHR
//SYSUT1 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT2 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT3 DD UNIT=disk,SPACE=(cyl,(2,2))
//SYSPUNCH DD DSN=&&PDTB,UNIT=disk,DISP=(NEW,PASS),
//    SPACE=(80,(400,40))
//SYSIN  DD  DSN=&&SYSPCH,DISP=(OLD,DELETE)
//LINK      EXEC  PGM=HEWL
//SYSPRINT      SYSOUT=A
//SYSLIN DD  DSN=&&PDTB,DISP=(OLD,DELETE)
//SYSUT1 DD  UNIT=disk,SPACE=(trk,(20,5))
//SYSLMOD DD  DSN=idms.loadlib(DLPDTAB),DISP=SHR
```

idms.loadlib	data set name of the CA IDMS/DB load library containing the subschema description and IDMSDLTG
cyl,(2,2)	space to be allocated in bytes per cylinders
disk	symbolic device type for the disk file
&&PDTB	temporary data set containing the output from the assembly step
yourHLQ.CAGJMAC	data set name of the macro library
&&SYSPCH	temporary data set containing the output from program definition table compiler (IDMSDLTG)
trk,(20,5)	space to be allocated in bytes per tracks
4000,(100,50)	space to be allocated in bytes per blocks
80,(400,40)	space to be allocated in bytes per blocks

DLPDTAB	required link edit module name in the SYSLMOD statement.

PROGRAM DEFINITION TABLE COMPILER (z/VSE)

```
// JOB
// LIBDEF *,SEARCH=idms.library
// LIBDEF *,CATALOG=pdtb.library
// DLBL IJSYSPH,'===.compiler.output',0
// EXTENT SYSPCH,nnnnnn,1,,ssss,llll
// ASSGN SYSPCH,DISK,VOL=nnnnnn,SHR
// EXEC IDMSDLTG
insert PDT  source statements here
/*
CLOSE SYSPCH,PUNCH
/*
// DLBL  IJSYSIN,'===.compiler.output',0
// EXTENT SYSIPT,nnnnnn,1,,ssss,llll
   ASSGN  SYSIPT,DISK,PERM,VOL=TECHD1,SHR
// OPTION DECK,NOEDECK,LIST,NORLD,NOXREF
// EXEC  ASSEMBLY
/*
CLOSE SYSIPT,SYSRDR
CLOSE SYSPCH,OOD
/*
DLBL IJSYSIN,'===.assembler.output',0:
// EXTENT SYSIPT,nnnnnn,1,,ssss,llll
   ASSGN SYSIPT,DISK,VOL=nnnnnn,SHR
// OPTION CATAL
   PHASE pdtbname,*
   INCLUDE
// EXEC LNKEDT
/*
   CLOSE SYSIPT,SYSRDR
/*
```

idms.library	name of the library
pdtb.library	name of the library that is to contain the compiled PDT modules
ijsysin	file name of the input file to the linkage editor
ijsyspch	file name of the output file
llll	number of tracks required for the disk file
nnnnnn	volume serial number of the disk unit
ssss	relative starting track of the disk file
sysipt	logical-unit assignment of the input file to the linkage editor

syspch	logical-unit assignment of the output file
pdtbname	name of the PDT runtime module (DLPDTAB)

Run-Time Interface JCL

Link Edit Batch Call-Level DL/I Applications

To link edit the DL/I application program with the language application program with the language interface/ interface, the JCL for z/OS and for z/VSE is provided below:

IDMSDLLI (LINK EDIT) (z/OS)

```
//LINK   EXEC  PGM=HEWL
//SYSPRINT DD  SYSOUT=A
//IDMSLIB  DD  DSN=idms.loadlib,DISP=SHR
//SYSLIB  DD  DSN=user.loadlib,DISP=SHR
//SYSUT1  DD  UNIT=SYSDA,SPACE=(trk,(20,5))
//SYSLMOD  DD  DSN=user.loadlib,DISP=SHR
//SYSLIN  DD  *
INCLUDE IDMSLIB(IDMSDLLI)
INCLUDE SYSLIB(userpgm)
ENTRY DLITCBL     (or appropriate entry point name)
NAME userpgm(R)
/*
//
```

idms.loadlib	data set name of the IDMS object library
trk,(20,5)	space to be allocated in bytes per track
user.loadlib	data set name of the load library that is to contain the resulting linked user application program
userpgm	name of the DL/I application program to be link edited to IDMSDLLI

IDMSDLLI (LINK EDIT) (z/VSE)

```
//JOB
//LIBDEF *,SEARCH=(idms.library, user.library)
//LIBDEF *,CATALOG=user.library
//OPTION CATAL
  PHASE userpgm,*
  INCLUDE IDMSDLLI
  INCLUDE userpgm}
  ENTRY userpgm  }      Assembler programs
//EXEC  LNKEDT
/*
CLOSE SYSIPT,SYSRDR
/*
```

idms.library	data set name of the CA IDMS DLI Transparency
user.library	data set name of the library containing the DL/I application program object
user.library	name of the library that is to contain the resulting linked user's application program
userpgm	name of the DL/I application program in the user's object library

COBOL and PL/I Programs

COBOL and PL/I programs should add an INCLUDE statement and replace the ENTRY statement, as follows:

- COBOL:

  ```
  INCLUDE IDMSDLBC
  ENTRY CBLCALLA
  ```

- PL/I:

  ```
  INCLUDE IDMSDLBP
  ENTRY PLICALLB
  ```

Link Edit Batch Command-Level DL/I (EXEC DLI) Applications

To link edit the DL/I application program using EXEC DLI commands with the language application program with the language interface/ interface, the JCL for z/OS and for z/VSE is provided below:

IDMSDLHI (LINK EDIT) (z/OS)

