IBM Data Language/I Disk Operating System/
Virtual Storage (DL/I DOS/VS)

# Application Programming: High Level Programming Interface

*Version 1 Release 7*

SH24-5009-02

IBM Data Language/I Disk Operating System/
Virtual Storage (DL/I DOS/VS)

# Application Programming: High Level Programming Interface

*Version 1 Release 7*

SH24-5009-02

**Note !**

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

**Third Edition (January 2003)**

This edition applies to Version 1 Release 7 of IBM Data Language/I Disk Operating System/Virtual Storage (DL/I DOS/VS), and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

You may also send your comments by FAX or via the Internet:

Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

# Contents

# Figures

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

Any pointers in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this publication or accessed through an IBM Web site that is mentioned in this publication.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
IBM Deutschland Informationssysteme GmbH
Department 0215
Pascal Str. 100
70569 Stuttgart
Germany
```

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

## Trademarks and Service Marks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

```
CICS
IBM
```

# Preface

This book provides information needed for the planning, writing, debugging, and execution of data base application programs using the functions and facilities of DL/I DOS/VS and the DL/I High Level Programming Interface, in conjunction with the COBOL or PL/I Optimizer language. It is directed to thos writing such programs and to those supporting such applications as part of the data base administration and system programming functions.

DL/I DOS/VS is referred to in this publication as DL/I. The DL/I High Level Programming Interface is abbreviated as DL/I HLPI.

It is assumed that you are familiar with DL/I to the extent that it is described in the *DL/I DOS/VS General Information* manual. Because you have to use either COBOL or PL/I Optimizer code with DL/I HLPI commands to make an executable program, it is assumed that you are familiar with one of these languages. Only the features of COBOL and PL/I that apply directly to DL/I applications are described in the text and examples in this book.

Use of the DL/I High Level Programming Interface requires that the CICS/VS EXEC translator be used for translation of the interface commands. Because of this translation requirement, you must be familiar with CICS/VS to the extent of being able to set up a job step to execute the translator.

If you are planning an online application, you must be familiar with CICS/VS.

This book is a guide while you are a new user of DL/I and the DL/I High Level Programming Interface. Once you are proficient in their use, or if you are already an experienced user of DL/I, it can be used as an application programming reference manual. To accomplish these purposes, the book is divided into four chapters.

- ***Chapter 1: DL/I, Data Bases, and the Application Programmer.*** This chapter gives you an overview of DL/I and data bases, and an introduction to the DL/I High Level Programming Interface and the way in which you use it to perform the DL/I functions.
- ***Chapter 2: DL/I High Level Programming Interface.*** This chapter describes the syntax of the DL/I High Level Programming Interface in detail. You should read this chapter to familiarize yourself with the formats and functions before you see them being used in the later chapters.
- ***Chapter 3: DL/I Application Program.*** This chapter guides you through the process of planning, writing, executing, and debugging a DL/I application program using the DL/I High Level Programming Interface. There are also programming examples, in each host language, showing how the commands are used in a sample application program.
- ***Chapter 4: DL/I HLPI Command Reference.*** This chapter consists of reference material that you will use as you write application programs. It lists each of the DL/I HLPI commands with all of the information you need to use and code them.

A number of terms and phrases used in this guide to describe or explain DL/I and DL/I data bases may be new to you or have new meanings in this context. In

general, they will be explained the first time they are used. The most important terms are defined in the glossary at the back of the book.

## Related Publications:

*DL/I VSE Release Guide*, SC33-6211-05

*DL/I DOS/VS Release Guide*, SC33-6211-04

*DL/I DOS/VS Data Base Administration*, SH24-5011

*DL/I DOS/VS Resource Definition and Utilities*, SH24-5021

*DL/I DOS/VS Interactive Resource Definition and Utilities*, SH24-5029.

*DL/I DOS/VS Recovery/Restart Guide*, SH24-5030.

Other DL/I publications:

*DL/I DOS/VS General Information*, GH20-1246

*DL/I DOS/VS Library Guide and Master Index*, GH24-5008

*DL/I DOS/VS Application Programming: High Level Programming Interface*, SH24-5009

*DL/I DOS/VS Application Programming: CALL and RQDLI Interface*, SH12-5411

*DL/I DOS/VS Guide for New Users*, SH24-5001

*DL/I DOS/VS Messages and Codes*, SH12-5414

*DL/I DOS/VS Diagnostic Guide*, SH24-5002

# Summary of Changes

**Summary of Changes**
**for SH24-5009-02**
**Version 1.7**

This edition has been revised to include information concerning the use of the MPS Restart Facility, Boolean AND and OR operators with the WHERE clause, and inclusion of the Key Feedback option. Various additions, corrections, and improvements are also included.

**Summary of Changes**
**for SH24-509-01**
**Version 1.6**

This edition has been revised to include changes in the title of this manual and in the titles of other DL/I DOS/VS manuals produced for Version 1.6 of DL/I DOS/VS. Miscellaneous changes also have been made for clarification of existing information.

# Chapter 1. DL/I, Data Bases and the Application Programmer

This chapter gives you an overview of DL/I and data bases, and an introduction to the DL/I High Level Programming Interface. If you have never used either DL/I or the DL/I HLPI before, you should read this entire chapter. If you are already an experienced DL/I user you can skim the first section, but you should read the material in the second section before going further in the book. The contents of the two sections are summarized here:

- Getting Acquainted With DL/I and Data Bases
  - DL/I data bases and their characteristics
  - functions that can be performed on data bases
  - views of the data base
  - DL/I operating environments
- Getting Acquainted With the DL/I High Level Programming Interface
  - commands for performing the DL/I functions
  - DL/I response to the commands

## Introduction

Data Language/I Disk Operating System/Virtual Storage (DL/I DOS/VS) is a data management control system developed to help you create, access, and maintain large common data bases. You can use it in conjunction with the IBM Customer Information Control System/Virtual Storage (CICS/VS) to create an online data base environment.

The DL/I High Level Programming Interface (DL/I HLPI) is a simple, easy-to-use method of performing all of the functions necessary for processing DL/I data bases in a batch, multiple-partition support (MPS) batch, or CICS/VS online environment. It consists of a series of commands similar to those in the CICS/VS command language. You can code these commands as needed in application programs written in either the COBOL or PL/I Optimizer language.

## Getting Acquainted With DL/I and Data Bases

### The Segment - the DL/I Unit of Information

The primary unit of data in a DL/I data base is called a segment. There may be a number of different types of segments in a data base. In order for application programs to distinguish between them, each segment type is assigned a name. There may be any number of occurrences of a particular segment type stored in the data base.

Most segment types are composed of one or more data fields that are related and are normally processed together. A field can contain up to 256 bytes of data. The

maximum size of a segment can vary from 4068 bytes to 32766 bytes, depending on the DL/I access method used.

## Functions DL/I Performs on the Segment

There are several functions that DL/I must be able to perform for you as your application program processes information stored in data bases.

### Loading

Before your application program can use a data base, data must be loaded into it in the form of segments. This is done by executing a batch application program whose only function is the initial loading of segments into the data base.

### Retrieving

Once segments have been loaded, you can retrieve them by issuing one of three different commands, called GET commands.

### Inserting

It will often be necessary to add data to an existing data base by adding segments. DL/I takes segments from the I/O areas that you specify in your program and places them in the data base by following the commands you code in your program, or by following rules established when the data base was defined.

### Replacing

You may also need to update the data in a data base. DL/I takes the segments from I/O areas and uses them to replace segments stored in the data base.

### Deleting

You will also need to delete segments from the data base when the information they contain becomes obsolete or incorrect.

### Checkpointing

When your application program reaches a point during its execution where you want to make sure that all changes made to that point have been physically entered in the data base, you will issue a command to take a checkpoint.

### Scheduling

When operating in an online environment, you will normally be sharing data bases with other online applications. In order to control simultaneous accesses to the data bases, you will need to notify DL/I of your intention to use them by issuing a scheduling command.

### Terminating

When operating in an online environment, you will need to notify DL/I when you have finished using a data base by issuing a terminating command.

## The Data Base Hierarchy—Building on the Segment

When working with a conventional file, you must be fully aware of the way in which it is organized, the format of the records it contains, and of the physical characteristics of the file (such as block size, record length, access method used, and the location of the file). You must tailor your program to these formats and characteristics. If any change is made to them, your program may also have to be altered.

On the other hand, in DL/I data bases:

- The data fields are grouped into segments.
- The segments are grouped into data base records. A data base record can be made up of a number of different segment types.
- Data base records are grouped into a data base.

You need to be aware of the format of only those fields and segments that store the data needed by your particular application. You do not need to know the physical characteristics of the data base or where it is located. Most changes made to the physical characteristics do not affect you, and your program does not need to be altered.

The segments making up a DL/I data base record can be viewed as being arranged into a hierarchical data structure. The upper part of Figure 1-1 shows a physical record in a conventional file with the data elements labeled NAME, ADDRESS, and PAYROLL. The same elements (segments) are shown in the lower portion of the figure as they might be viewed by DL/I in the form of a hierarchical data structure. The way the segments are physically stored may differ significantly from the way the data is viewed as a data structure.

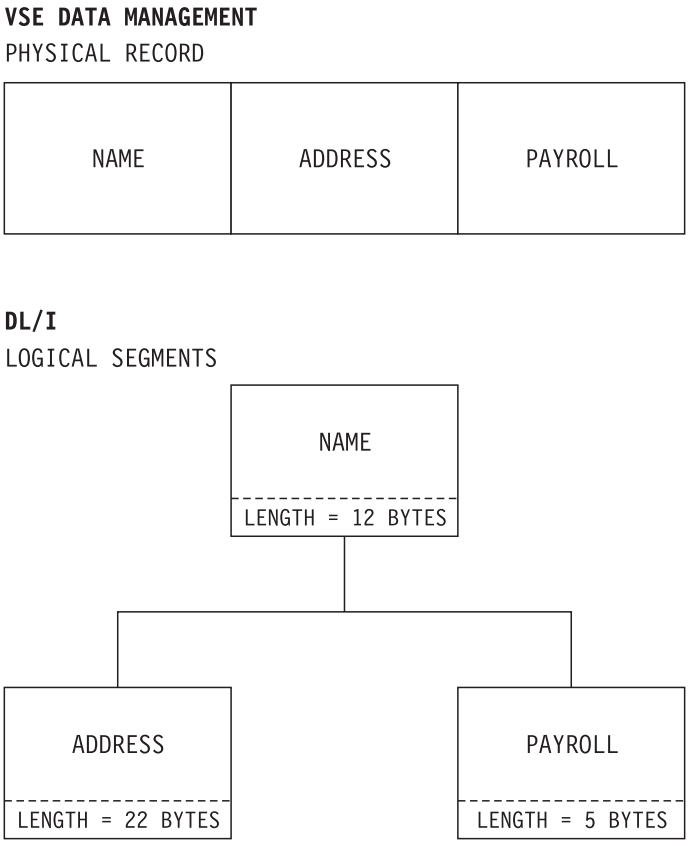



*Figure 1-1. Physical Record - Segment Relationship (Example 1)*

Another example of a conventional physical record is shown in Figure 1-2 on page 1-4. Again, the lower part of the figure illustrates a hierarchical data structure that DL/I might make available to you. SKILL, NAME, EXPERIEN, and EDUCAT are the segments that make up the data base records of the skills inventory data base. Although not shown in Figure 1-2 on page 1-4, there may be multiple

EXPERIEN and EDUCAT segments for each name, many names for each skill, and many skills. Figure 1-3 on page 1-5 illustrates this with a typical data base record from the skills inventory data base. Notice that, because many employees may have the same skill, multiple NAME segments exist under the first SKILL segment. Similarly, multiple EDUCAT segments exist under each NAME segment, and multiple EXPERIEN segments exist under two of the NAME segments.