```
//LINK   EXEC  PGM=HEWL
//SYSPRINT DD  SYSOUT=A
//IDMSLIB  DD  DSN=idms.loadlib,DISP=SHR
//SYSLIB  DD  DSN=user.loadlib,DISP=SHR
//SYSUT1  DD  UNIT=SYSDA,SPACE=(trk,(20,5))
//SYSLMOD  DD  DSN=user.loadlib,DISP=SHR
//SYSLIN  DD  *
INCLUDE IDMSLIB(IDMSDLHI)
INCLUDE SYSLIB(userpgm)
ENTRY DLITCBL     (or appropriate entry point name)
NAME userpgm(R)
/*
//
```

idms.loadlib	data set name of the IDMS object library
trk,(20,5)	space to be allocated in bytes per track
user.loadlib	data set name of the load library that is to contain the resulting linked user application program
userpgm	name of the DL/I application program to be link edited to IDMSDLHI

IDMSDLHI (LINK EDIT) (z/VSE)

```
//JOB

//LIBDEF *,SEARCH=(idms.library, user.library)
//LIBDEF *,CATALOG=user.library
//OPTION CATAL
 PHASE userpgm,*
 INCLUDE IDMSDLHI
 INCLUDE userpgm}
 ENTRY userpgm }    Assembler programs
//EXEC LNKEDT
/*
CLOSE SYSIPT,SYSRDR
/*
```

idms.library	data set name of the CA IDMS DLI Transparency
user.library	data set name of the library containing the DL/I application program object
user.library	name of the library that is to contain the resulting linked user's application program
userpgm	name of the DL/I application program in the user's object library

Execute DL/I Batch Application Program

The JCL to execute a DL/I batch application program is shown below: batch application program is shown below:

Central Version

EXECUTE BATCH APPLICATION (z/OS)

```
//DLI  EXEC  PGM=IDMSDLRC,PARM='DLI,userprog,ipsb'
//STEPLIB DD  DSN=idms.loadlib,DISP=SHR
//        DD  DSN=user.loadlib,DISP=SHR
//sysctl  DD  DSN=idms.sysctl,DISP=SHR
//SYSLST  DD  SYSOUT=A
//SYSIDMS DD *

Put SYSIDMS parameters here

DBNAME=database name or segment name
DMCL=DMCL name if other than default of IDMSDMCL
/*
//SYSIN  DD  *
any additional statements required to run DL/I application
program
/*
```

Note: The user can specify either DLI or DB in the PARM parameter. If ipsb and userprog have the same names, ipsb can be omitted. For more information about all SYSIDMS parameters, see the *CA IDMS Common Facilities Guide*.

idms.loadlib	data set name of the CA IDMS DLI Transparency load library
idms.sysctl	data set name of the SYSCTL file

ipsb	the name of the IPSB associated with the DL/I application program
sysctl	ddname of the SYSCTL file
user.loadlib	data set name of the load library containing the DL/I application program
userprog	the name of the DL/I application program

Local Mode

To execute the DL/I batch application program in local mode:

- Remove the SYSCTL statement.
- Replace the SYSCTL statement with the following:

```
//sysjrnl DD DSN=idms.tapejrnl,DISP=(NEW,KEEP),UNIT=tape
//userdb  DD  DSN=user.userdb,DISP=SHR
```

idms.tapejrnl	data set name of the tape journal file
sysjrnl	ddname of the tape journal file
tape	symbolic device type for the tape journal file
userdb	ddname of the user database
user.userdb	data set name of the user database

Central Version

EXECUTE BATCH APPLICATION (z/VSE)

Note: The following JCL is for use if IDMSDLRC includes IDMSDLPC in the linkedit.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// OPTION LOG
// DLBL  SYSCTL,'idms.sysctl',0,SD
// EXTENT SYS000,nnnnnn
// ASSGN SYS000,DISK,VOL=nnnnnn,SHR
// DLBL SYSIDMS,'#SYSIPT',0,SD
// EXEC IDMSDLRC
sysidms parameter statements
/*
DLI,userprog,ipsbname
/*

additional JCL as required to run DL/I application program
```

Note: The user can specify either DLI or DB. The user can omit ipsb if it has the same name as userprog.

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the library that contains the user's application program
idms.sysctl	name of the DL/I application program in the user's library
ipsbname	name of the IPSB (Interface PSB) that is used by the application program
nnnnnn	volume serial number
userprog	the name of the DL/I application program

Note: The following Run-Time interface JCL is for use if IDMSDLPC is not included in IDMSDLRC.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// OPTION LOG
// DLBL  SYSCTL,'idms.sysctl',0,SD
// EXTENT SYS000,nnnnnn
// ASSGN SYS000,DISK,VOL=nnnnnn,SHR
// DLBL SYSIDMS,'#SYSIPT',0,SD
// EXEC IDMSDLRC PARM='DLI,userprog,ipsbname'
sysidms parameter statements
/*

additional JCL as required to run DL/I application program
```

Note: The user can specify either DLI or DB. The user can omit ipsb if it has the same name as userprog.

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the library that contains the user's application program
idms.sysctl	name of the DL/I application program in the user's library
ipsbname	name of the IPSB (Interface PSB) that is used by the application program
nnnnnn	volume serial number
userprog	the name of the DL/I application program

Local Mode JCL

To execute the DL/I application program in local mode:

- Remove the UPSI statement.
- Insert the following after the ASSGN statement:

```
// TLBL  journal,'idms.tapejrnl'
// ASSGN sys009,X'ttt'
// DLBL  userdb,'user.userdb'
// EXTENT sys018,nnnnnn,1,ssss,llll
// ASSGN sys018,dddd,VOL=nnnnnn,SHR
```

idms.tape.jrnl	file-id of the tape journal
dddd	device assignment for the disk file
journal	filename of the tape journal
llll	number of tracks required for the disk file

nnnnnn	volume serial identifier of the appropriate disk volume
sys009	logical-unit assignment of the tape journal file
sys018	logical-unit assignment of the user database
ssss	relative starting track of the disk file
ttt	channel-unit assignment of the journal file
userdb	filename of the user database
user.userdb	file-id of the user database

Assemble IDMSDL1C For CICS Call-Level DL/I Usage (z/OS)

Use the following JCL to assemble IDMSDL1C:

IDMSDL1C (z/OS)

```
//      EXEC HLASMCL
//C.SYSLIB   DD DSN=cics.maclib,DISP=OLD
//                 DD DSN=yourHLQ.CAGJMAC,DISP=OLD
//C.SYSIN DD *
               COPY  #LREDS
               COPY  #OPIDS
IDMSDL1C CWADISP=nn
         END
/*
//L.SYSLMOD DD DSN=idms.loadlib,DISP=OLD
//L.SYSIN DD *
ENTRY IDMSDL1C
MODE AMODE(31),RMODE(24)
NAME  IDMSDL1C(R)
//
```

yourHLQ.CAGJMAC	Data set name of the IDMS macro library
cics.maclib	Data set name of the CICS macro library
idmsdl1c	Name of the IDMSDL1C module
idms.loadlib	Data set name of the CA IDMS load library containing CA IDMS system modules

Syntax

```
►►─ IDMSDL1C CWADISP=cwa-intc-address-displacement ─────────────────────►◄
```

Parameters

CWADISP=

Identifies the displacement within the CICS CWA of a fullword that holds the address of the IDMSINTC module.

cwa-intc-address-displacement

Specify the same value given to the CWADISP parameter of the CICSOPTS macro.

Note: When IDMSDL1C is link edited to the CICS DL/I application program, DFHEAI0 must be included in the linkage editor input (if not already included). Also ensure that entry point DFHEI1 has been resolved in this application link edit. For command-level programs entry point DFHEI1 is typically resolved in the language- dependent command-level interface module already present in the link edit. IDMSDL1C requires that entry points DFHEI1 and DFHEAI0 be resolved for successful operation.

Assemble IDMSDL1V For CICS Call-Level DL/I Usage (z/VSE)

The JCL to assemble IDMSDL1V in a z/VSE environment is shown below:

IDMSDL1V (z/VSE)

```
// JOB
// LIBDEF *,SEARCH=(idms.library, cics.library)
// OPTION CATAL,DECK
// EXEC ASSEMBLY
         COPY  #LREDS
         COPY  #OPIDS
         END
/*
```

Note: IDMSDL1V and the IDMS macros and copy books must be accessible from the assigned source-statement library.

cics.library	data set name of the IBM-supplied CICS library
idms.library	data set name of the CA IDMS DLI Transparency library
nn	CWADISP specifications corresponding to the IDMSINTC CWADISP

Syntax

```
►►── IDMSDL1V CWADISP=cwa-intc-address-displacement ─────────────────────────►◄
```

Parameters

CWADISP=

Identifies the displacement within the CICS CWA of a fullword that holds the address of the IDMSINTC module.

cwa-intc-address-displacement

Specify the same value given to the CWADISP parameter of the CICSOPTS macro.

Assemble Language Interfaces For Command-Level DL/I (EXEC DLI) Usage

Use the following JCL to assembe the language interfaces:

IDMSDLHC/IDMSDLHP /IDMSDLHA (z/OS)

```
//ASM   EXEC PGM=ASMA90
//SYSLIB DD DSN=cics.maclib,DISP=SHR
//       DD DSN=yourHLQ.CAGJMAC,DISP=SHR
//SYSUT1 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT2 DD  UNIT=disk,SPACE=(cyl,(2,2))
//SYSUT3 DD UNIT=disk,SPACE=(cyl,(2,2))
//SYSPUNCH DD DSN=&&syspch,UNIT=disk,DISP=(NEW,PASS),
//       SPACE=(80,(400,40))
//SYSIN  DD *
         IDMSDLHC CWADISP=nn      ← for COBOL applications, use this line only
         IDMSDLHP CWADISP=nn      ← for PL/I applications, use this line only
         IDMSDLHA CWADISP=nn      ← for ASM applications, use this line only
         END
/*
//LINK   EXEC PGM=HEWL
//SYSLMOD DD DSN=idms.loadlib,DISP=SHR
//SYSLIN DD DSN=&&syspch,UNIT=disk,DISP=(OLD,DELETE),
 ENTRY IDMSDLXX                   ← change to the particular interface used
 MODE AMODE(31),RMODE(ANY)
 NAME  IDMSDLXX(R)                ← change to the particular interface used
//
```

yourHLQ.CAGJMAC	Data set name of the IDMS macro library
cics.maclib	Data set name of the CICS macro library

idmsdlxx:	
IDMSDLHC	Name of the COBOL interface module
IDMSDLHP	Name of the PL/I interface module
IDMSDLHA	Name of the Assembler interface module
idms.loadlib	Data set name of the CA IDMS load library containing CA IDMS system modules

The JCL to assemble in a z/VSE environment is shown below:

IDMSDLCV/IDMSDLPV/IDMSDLAV (z/VSE)

```
// JOB
// LIBDEF *,SEARCH=(idms.library)
// OPTION CATAL,DECK
// EXEC ASSEMBLY
         IDMSDLCV CWADISP=nn        ← for COBOL applications, use this line only
         IDMSDLPV CWADISP=nn        ← for PL/I applications, use this line only
         IDMSDLAV CWADISP=nn        ← for ASM applications, use this line only
         END
/*
```

Note: The IDMS macros and copy books must be accessible from the assigned source-statement library. Only one of the interfaces listed above should be assembled at a time. Each interface is specific to a programming language.

idms.library	data set name of the CA IDMS DLI Transparency library
nn	CWADISP specifications corresponding to the IDMSINTC CWADISP

Parameters

CWADISP=

Identifies the displacement within the CICS CWA of a fullword that holds the address of the IDMSINTC module.

Load Utility JCL

Preload CALC Processing (Step 1)

The JCL to perform CALC processing and preload sorting on the unloaded DL/I data is shown below:

Preload CALC Processing (Step 1, Part 1) (z/OS)