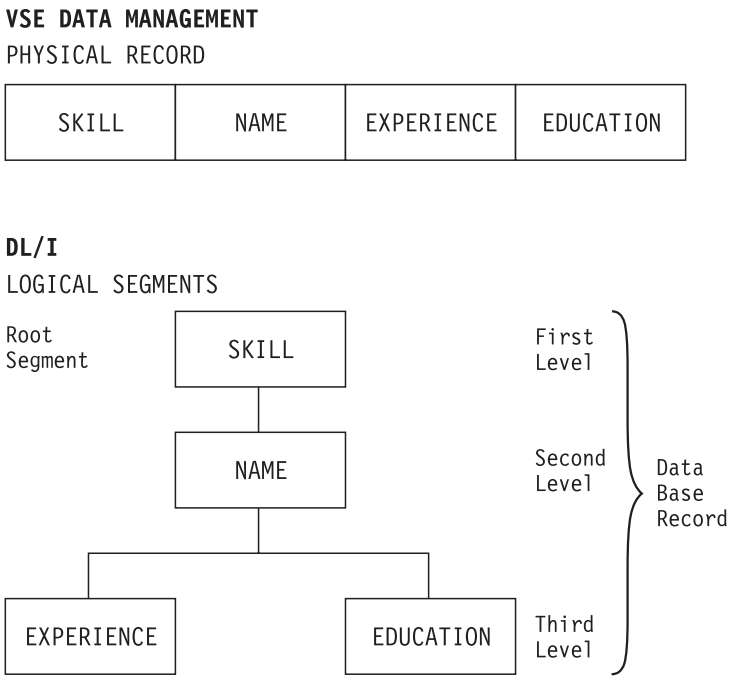



*Figure 1-2. Physical Record - Segment Relationship (Example 2)*

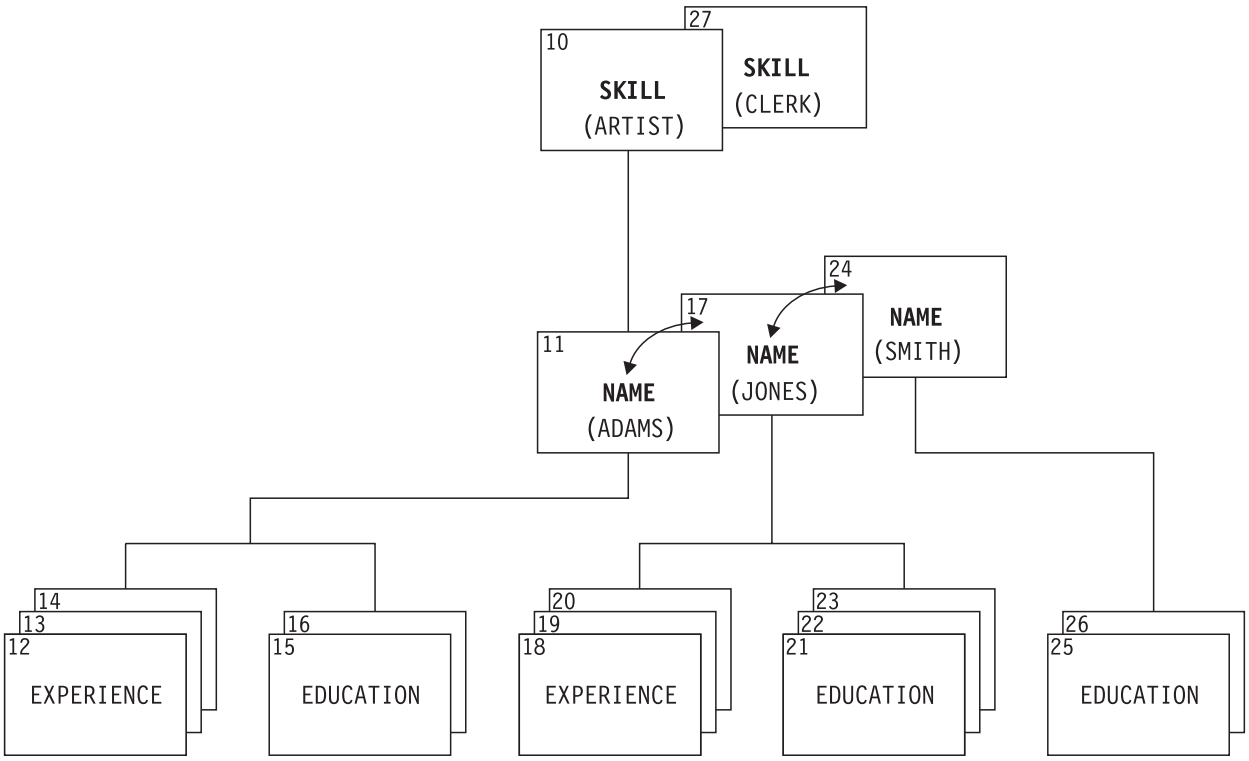



*Figure 1-3. Expanded Data Base Structure*

The following summary of the characteristics of DL/I data bases introduces many of the terms used in the rest of this guide:

- Looking at Figure 1-2 on page 1-4, you can see that all of the segments in the data structure illustrated there seem to be dependent on one segment shown at the top of the hierarchy. This segment (the SKILL segment) is called a root segment.
- A data base record is the hierarchical structure made up of all segments dependent on the single root segment. There can be only one root segment type in a data base record, and it can have only one occurrence in that data base record. A data base record can consist of one to 255 segment types; each, other than the root segment, having zero to n occurrences, as shown in Figure 1-3 A data base record can have a maximum of 15 hierarchical levels.

  Figure 1-2 on page 1-4 shows three levels of hierarchy: SKILL is at the first level, NAME at the second, and EXPERIEN and EDUCAT at the third.
- A data base consists of one to n data base records, all having the same root segment type.
- The root segment always contains a key field (except for data bases using the HSAM or simple HSAM organization). The value in the key field controls the position of the data base record within the data base.
- Segments lower in level in the hierarchy can also have key fields. These keys may be used to sequence multiple occurrences of the same segment type within a data base record.
- Segments at hierarchical levels below the root segment are said to be dependent on those above. In Figure 1-2 on page 1-4, NAME is dependent on the root segment SKILL. SKILL is the parent of NAME. NAME is a child of

SKILL. EXPERIEN and EDUCAT are dependent children of NAME. NAME is the parent of both of them. NAME, EXPERIEN, and EDUCAT are all dependent on SKILL. There can be zero to n dependent child segments per parent.

- The segments in a hierarchical structure are always referenced in the hierarchical sequence of top-to-bottom, left-to-right, front-to-back, as indicated by the numbers 10 through 27 in Figure 1-3 on page 1-5.

## Views of the Data Base

Your application program may not need to refer to all of the segment types that make up the hierarchical structure of a particular data base. Also, for security reasons, it may be advisable to restrict your program's access to certain segment types. In other words, your program needs only a particular view of the data base. A view of a data base is the portion of the total hierarchical structure to which your program has access. A data base view consists of only those segments and fields to which your program is sensitive, as though the rest of the data base does not exist.

Defining the views of each data base is a function of Data Base Administration (DBA).

### PCBs

A view of a data base is represented by a Program Communication Block (PCB). Every data base accessed by your program has a PCB associated with it. You can specify which PCB or view of the data base DL/I is to use in each data base command. Other programs can use the same views of the data base.

### PSBs

PCBs are grouped within a Program Specification Block (PSB) generated by DBA. There is at least one PSB generated and cataloged in a core image library for each application program that uses DL/I. You must tell DL/I which PSB your program is to use before executing any data base commands.

### DBDs

The information describing a data base record—the relationships between segments, the physical auxiliary storage device used, and the access method used by DL/I—is stored apart from both the data base and the application program in a data base description (DBD) block. The DBD is a control block that is normally generated once for each data base, and stored in a core image library.

## Preparing to Use DL/I

DL/I is a program product that acts as an intermediary between your application program and a data base or bases stored on auxiliary storage devices. Figure 1-4 on page 1-8 shows the environments in which DL/I operates; and the relationships between DL/I, the data base, and your program.

Before your program can execute in conjunction with DL/I, several steps must be performed by DBA:

- Generate DBDs for the data bases to be used
- Generate PSBs for your program
- Generate DL/I control blocks
- Prepare VSE/VSAM files
- Load the data bases to be used.

Now, the application program you have written in COBOL or PL/I can be translated, compiled, link-edited, cataloged into a core image library, and executed.

## Operating Environments

DL/I supports three operating environments: batch, multiple-partition support (MPS) batch, and online. The differences in the ways in which you plan, write, and execute your application program, depending on which environment it will be executed in, are described in Chapter 3.

Figure 1-4 on page 1-8 illustrates the differences between the three DL/I application program environments. The DIB shown in the figure is a collection of variables in your program into which DL/I returns status information.

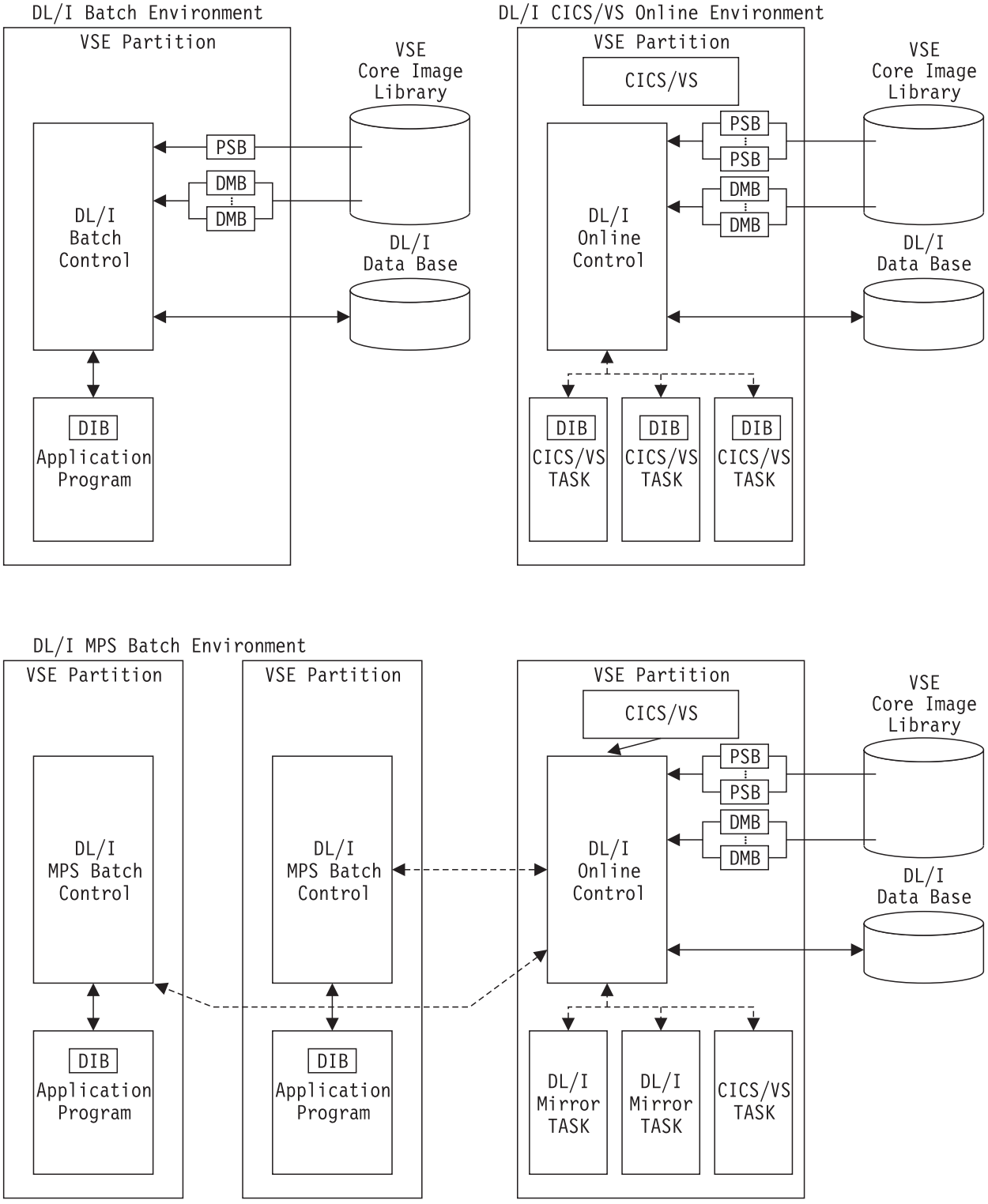



*Figure 1-4. The DL/I Environments*

## Getting Acquainted With the DL/I High Level Programming Interface

The DL/I HLPI is your means of performing the DL/I functions that were described earlier. It provides you with a set of easily coded commands that provide all of the information that DL/I needs in order to execute those functions. This section shows you what the commands look like, and how to use them in your program. More detailed information on all aspects of the DL/I HLPI is given in Chapter 2 for your use once you are familiar with the basic concepts.

## The DL/I Commands for Performing the Functions

The DL/I HLPI commands are similar to those in the CICS/VS command language. You code them in your application program at the points where you need to perform their functions.

The syntax of the commands looks like this:

```
trigger  function  [options and arguments]  command-delimiter
```

A unique identifier is required for DL/I to recognize that you have coded a command. This identifier is called a trigger. The function portion of the command tells DL/I what you want it to do. The information DL/I needs to perform that function is provided in the form of options and arguments. The command-delimiter signals the end of the command. An example of a command, as used in a PL/I program, is:

```
EXECUTE DLI GET NEXT  INTO(IOAREA)   ;
|_________| |_____|  |__________|  |_|
  trigger   function  option with   command-
                        argument    delimiter
```

### GET Commands

The DL/I HLPI provides three commands, called GET commands, for you to use in retrieving segments from data bases. They are GET NEXT, GET NEXT IN PARENT, and GET UNIQUE.

### GET NEXT Command

In order for you to sequentially retrieve segments from a data base, DL/I maintains a position pointer that remembers your current position in the data base. Each time you request DL/I to sequentially retrieve a segment, DL/I retrieves the segment pointed to by the position pointer and updates the pointer to the next segment.

The GET NEXT command is designed to always get the next segment—the one pointed to by the position pointer. If your program executed a continuous series of GET NEXT commands, it would eventually retrieve, in hierarchical order, every segment in your program's view of the data base.

A GET NEXT command as coded in a PL/I application program might look like this:

```
EXECUTE DLI GET NEXT INTO(AREA);
```

EXECUTE DLI is the trigger that indicates that this is a DL/I HLPI command. GET NEXT names the function to be performed. INTO(AREA) is an option with an argument that indicates the area, which you have previously defined in the host language of your program, into which you want to transfer the data contained in the segment that is retrieved. The ";" is the command-delimiter used in PL/I programs to signal the end of a command (in a COBOL program, the command-delimiter

would be END-EXEC). If this was the first DL/I command to be executed in your program, the segment that would appear in AREA would be the root segment of the first data base record in the data base. If this was the skills inventory data base as shown in Figure 1-3 on page 1-5, this segment would be the SKILL segment for ARTIST. The position pointer would then point to the NAME segment for ADAMS. If the same GET NEXT command was executed again, that segment would appear in AREA. The position pointer would then point to the first EXPERIEN segment of ADAMS. Executing the GET NEXT again would retrieve that segment. This process, if continued, would proceed through the data base from top-to-bottom, left-to-right, front-to-back in hierarchical order.

## GET NEXT IN PARENT Command

The GET NEXT IN PARENT command is a variation of the GET NEXT command. As was described above, segments higher than others in the hierarchical structure of a data base are called parent segments of those below them in the same hierarchical path. The GET NEXT IN PARENT command makes it possible for you to sequentially retrieve the data from all of the segments that are children of a particular parent segment. The parentage that applies to this command is that established by the last previous GET NEXT or GET UNIQUE command. For instance, if the command

```
EXECUTE DLI GET NEXT SEGMENT(NAME) INTO(NAMEIO);
```

had just been successfully executed, a GET NEXT IN PARENT command would use the NAME segment as the designated parent.

If the GET NEXT command shown above had been executed, the GET NEXT IN PARENT commands