```
//CALC   EXEC PGM=IDMSDLRC,PARM='CALC,IDMSDLLD,ipsbname'
//STEPLIB DD DSN=idms.loadlib,DISP=SHR
//        DD DSN=ipsb.loadlib,DISP=SHR
//SYSOUT DD SYSOUT=A
//SYSLST DD SYSOUT=A
//SYSPRINT  DD  SYSOUT=A
//SYS001 DD DSN=unloaded.dli.data,DISP=OLD
//SYS002 DD DSN=unsorted.dli.calc.data,
//    UNIT=TAPE,DISP=(NEW,KEEP)
//
//
```

idms.loadlib	data set name of the CA IDMS DLI Transparency load library
ipsb.loadlib	data set name of the IPSB load library
ipsbname	name of the IPSB load module
unloaded.dli.data	data set name of the unloaded DL/I data
unsorted.dli.calc.data	data set name of the unsorted DL/I CALC data

PreLoad CALC Processing (Step 1, Part 1) (z/VSE)

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// DLBL fileid,'idms.database',,DA
// EXTENT SYS018,nnnnnn
// ASSGN SYS018,DISK,VOL=nnnnnn,SHR
// TLBL  SYS001,'unloaded.dli.data'
// ASSGN SYS001,nnn
// TLBL  SYS002,'unsorted.dli.data'
// ASSGN SYS002,nnn
// EXEC IDMSDLRC
sysidms parameter statements
/*
CALC,IDMSDLLD,ipsbname
/*
DMCL=dmclname
/*
```

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the load library that contains the IPSB and SUBSCEHEMA modules.
fileid	DCML database file assignment
idms.database	name of the CA IDMS database file
nnnnnn	volume serial number of the disk unit
unloaded.dli.data	name of the tape data set that contains the HD UNLOAD DLI data
unsorted.dli.data	name of the tape data set that contains the CALC DLI data output
nnn	cuu address of the tape unit
ibsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')
dmclname	name of the CA IDMS DMCL module

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// DLBL fileid,'idms.database',,DA
// EXTENT SYS018,nnnnnn
// ASSGN SYS018,DISK,VOL=nnnnnn,SHR
// TLBL  SYS001,'unloaded.dli.data'
// ASSGN SYS001,nnn
// TLBL  SYS002,'unsorted.dli.data'
// ASSGN SYS002,nnn
// EXEC  IDMSDLRC,PARM='CALC,IDMSDLLD,ipsbname'
sysidms parameter statements
/*
DMCL=dmclname
/*
```

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the load library that contains the IPSB and SUBSCEHEMA modules.
fileid	DCML database file assignment
idms.database	name of the CA IDMS database file
nnnnnn	volume serial number of the disk unit
unloaded.dli.data	name of the tape data set that contains the HD UNLOAD DLI data
unsorted.dli.data	name of the tape data set that contains the CALC DLI data output
nnn	cuu address of the tape unit
ibsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')
dmclname	name of the CA IDMS DMCL module

PreLoad CALC Sort (Step 1, Part 2) (z/OS)

```
//SORT  EXEC  SORT
//SORTIN DD DSN=unsorted.calc.dli.data,DISP=OLD,UNIT=TAPE
//SORTOUT DD DSN=sorted.calc.dli.data,DISP=OLD,UNIT=TAPE
//SORTWK01 DD UNIT=DISK,SPACE=(CYL,(n),,CONTIG)
//SORTWKO2 DD UNIT=DISK,SPACE=(CYL,(n),,CONTIG)
//SORTWK03 DD UNIT=DISK,SPACE=(CYL,(n),,CONTIG)
//SYSIN  DD *
SORT  FIELDS=(20,4,BI,D,24,4,BI,A)
/*
//
```

n	number of cylinders for space allocation
sorted.calc.dli.data	data set name of the sorted DL/I CALC data
unsorted.calc.dli.data	data set name of the unloaded DL/I CALC data (from Step 1, Part 1)

Note: This step requires that you use your own sort/merge facility.

PreLoad CALC Sort (Step1, Part 2) (z/VSE)

```
// TLB  SORTIN1,'unsorted.dli.data',,SD
// ASSGN SYS001,nnn
// TLBL SORTOUT,'sorted.dli.data',,SD
// ASSGN SYS002,nnn
// DLBL SORTWK1,'work.file1',,SD
// EXTENT SYS003,nnnnnn,1,0,ssss,llll
// ASSGN SYS003,DISK,VOL=nnnnnn,SHR
// DLBL SORTWK2,'workfile2',,SD
// EXTENT SYS004,1,0,ssss,llll
// ASSGN SYS004,DISK,VOL=nnnnnn,SHR
// DLBL SORTWK3,'work.file3',,SD
// EXTENT SYS005,ERES00,1,0,ssss,llll
// ASSGN SYS005,DISK,VOL=nnnnnn,SHR
// EXEC SORT
   SORT FIELDS=(20,4,BI,D,24,4,BI,A),FILES=1,WORK=3
   RECORD TYPE=V
   INPFIL BLKSIZE=8000
   OUTFIL BLKSIZE=8000
/*
//
```

unsorted.dli.data	data set name of the file created by the CALC processing step
sorted.dli.data	data set name of the sorted workfile produced by this sort step

nnn	cuu address of the tape unit
nnnnnn	volume serial number of the disk unit
work.file1	file id of the 1st SORT work file
work.file2	file id of the 2nd SORT work file
work.file3	file id of the 3rd SORT work file
logical.workfile	name of the data set that will receive data concerning logical relationships
ssss	starting track in disk extent
llll	number of tracks required for the disk file

Database Load (Step 2)

The JCL to load the DL/I data in the CA IDMS/DB database is shown below:

Central Version

Database Load (Step 2) (z/OS)

```
//LOAD  EXEC PGM=IDMSDLRC,PARM='LOAD,IDMSDLLD,ipsbname'
//STEPLIB DD DSN=idms.loadlib,DISP=SHR
//        DD DSN=ipsb.loadlib,DISP=SHR
//sysctl DD DSN=idms.sysctl,DISP=SHR
//SYSOUT DD SYSOUT=A
//SYSLST DD SYSOUT=A
//SYSPRINT  DD SYSOUT=A
//SYS001 DD DSN=unloaded.dli.data,
//    UNIT=TAPE,VOL=SER=nnnnnn,DISP=OLD
//SYS003 DD DSN=step2.workfile,
//              UNIT=TAPE,DISP=(NEW,KEEP),
//    DCB=(RECFM=FB,LRECL=288,BLKSIZE=5760)
//
```

idms.loadlib	data set name of the CA IDMS DLI Transparency load library
ipsb.loadlib	data set name of the IPSB load library
idms.sysctl	data set name of the SYSCTL file
ipsbname	name of the IPSB load module
nnnnnn	volume serial identifier for the tape/disk volume
step2.workfile	logical workfile output by the load
sysctl	ddname of the SYSCTL file

unloaded.dli.data	data set name of the unloaded DL/I data

Local Mode

To execute the load process in local mode, remove the SYSCTL statement and replace with the following:

```
//dictdb DD DSN=idms.dictdb
//sysjrnl DD DSN=idms.tapejrnl,DISP=(NEW,KEEP),UNIT=tape
//userdb DD DSN=user.userdb,DISP=SHR
```

idms.dictdb	data set name of the data dictionary
idms.tapejrnl	data set name of the tape journal file
dictdb	ddname of the data dictionary
sysjrnl	ddname of the tape journal file
tape	symbolic device type for the tape journal file
user.userdb	data set name of the user database
userdb	ddname of the user database

Central Version

Database Load (Step 2) (z/VSE)

Note: Use this Load utility JCL if the IDMSDLPC is included in the IDMSDLRC linkedit.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// DLBL fileid,'idms.database',,DA
// EXTENT SYS018,nnnnnn
// ASSGN SYS018,DISK,VOL=nnnnnn,SHR
// TLBL SYS001,'sorted.dli.data'
// ASSGN SYS001,nnn
// TLBL  SYS003,'logical.workfile'
// ASSGN SYS003,nnn
// EXEC  IDMSDLRC
sysidms parameter statements
/*
LOAD,IDMSDLLD,ipsbname
/*
/DMCL=ipsbname
/*
```

idms.library	data set name of the DLI Transparency library

user.library	name of the library that contains the IPSB and SUBSCHEMA modules
fileid	DMCL database file assignment
idms.database	name of the CA IDMS database file
nnnnnn	volume serial number of the disk unit
sorted.dli.data	name of the tape data set that contains the sorted CALC DLI data
logical.workfile	name of the data set that will receive data concerning logical relationships
nnn	cuu address of the tape unit
ipsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')
dmclname	name of CA IDMS DMCL module

Note: This LOAD utility JCL is for use if IDMSDLPC is not included in IDMSDLRC.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// DLBL fileid,'idms.database',,DA
// EXTENT SYS018,nnnnnn
// ASSGN SYS018,DISK,VOL=nnnnnn,SHR
// TLBL SYS001,'sorted.dli.data'
// ASSGN SYS001,nnn
// TLBL  SYS003,'logical.workfile'
// ASSGN SYS003,nnn
// EXEC IDMSDLRC,PARM='LOAD,IDMSDLLD,ipsbname
sysidms parameter statements
/*
LOAD,IDMSDLLD,ipsbname
/*
/DMCL=dmclname
/*
```

idms.library	data set name of the DLI Transparency library
user.library	name of the library that contains the IPSB and SUBSCHEMA modules
fileid	DMCL database file assignment
idms.database	name of the CA IDMS database file
nnnnnn	volume serial number of the disk unit

sorted.dli.data	name of the tape data set that contains the sorted CALC DLI data
logical.workfile	name of the data set that will receive data concerning logical relationships
nnn	cuu address of the tape unit
ipsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')
dmclname	name of CA IDMS DMCL module

Local Mode JCL

To execute the load process in local mode, remove the UPSI statement and insert the following after the ASSGN statement:

```
// DLBL  dictdb,'idms.dictdb'
// EXTENT sys015,nnnnnn,1,,SSSS,LLLL
// ASSGN sys015,dddd,VOL=nnnnnn,SHR
// TLBL  journal,'idms.tapejrnl'
// ASSGN SYS009,X'ttt'
// DLBL  userdb,'user.userdb',,DA
// EXTENT sys018,nnnnnn,1,,SSSS,LLLL
// ASSGN sys018,dddd,VOL=nnnnnn,SHR
```

idms.dictdb	file-id of the data dictionary
idms.tapejrnl	data set name of the tape journal file
dddd	device assignment for the disk file
dictdb	filename of the data dictionary
journal	filename of the tape journal
nnnnnn	volume serial number
sys018	logical-unit assignment of the user database
sys015	logical-unit assignment of the data dictionary
ttt	channel-unit assignment of the journal file
user.userdb	file-id of the user database
userdb	name of the user database

Workfile Sort/Merge (Step 3)

The JCL to merge/sort the logical workfiles produced by the load step is shown below:

Workfile Sort/Merge (Step 3) (z/OS)

```
// SORT  EXEC  SORT
//SORTIN DD DSN=step2.workfile,DISP=OLD,UNIT=TAPE
//SORTOUT DD DSN=step3.workfile,DISP=OLD,UNIT=TAPE
//SORTWK01 DD UNIT=DISK,SPACE=(CYL,(n),,CONTIG)
//SORTWK02 DD UNIT=DISK,SPACE=(CYL,(n),,CONTIG)
//SORTWK03 DD UNIT=DISK,SPACE=(CYL,(n),,CONTIG)
//SYSIN  DD *
SORT  FIELDS=(25,5,BI,A)
/*
//
```

n	number of cylinders for space allocation
step2.workfile	the workfile output from Step 2
step3.workfile	the sorted workfile output by this step

Note: This step requires that you use your own sort/merge facility.

Workfile Sort/Merge (Step 3) (z/VSE)

```
// TLBL  SORTIN1,'logical.workfile'
// ASSGN  SYS001,nnn
// TLBL  SORTOUT,'sorted.workfile'
// ASSGN  SYS002,nnn
// DLBL  SORTWK1,'work.file1'
// EXTENT  SYS003,1,0,ssss,llll
// ASSGN  SYS003,DISK,VOL=nnnnnn,SHR
// DLBL  SORTWK2,'work.file2'
// EXTENT  SYS004,1,0,ssss,llll
// ASSGN  SYS004,DISK,VOL=nnnnnn,SHR
// DLBL  SORTWK3,'work.file3'
// EXTENT  SYS005,ERES00,1,,ssss,llll
// ASSGN  SYS005,DISK,VOL=nnnnnn,SHR
// EXEC SORT
   SORT FIELDS=(25,5,BI,A),FILES=1,WORK=3
   RECORD TYPE=F,LENGTH=288
   INPFIL BLKSIZE=32544
   OUTFIL BLKSIZE=32544
/*
```

logical.workfile	data set name of the logical workfile produced by LOAD processing
sorted.workfile	data set name of the sorted workfile produced by this SORT set
nnn	cuu address of the tape unit
nnnnnn	volume serial number of the disk unit
work.file1	file-id of the first SORT work file
work.file2	file id of the second SORT work file
work.file3	file id of the third SORT work file
ssss	starting track in disk extent
llll	number of tracks in disk extent

Prefix (Concatenated Key) Resolution (Step 4)

The JCL to resolve the prefixes (concatenated keys) for the logical records in the workfile from Step 3 is shown below:

Prefix (Concatenated Key) Resolution (Step 4) (z/OS)

```
//PFXR   EXEC   PGM=IDMSDLRC,PARM='PFXR,IDMSDLLD,ipsbname'
//STEPLIB DD DSN=idms.loadlib,DISP=SHR
//        DD DSN=ipsb.loadlib,DISP=SHR
//SYSOUT DD SYSOUT=A
//SYSLST DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYS004 DD DSN=step3.workfile,DISP=OLD,UNIT=TAPE
//SYS003 DD DSN=step4.workfile.DISP=OLD,UNIT=TAPE
//
```

idms.loadlib	data set name of the CA IDMS DLI Transparency load library
ipsb.loadlib	data set name of the IPSB load library
ipsbname	name of the IPSB load module
step3.workfile	sorted output from Step 3
step4.workfile	the workfile output by this step

Prefix (Concatenated Key) Resolution (Step 4) (z/VSE)

Note: For use if IDMSDLPC is included in the IDMSDLRC linkedit.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// TLBL SYS004,'sorted.workfile'
// ASSGN SYS004,nnn
// TLBL SYS003,'hierarchic.workfile'
// ASSGN SYS003,nnn
// EXEC  IDMSDLRC
sysidms parameter statements
PFXR,IDMSDLLD,ipsbname
/*
```

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the library that contains the IPSB and SUBSCHEMA modules
sorted.workfile	name of the tape data set that contains the output of the previous step's SORT
hierarchic.workfile	name of the tape data set that contains the output of this step's SORT
nnn	cuu address of the tape unit
ipsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')

Note: For use if IDMSDLRC does not include IDMSDLPC in the linkedit.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// TLBL SYS004,'sorted.workfile'
// ASSGN SYS004,nnn
// TLBL SYS003,'hierarchic.workfile'
// ASSGN SYS003,nnn
// EXEC IDMSDLRC,PARM='PFXR,IDMSDLLD,ipsbname
sysidms parameter statements
/*
```

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the library that contains the IPSB and SUBSCHEMA modules
sorted.workfile	name of the tape data set that contains the output of the previous step's SORT

hierarchic.workfile	name of the tape data set that contains the output of this step's SORT
nnn	cuu address of the tape unit
ipsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')

Workfile Hierarchical Sort (Step 5)

The JCL to hierarchically sort the workfile from Step 4 is shown below:

Workfile Hierarchical Sort (Step 5) (z/OS)

```
//SORT  EXEC  SORT
//SORTIN DD DSN=step4.workfile,DISP=OLD,UNIT=TAPE
//SORTOUT DD DSN=step5.workfile,DISP=OLD,UNIT=TAPE
//SORTWK01 DD UNIT=DISK,SPACE=(CYL,(1),,CONTIG)
//SORTWKO2 DD UNIT=DISK,SPACE=(CYL,(1),,CONTIG)
//SORTWK03 DD UNIT=DISK,SPACE=(CYL,(1),,CONTIG)
//SYSIN  DD *
SORT  FIELDS=(17,8,BI,A)
/*
//
```

Note: This step requires that you use your own sort/merge facility.

step4.workfile	the workfile output from Step 4
step5.workfile	hierarchically sorted workfile output by this step

Workfile Hierarchical Sort (Step 5) (z/VSE)

```
// TLBL SORTIN1,'hierarchic.workfile',,SD
// ASSGN  SYS001,nnn
// TLBL  SORTOUT,'final.workfile',,SD
// ASSGN  SYS002,nnn
// DLBL  SORTWK1,'work.file1',,SD
// EXTENT  SYS003,1,0,ssss,llll
// ASSGN  SYS003,DISK,VOL=nnnnnn,SHR
// DLBL  SORTWK2,'work.file2',,SD
// EXTENT  SYS004,1,0,ssss,llll
// ASSGN  SYS004,DISK,VOL=nnnnnn,SHR
// DLBL  SORTWK3,'work.file3'
// EXTENT  SYS005,ERES00,1,0,ssss,llll
// ASSGN  SYS005,DISK,VOL=nnnnnn,SHR
// EXEC SORT
   SORT FIELDS=(17,8,BI,A),FILES=1,WORK=3
   RECORD TYPE=F,LENGTH=288
   INPFIL BLKSIZE=32544
   OUTFIL BLKSIZE=32544
/*
```

hierarchic.workfile	data set name of the output from the PFXR step
final.workfile	data set name of the sorted workfile produced by this SORT step
nnn	cuu address of the tape unit
nnnnnn	volume serial number of the disk unit
work.file1	file id of the third SORT work file
work.file2	file id of the second SORT work file
work.file3	file id of the third SORT work file
ssss	starting track in disk extent
llll	number of tracks in disk extent

Prefix Update (Step 6)

The JCL to update the logical child database records with the resolved prefixes is shown below. This step uses the hierarchically sorted workfile from Step 5.

Central Version

Prefix Update (Step 6) (z/OS)

```
//PFXU   EXEC PGM=IDMSDLRC,PARM='PFXU,IDMSDLLD,ipsbname'
//STEPLIB DD DSN=idms.loadlib,DISP=SHR
//    DD DSN=ipsb.loadlib,DISP=SHR
//sysctl DD DSN=idms.sysctl,DISP=SHR
//SYSOUT DD SYSOUT=A
//SYSLST DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYS004 DD DSN=step5.workfile,DISP=OLD,UNIT=TAPE
//
```

idms.loadlib	data set name of the CA IDMS DLI Transparency load library
idms.sysctl	data set name of the SYSCTL file
ipsb.loadlib	data set name of the IPSB load library
ipsbname	name of the IPSB load module
step5.workfile	hierarchically sorted workfile from step 5
sysctl	ddname of the SYSCTL file

Local Mode JCL

To execute the prefix update process in local mode, remove the SYSCTL statement and replace with the following:

```
//dictdb DD DSN=idms.dictdb
//sysjrnlDD DSN=idms.tapejrnl,DISP=(NEW,KEEP),UNIT=tape
//userdb DD DSN=user.userdb,DISP=SHR
```

idms.dictdb	data set name of the data dictionary
idms.tapejrnl	data set name of the tape journal file
dictdb	ddname of the data dictionary
sysjrnl	ddname of the tape journal file
tape	symbolic device type for the tape journal file
user.userdb	data set name of the user database

userdb	ddname of the user database

Central Version

Prefix Update (Step 6) (z/VSE)

Note: Use the following LOAD utility if IDMSDLPC is included in the IDMSDLRC linkedit.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// DLBL fileid,'idms.database',,DA
// EXTENT SYS018,nnnnnn
// ASSGN SYS018,DISK,VOL,=nnnnnn,SHR
// TLBL  SYS004,'final.workfile'
// ASSGN SYS004,nnn
// EXEC  IDMSDLRC
system parameter statements
/*
PFXU,IDMSDLLD,ipsbname
/*
/&
```

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the library that contains the IPSB and SUBSCHEMA modules
fileid	DMCL database file assignment
idms.database	name of the CA IDMS database file
nnnnnn	volume serial number of the disk unit
final.workfile	name of the tape dataset that contains the previous step's sorted output
ipsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')

Note: The following LOAD utility JCL is for use if IDMSDLPC is not included in the IDMSDLRC linkedit.

```
// JOB
// LIBDEF *,SEARCH=(idms.library,user.library)
// DLBL fileid,'idms.database',,DA
// EXTENT SYS018,nnnnnn
// ASSGN SYS018,DISK,VOL,=nnnnnn,SHR
// TLBL  SYS004,'final.workfile'
// ASSGN SYS004,nnn
// EXEC IDMSDLRC,PARM='PFXU.IDMSDLLD,ipsbname
system parameter statements
/*
/&
```

idms.library	data set name of the CA IDMS DLI Transparency library
user.library	name of the library that contains the IPSB and SUBSCHEMA modules
fileid	DMCL database file assignment
idms.database	name of the CA IDMS database file
nnnnnn	volume serial number of the disk unit
final.workfile	name of the tape dataset that contains the previous step's sorted output
ipsbname	name of the LOAD IPSB (Interface PSB with processing options of 'LOAD')

Local Mode JCL

To execute the prefix update process in local mode, remove the UPSI statement and insert the following after the ASSGN statement:

```
// DLBL  dictdb,'idms.dictdb'
// EXTENT sys015,nnnnnn,1,,SSSS,LLLL
// ASSGN sys015,dddd,VOL=nnnnnn,SHR
// TLBL  journal,idms.tapejrnl'
// ASSGN SYS009,X'ttt'
// DLBL  userdb,'user.userdb',,DA
// EXTENT sys018,nnnnnn,1,,SSSS,LLLL
// ASSGN sys018,dddd,VOL=nnnnnn,SHR
```

idms.dictdb	file-id of the data dictionary
idms.tapejrnl	data set name of the tape journal file
dddd	device assignment for the disk file
dictdb	filename of the data dictionary

journal	filename of the tape journal
nnnnnn	volume serial number
sys015	logical-unit assignment of the data dictionary
sys018	logical-unit assignment of the user database
ttt	channel-unit assignment of the journal file
user.userdb	file-id of the user database
userdb	filename of the user database

IPSB Decompiler JCL

The JCL necessary to execute the IPSB decompiler is shown below:

IPSB Decompiler (z/OS)

```
//DECOMPIL EXEC PGM=IDMSDLID
//STEPLIB  DD  DSN=idms.loadlib,DISP=SHR
//SYSOUT   DD  SYSOUT=A
//SYSLST   DD  SYSOUT=A
//SYSPCH   DD  DSN=ipsb.source.library(ipsbname),DISP=OLD
//SYSPRINT DD  SYSOUT=A
//SYSIPT   DD  *
IPSB=ipsb-load-module-name
/*
//
```

idms.loadlib	data set name of the CA IDMS DLI Transparency load library
ipsb.loadlib	data set name of the IPSB load library
ipsb.source.library	data set name of the IPSB source library
IPSB=ipsb-load-module-name	identifies the IPSB (required)

IPSB Decompiler (z/VSE)

```
// JOB
// LIBDEF *,SEARCH=(idms.library,ipsb.library)
// DLBL ijsyspch 'ipsb.source'
// EXTENT syspch,nnnnnn,1,0,ssss,llll
// ASSGN syspch,x,'ddd'
// ASSGN syslst,x'00E'
// EXEC IDMSDLID
sysidms parameter statements
/*
IPSB=ipsbname
/&
```

idms.library	data set name of the CA IDMS DLI Transparency library
ipsb.library	name of the library that contains the IPSB load modules
ipsb.source	data set name of the IPSB source statements
ijsyspch	filename of the output file
nnnnnn	volume serial number of the disk unit
syspch	logical unit assignment of the output file
ddd	device assignment of the disk file
llll	number of tracks required for the disk file
ssss	relative starting track of the disk file
ipbsname	identifies the IPSB for decompiliation

Appendix E: CA IDMS DLI Transparency IPSB Decompiler

This section contains the following topics:

About This Appendix

CA IDMS DLI Transparency includes an IPSB decompiler that creates CA IDMS DLI Transparency IPSB source statements from CA IDMS DLI Transparency IPSB load modules.

This appendix describes how to use the IPSB decompiler.

Using the IPSB Decompiler

Follow these steps when using the IPSB decompiler:

1. Identify all IPSB load modules for decompilation.
2. Allocate a direct access data set to receive the newly created IPSB source.
3. Create appropriate JCL for IPSB decompilation (as described in CA IDMS DLI Transparency JCL (see page 257)).
4. Run the IPSB decompiler once for each IPSB load module to be decompiled.
5. Review SYSLST messages for each decompilation run to be sure the job was successful.

Note: Although the IPSB compiler requires the subschema load module, the decompiler does not.

IPSB Decompiler Run-Time Operations

IPSB Decompiler Functions

The IPSB decompiler performs the following functions:

- Reads SYSIPT for IPSB-directive control statement
- Accesses the IPSB named in the control statement
- Validates the identity of the IPSB
- Writes representative IPSB source statements to SYSPCH
- Writes informative messages to SYSLST

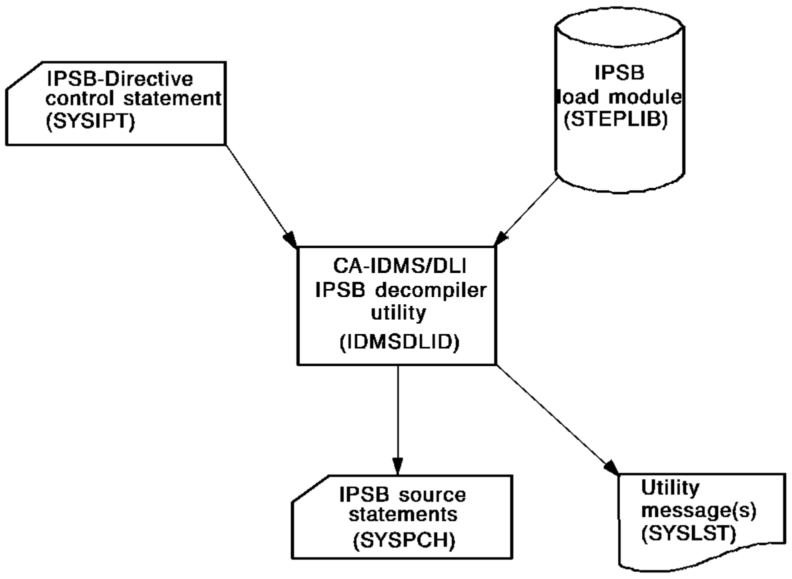

IPSB-Directive control statement (SYSIPT)
IPSB load module (STEPLIB)
CA-IDMS/DLI IPSB decompiler utility (IDMSDLID)
IPSB source statements (SYSPCH)
Utility message(s) (SYSLST)

Figure 74. Decompilation process

IPSB Decompiler Run-Time Considerations

To execute the decompiler:

- The IPSB load module to be decompiled must be available through use of a STEPLIB JCL statement.
- The utility control statement (IPSB-directive) must be supplied for input using a SYSIPT JCL statement.
- If the IPSB source statements are to be reviewed, the SYSPCH JCL statement should be directed to an output device.

- If the IPSB source statements are to be used for recompilation, The SYSPCH JCL statement should be directed to a direct access library suitable for containing IPSB source statements.
- The SYSLST JCL statement should be directed to an output device. Check the messages issued by the decompiler for errors. Correct the errors and rerun until there are no errors. Note that return codes are *not* used. The SYSLST messages are the indicators of the actual results of the process.

For more information about file usage with the decompiler, see CA IDMS DLI Transparency JCL (see page 257).

Index