```
EXECUTE DLI GET NEXT IN PARENT SEGMENT(EXPERIEN) INTO(EXPRIO);
```

*or*

```
EXECUTE DLI GET NEXT IN PARENT SEGMENT(EDUCAT) INTO(EDUCIO);
```

would retrieve their respective segments. The first EXPERIEN segment under the NAME segment would appear in EXPRIO, or the first EDUCAT segment under the NAME segment would appear in EDUCIO.

## GET UNIQUE Command

The final type of GET command is GET UNIQUE. You can use it to randomly retrieve any segment in your program's view of the data base. Unlike the two GET NEXT commands, it does not depend on the position pointer, and thus is able to retrieve a segment from any location in the data base—even from levels higher in the hierarchy than the position pointer was indicating before the GET UNIQUE command was executed.

You must code the SEGMENT option and argument in a GET UNIQUE command to indicate the segment type you want to retrieve. This is called unqualified segment selection. The first occurrence of the named segment type is retrieved.

A GET UNIQUE command with unqualified segment selection might look like this:

```
EXECUTE DLI GET UNIQUE
        SEGMENT(SKILL) INTO(SKILLIO);
```

If this were the first command in the program, this would retrieve the ARTIST segment (see Figure 1-3 on page 1-5) from the data base and place it in SKILLIO.

If this command were repeated, the CLERK segment would be retrieved and placed in SKILLIO.

When you need to retrieve a specific occurrence of a segment type you must qualify the segment selection. You do this by coding the WHERE option after the SEGMENT option. The WHERE option provides the information that DL/I needs to locate the specific segment that you want.

A GET UNIQUE command with qualified segment selection might look like this:

```
EXECUTE DLI GET UNIQUE
        SEGMENT(SKILL) INTO(SKILLIO) WHERE(SKILCODE=SKILVAR);
```

SKILCODE is the name of a field in the SKILL segment. SKILVAR is the name of a variable previously defined in the host language of your program.

In this example, the WHERE option specifies that the occurrence of the SKILL segment type in which the SKILCODE field equals the value in SKILVAR will be searched for and returned in the SKILLIO area if it is found.

After the successful execution of this command, the position pointer would point to the segment following the SKILL segment that was retrieved. In Figure 1-3 on page 1-5, that would be the first NAME segment (ADAMS) under the SKILL segment (ARTIST). A GET NEXT or GET NEXT IN PARENT command could be used to retrieve that segment.

## INSERT Command

The INSERT command is used to add new segments to an existing data base. When the command is executed, DL/I takes the data from the area you specified and adds the segment to the data base. An INSERT command looks like this:

```
EXECUTE DLI INSERT SEGMENT(EXPERIEN) FROM(EXPRIO);
```

Notice the option FROM. FROM is used in place of INTO to indicate that data is to be transferred from an area in your program to the data base.

## REPLACE Command

The REPLACE command replaces the data in a segment in the data base with different data. Before executing a REPLACE command, your program must have retrieved the existing data from the data base with a GET command. Your program can then modify that data and execute the REPLACE command to place the modified data in the data base. This is an example of a REPLACE command (the preceding GET command is assumed):

```
EXECUTE DLI REPLACE SEGMENT(EXPERIEN) FROM(EXPRIO);
```

## DELETE Command

The DL/I HLPI provides the DELETE command for use when you must delete a segment from the data base because it contains data that is obsolete, extraneous, or incorrect. Before your program executes a DELETE command, it must have retrieved the segment to be deleted by executing one of the GET commands. The DELETE command will then remove the named segment and all of its dependent children from the data base. Because of this deletion of dependent children, you must use caution when you issue a DELETE. An example of a DELETE command looks like this (assuming that a GET command has been executed that specified the same segment).

```
EXECUTE DLI DELETE SEGMENT(NAME) FROM(NAMEIO);
```

The NAME segment and its dependent children—all of the EXPERIEN and EDUCAT segments under it—would be removed from the data base.

## LOAD Command

The DL/I HLPI LOAD command is used in a batch application program to initially load segments into an empty data base. No other DL/I HLPI commands are allowed in this program. The LOAD command may not be used in any other type of application such as MPS batch or online.

Here is an example of a LOAD command:

```
EXECUTE DLI LOAD SEGMENT(SKILL) FROM(SEGDATA);
```

## CHECKPOINT Command

If your program ends abnormally, any changes you made to the data base to that point should be backed out (restored to their previous state) so that the data base is not left in a partially updated condition for access by other application programs. The backout is performed by a DL/I utility program in the batch environment and by CICS/VS dynamic transaction backout in the MPS batch and online environments.

If your program is a long-running one, you can reduce the amount of backout that might be necessary by taking checkpoints. When a logically complete set of updates has been completed, and your program is about to begin another set of updates, take a checkpoint by issuing the CHECKPOINT command. This signals the back-out utility to stop at this point. You can then restart your program from the same point to resume execution.

Examples of CHECKPOINT commands are:

```
EXECUTE DLI CHECKPOINT ID(CHKPID);
```

*and*

```
EXECUTE DLI CHECKPOINT ID('CHKP0007');
```

where CHKPID is the name of an eight-byte character string that uniquely identifies this checkpoint. Alternatively, the checkpoint ID can be coded in the command, enclosed in single quotes as in the second example.

## SCHEDULE Command

Before an online application can access DL/I data bases, it must schedule a PSB for its use. The DL/I HLPI provides the SCHEDULE command for this purpose. It is coded like this:

```
EXECUTE DLI SCHEDULE PSB(CUSTDB);
```

where CUSTDB is a constant naming one of the PSBs available to your application.

## TERMINATE Command

When your online application has no further immediate need for a PSB, it should be released so that it will be available for other applications that may be active. You do this by issuing a DL/I TERMINATE command that looks like this:

```
EXECUTE DLI TERMINATE;
```

If your program needs to refer to a data base again at a later point, you must reschedule a PSB with another SCHEDULE command at that time.

## The DL/I Response to the Commands

To make DL/I status information available to your program, variables called the DL/I Interface Block (DIB) are automatically defined in your program. The most important of these is named DIBSTAT. In it, DL/I returns a two-character status code. The status code makes it possible for you to check the results of each command you issue, immediately following the execution of that command, so you can be aware of the condition of the data base and take appropriate action where needed.

# Chapter 2. DL/I High Level Programming Interface

This chapter describes the syntax of the DL/I High Level Programming Interface commands and gives the purpose and use of each element of the syntax. You should be thoroughly familiar with this material before reading the rest of this book.

This chapter includes:

- a brief description of the DL/I HLPI
- the elements of the command language
- a summary of the syntax of the commands.

## What it is

The DL/I High Level Programming Interface is an easy-to-use method of performing all of the functions necessary for processing DL/I data bases in a batch, MPS batch, or CICS/VS online environment. It is made up of a series of commands similar in syntax to those in the CICS/VS command language. You can code these commands as needed in an application program written in either the COBOL or PL/I Optimizer language. Before compiling a program using the DL/I HLPI, you must execute the CICS/VS EXEC translator, as a separate job step, to convert the commands into statements appropriate to the host language.

## Coding Conventions

The following conventions are followed in illustrating the format and coding of commands, control statements, and messages:

- Commands and control statements are free form unless stated otherwise. Where keywords, operands, and parameters are shown separated by commas, no blanks may appear immediately before or after the comma. Where keywords, operands, and parameters are shown separated by blanks, any number of blanks may be used. For example:

  ```
  DLI,progname,psbname
  ```

  shows the use of commas, and

  ```
  EXECUTE  DLI  TERMINATE;
  ```

  shows the use of blanks.

- Uppercase letters, stand-alone numbers, and punctuation marks (including parentheses) must be coded exactly as shown. For example:

  ```
  END-EXEC
  ```

  is coded as shown.

  The only exceptions to this convention are brackets [ ], braces { }, ellipses ..., and subscripts. These are never coded.

- Lowercase letters, words, and associated numbers represent variables for which specific information or values must be substituted. For example:

```
SEGMENT(name)
```

where “name” must be replaced by the appropriate segment name.

- The symbol b is used to indicate one blank position at points where confusion might result if it was omitted.
- Brackets [ ] indicate that the items or groups of items within them are optional; they may be omitted if not required. For example:

  ```
  [VARIABLE]
  ```

  is optional.

  Any item or group of items not within brackets must be coded.
- All stacked items enclosed within braces { } represent alternatives, one of which must be coded unless the braces are enclosed within brackets. No more than one of the stacked items may be coded. For example:

  ```
  {EXECUTE}
  {EXEC   }
  ```

  where one of the items must be coded.
- If an alternative item is underlined, that item is the default: that is, DL/I automatically assumes that the underlined item is the choice if none of the items is coded. For example:

  ```
  {PAUSE  }
  {NOPAUSE}
  ```

  where PAUSE is the default.
- Ellipses, ... , indicate that the preceding item or group of items can be coded more than once in succession, as necessary. For example:

  ```
  [SEGMENT(name) FROM(reference)]...
  ```

  where everything within the brackets is optional, but may be repeated if coded.

# Elements of the Command Language

## Syntax Description

The syntax of the DL/I High Level Programming Interface commands looks like this:

```
trigger   function  [options and arguments]   command-delimiter
```

- *Trigger* identifies the statement to the translator as being a DL/I HLPI command.
- *Function* names the operation the command is to perform.
- *Options* provide information needed to perform the particular operation named by *function.*
- *Arguments* are included, within parentheses, with certain options to pass names, constants, or variable values associated with the option.
- *Command-delimiter* identifies the end of the statement to the translator.

This is an example of a command as it would appear in a PL/I program:

```
EXECUTE DLI  TERMINATE   ;
|________|  |_______|  |_|
  trigger    function   command-
                        delimiter
```

Here is an example of another command as it would appear in a COBOL program:

```
EXECUTE DLI  GET NEXT  USING PCB(1) KEYFEEDBACK(SKILLCOD)
                            FEEDBACKLEN(10)
                            SEGMENT(SKILL) SEGLENGTH(8)
                            WHERE(CODE=ARTIST) INTO(IOAREA)  END-EXEC
|________|  |_____|  |_____________________________________|  |_____|
  trigger    function         options with arguments         command-
                                                             delimiter
```

The rest of this section describes, in detail, the various parts that make up the commands and their purposes in performing the functions called for by the commands.

Every command must include a trigger, a function, and a command-delimiter. In addition, most commands include options and arguments.

Each item in the list below represents a part of the syntax of DL/I HLPI commands. Each of these parts is described in a section of the text that follows. The numbers in the left margin indicate the number of the section where that item is discussed.

```
1)   {EXECUTE} DLI
     {EXEC   }

2)    GET NEXT

3)    USING PCB(expression)

4)    KEYFEEDBACK(reference)
      FEEDBACKLEN(expression)

5)    SEGMENT(name)

     {INTO(reference)}
     {FROM(reference)}

      SEGLENGTH(expression)
      WHERE(name op reference)
      FIELDLENGTH(expression)

6)   {;       }
     {END-EXEC}

7)    VARIABLE

8)   {FIRST}
     {LAST }

9)    LOCKED

10)   OFFSET

11)   PSB(name)

12)   ID(expression)
```

## 1. Trigger—EXECUTE DLI

To identify a DL/I HLPI command to the translator, a unique identifier, called a trigger, is required. You must code it at the beginning of every command. Either of these forms is accepted:

```
{EXECUTE} DLI
{EXEC   }
```

These keywords are reserved for this use only. Do not use either combination of them for any other purpose in your program.

## 2. Function to be Performed

Always coded next within a command is the function that defines the operation you wish it to perform. The associated options and arguments coded with the function provide all the information necessary to complete the operation. Each command must have a function. The individual functions are discussed in detail in Chapter 4.

The functions that perform data base operations are:

- GET NEXT
- GET NEXT IN PARENT
- GET UNIQUE
- INSERT
- REPLACE
- DELETE

- LOAD

The function that takes a DL/I checkpoint is:

- CHECKPOINT

The functions that schedule and terminate PSBs in online operation are:

- SCHEDULE
- TERMINATE

## 3. Specifying the PCB

You can specify a particular PCB (view of a data base) to be used for performing a data base function by coding the PCB option. If you do not specify the PCB option, DL/I uses the first PCB defined in the currently scheduled PSB. If you do code the PCB option, it must follow the function and looks like this:

```
USING PCB(expression)
```

“expression” in COBOL is any valid integer variable or integer constant. “expression” in PL/I is any valid expression that converts to the integer data type. The value of the expression must be a positive integer not greater than the number of PCBs generated for the PSB. The first PCB is PCB(1), the second is PCB(2), and so on.

## 4. Retrieving Key Feedback

You may optionally provide an area in your application program for DL/I key feedback information. KEYFEEDBACK and FEEDBACKLEN can be specified with GET commands to retrieve the concatenated key from the PCB into the area referenced by KEYFEEDBACK. FEEDBACKLEN may be specified to tell HLP1 the length of the area referenced by KEYFEEDBACK. If the KEYFEEDBACK area is not large enough for the concatenated key as found in the PCB, as much of the key as fits is moved into the user's area (truncated on the right). Before the move id done, the length of the concatenated key is placed in a field called DIBKFBL in the user DIB. In COBOL, FEEDBACKLEN must be specified whenever KEYFEEDBACK is used. In PL/I, however, FEEDBACKLEN is optional; if it is not specified, the default length of the area is used.

If you use the key feedback option, it must be coded after the function and before the parent or object segment options. The key feedback option looks like this:

```
KEYFEEDBACK(reference) FEEDBACKLEN(expression)
```

where “reference” is the name of an area previously defined in the host language of your application program where you want the key placed and “expression” in COBOL is any valid integer variable or integer constant. “expression” in PL/I is any valid expression that converts to the integer data type. If “expression” is variable, it should be declared as a binary halfword value. Since KEYFEEDBACK and FEEDBACKLEN are statement level parameters, they may be specified only once in an EXEC call.

**Note:** If you specify FEEDBACKLEN as being larger than the user key feedback area, it is possible to overlay storage.

# 5. Selection of Segments

In each command that accesses a data base, you must specify to DL/I the segment or segments that you want it to operate on (except in GET NEXT and GET NEXT IN PARENT commands, where it is optional).

## Parent Segments

A parent segment is a segment used to help identify the hierarchical path leading to the segment at the lowest level in which you are interested (the object segment). A maximum of 14 parent segments can be specified in one data base function command corresponding to the 14 possible hierarchical levels above the lowest level in a DL/I data base hierarchy. Parent segments, when specified, must be coded in hierarchical order from top-to-bottom, the highest level parent segment first. It is not necessary to specify a parent segment on every hierarchical level. When you want data to be transferred for a parent segment, you must code the INTO or FROM option (described below) following the SEGMENT option.

Parent segment selection looks like this:

```
SEGMENT(name)
```

where “name” is the name of a segment.

## Object Segments

The object segment in a data base function command is the segment at the lowest hierarchical level in which you are interested. You must specify at least an object segment in every data base function command except GET NEXT and GET NEXT IN PARENT, where it is optional. INTO or FROM (described below) must be coded with the object segment selection specification in order to identify an I/O area for the segment data. The object segment selection specification must be coded after the last of any parent segment selection.

Object segment selection looks like this:

```
SEGMENT(name)
```

where “name” is the name of a segment.

## Segment I/O Area

To indicate that you want a segment to be transferred to or from a segment I/O area, you must code the INTO or FROM option following the corresponding SEGMENT option in a data base command.

INTO is coded in commands that retrieve segments from the data base (GET UNIQUE, GET NEXT, and GET NEXT IN PARENT) and placed into a segment I/O area. FROM is coded in commands that modify the data base (INSERT, REPLACE, DELETE, and LOAD) with data from a segment I/O area.

The INTO or FROM argument identifies an area in your program large enough to contain the segment being transferred.

You code INTO and FROM like this:

```
INTO(reference)
```

*or*

```
FROM(reference)
```

where “reference” is the name of an area previously defined in the host language of your application program.

When data transfer is requested for one or more parent segments, that request is known as a path call. Coding the SEGMENT option for a parent segment, without also coding INTO or FROM, indicates that the named segment is to be used in establishing the hierarchical path to the object segment named in the command, but that you do not want data for that segment to be transferred. Within a command, SEGMENT options both with and without FROM or INTO can be used. Data is transferred for only those segments with FROM or INTO specified.

In case of a segment-not-found condition in a path call, data is transferred for all segments in the path, for which FROM or INTO was specified, above that which could not be found.

Processing option “P” must be specified during PSB generation for any segment to be used in path calls. Otherwise, an AM status code is returned and the program terminated.

You can insert multiple segments in a hierarchical path with one INSERT command. However, this is not permitted if a logical child segment (see *DL/I DOS/VS Data Base Administration*) is present in the path.

In a REPLACE command following an associated path call GET command, code FROM for each segment to be replaced from those retrieved by the GET.

INTO is used in commands with the GET UNIQUE, GET NEXT, and GET NEXT IN PARENT functions. FROM is used in commands with the INSERT, LOAD, REPLACE, and DELETE functions.

### Segment Length

The SEGLENGTH option defines the length of the segment named by the SEGMENT option.

In an application program using COBOL as the host language, you must code the SEGLENGTH option if you code the INTO or FROM option. The SEGLENGTH requirement in an HLPI COBOL program is due to a COBOL language restriction. This restriction makes it impossible to determine the declared length of an IOAREA at application program execution time. Since PL/I does not have this restriction, SEGLENGTH is optional in a PL/I program. If you do not code SEGLENGTH, the length always defaults to the length of the area named in the INTO or FROM option.

You code the SEGLENGTH option following the SEGMENT option, like this:

```
SEGLENGTH(expression)
```

“expression” in COBOL is any valid integer variable or integer constant. “expression” in PL/I is any valid expression that converts to the integer data type. If “expression” is a variable, it should be declared as a binary halfword value.

For *fixed length segments,* you are responsible for ensuring that the value of SEGLENGTH, if specified, is the proper value. On a GET command, storage in your application program can be overlaid if the I/O area is not large enough to contain the retrieved segment. Invalid data can be placed in the data base if the I/O area length is incorrectly specified in an INSERT or REPLACE command.

For *variable length segments,* SEGLENGTH defines the maximum segment length plus the two-byte length field at the beginning of the data area. For GET commands, it must be at least as large as the longest occurrence of the associated segment that can be retrieved by the command.

For concatenated segments, SEGLENGTH also includes the lengths of the concatenated key, the intersection data, the destination parent segment, and, for variable length destination parent segments, the two byte length field. (See "OFFSET.")

## Qualified Segment Selection

Qualified segment selection causes DL/I to search for an occurrence of a segment that contains a field with data that meets the criteria you specify.

Any segment field defined to DL/I can be used in segment selection. However, for performance reasons, qualification of root segments using fields other than the key field should be avoided. DL/I has to scan the data base sequentially for such requests. Qualification of dependent segments on non-key fields is often desirable and should be used as required.

Qualified segment selection is coded after the associated SEGMENT option like this:

```
WHERE(name op reference[{AND} name op reference]...)
                        {OR }
[FIELDLENGTH(expression[,expression]...)]
```

where "name" is the name of a field in the associated segment type. You can not use a variable for name. The value that you specify for "op" must be one of the following:

**>** (greater than)

**<** (less than)

**=** (equal to)

**¬=** (not equal to)

**<=** (less than or equal to)

**>=** (greater than or equal to)

**=<** (equal to or less than)

**¬>** (not greater than)

**=>** (equal to or greater than)

**¬<** (not less than)

In "reference," you code a variable name previously defined in the host language of your application program. Its length must be equal to the length of the field you specified with "name." The length value of this variable has a maximum of 255 bytes and is given by the FIELDLENGTH keyword which is described later in this section.

Boolean operators are available in the HLPI commands to allow multiple qualifications on a single segment type and therefore make HLPI commands more versatile.

The Boolean operators AND and OR may be used in the WHERE clause to provide you with Boolean logic capability in segment select qualification.

All relational statements connected by AND operators are considered a qualification statement. An OR operator separates qualification statements. A qualification statement can consist of one or more relational statements. To satisfy a WHERE clause, a segment can satisfy any qualification statement. To satisfy any qualification statement, the segment must satisfy all relational statements within that qualification statement.

The qualification statement test is terminated as soon as a segment type that satisfies a qualification statement is found in the data base. This procedure continues for all WHERE clauses in a DL/I data base command, until the desired segment is found.

The maximum of twelve relational conditions, connected by a maximum of eleven Boolean operators in any combination of ANDs and ORs can be specified in a single WHERE clause. Parentheses are not allowed within the WHERE clause and Boolean operators AND and OR must be surrounded by blanks.

FIELDLENGTH is a variable that has a maximum length of 255 bytes and must be specified when the host language is COBOL, but need not be coded when the host language is PL/I. When not specified in PL/I, the value of FIELDLENGTH defaults to the length of the “reference.”

Each “expression” for the FIELDLENGTH keyword corresponds one-to-one for each “reference” in the WHERE clause. “expression” in COBOL is any valid integer variable or integer constant. “expression” in PL/I is any valid expression that converts to the integer data type. If “expression” is a variable, it should be declared as a binary halfword value.

The WHERE option operates as follows: DL/I examines the data in the field you name, in each occurrence of the segment type specified in the SEGMENT option, and compares it with the value specified in “reference” according to the operator coded in “op.” The Boolean AND and OR operators are analyzed to determine further conditions on the selection. DL/I stops its examination when it finds a value that satisfies the relationship or reaches the last occurrence of the named segment type.

## Segment Selection Examples

### Example of Using AND and OR in a WHERE Statement

Figure 2-1 on page 2-10 shows a portion of a data base to illustrate the use of the Boolean operators AND and OR in a WHERE statement.

For this example, we will use a qualified segment selection as follows:

```
WHERE (TYPENO<C800 AND TYPENO>C700 AND DESCR=JOBS OR MONTHS>C100)
```

where C800, C700, C100, and JOBS are program-defined variables with respective values of 800, 700, 100, and COMIC.

Within the WHERE clause, one or more relational conditions are connected by the AND operator to form a qualification statement. The OR operator marks the beginning of a new qualification statement.

There are two qualification statements in the WHERE clause used in this example:

```
TYPENO<C800 AND TYPENO>C700 AND DESCR=JOBS   (Statement 1)
OR MONTHS>C100                               (Statement 2)
```

A segment is selected if it satisfies any qualification statement within the WHERE clause. To satisfy a qualification statement, a segment must satisfy all the relational conditions within the qualification statement.

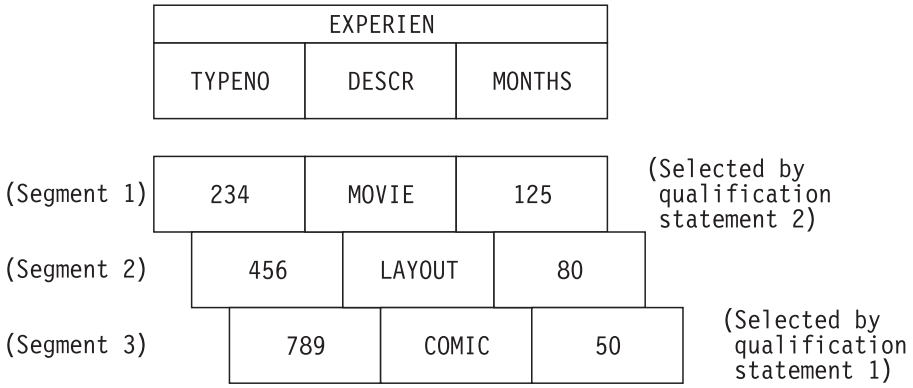



*Figure 2-1. Data Base to Illustrate AND and OR Example*

In the above example, if this WHERE clause is part of a GET NEXT command, and if we are positioned in front of these three segment occurrences, the command would retrieve the first EXPERIEN segment (Segment 1) because it satisfies the second qualification statement (MONTHS>C100). The last EXPERIEN segment (Segment 3) satisfies the first qualification statement (TYPENO<C800 AND TYPENO>C700 AND DESCR=JOBS) and would be retrieved if the command is repeated.

Figure 2-2 shows a portion of the data base to further illustrate the use of the Boolean operators in a WHERE statement.

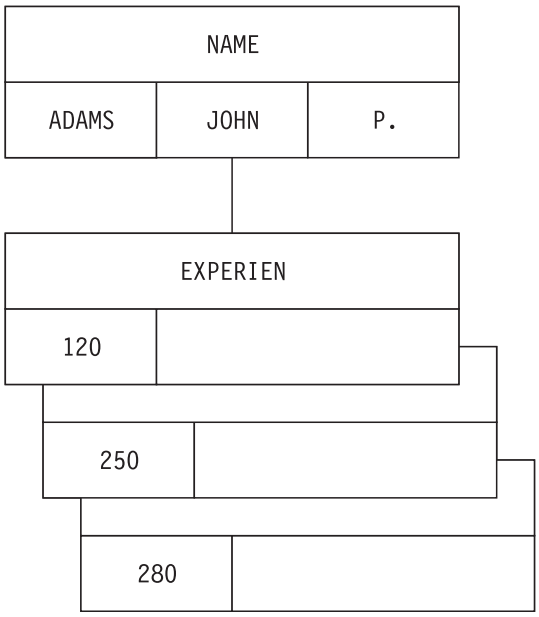



*Figure 2-2. Data Base Portion to Illustrate Coding Boolean Operators*

If we wanted to retrieve the EXPERIEN segment occurrence with type number 250 in the above data base example, we could code the following PL/I example:

```
DCL EXPRAREA                  CHAR(20);
DCL TYPENO1                   CHAR(3) INIT('100');
DCL TYPENO2                   CHAR(3) INIT('200');
EXEC  DLI GET UNIQUE
      SEGMENT(EXPERIEN)
      INTO(EXPRAREA)
      WHERE (TYPENO<TYPENO1 or TYPENO>TYPENO2);
```

If this command is repeated, the EXPERIEN segment with type number 280 would be retrieved.

The following example illustrates parent and object segments and the use of the INTO and WHERE options:

```
EXECUTE DL GET UNIQUE
        SEGMENT(SKILL) WHERE(SKILCODE=SKILVAR) (parent segment)
        SEGMENT(NAME) INTO(NAMEI0);            (object segment)
```

## 6. Command-Delimiter

Every DL/I High Level Programming Interface command that you code must include a command-delimiter to identify the end of the command to the translator. The particular delimiter you code depends on which host language you are using.

For ANS COBOL programs, the command-delimiter looks like this:

```
END-EXEC
```

This allows the command to be included within a THEN clause without being the only statement of the THEN clause.

In PL/I Optimizer programs, the command-delimiter is the semicolon, as with all other PL/I statements.

## 7. Variable Length Segments (HDAM and HIDAM Data Bases Only)

Occasionally, particularly when processing descriptive text data, it is advantageous to have segments of variable length. This results in a saving of storage space since every segment doesn't have to be as long as the longest segment anticipated. DL/I provides this variable length capability only for data bases using the HDAM or HIDAM organization.

A variable length segment, as it appears in the segment I/O area of your program, contains, in its first two bytes, a binary value describing the segment size, followed by the segment data. This two byte field describes the segment length as the application program sees it, and includes the two bytes of the size field itself. The minimum valid segment size is four bytes.

Your application program processes variable length segments in the same way as fixed length segments, except that it must create and maintain the correct values in the size fields of these segments.

Segment retrieval, including path calls, follows normal retrieval rules. If a field used in qualified segment selection is not present because the segment is too short, a segment may be retrieved that meets the qualification, but is not the segment you intended.

On a REPLACE command, if the segment length has not changed, a one-for-one replacement takes place. Otherwise, you must change the value in the segment size field.

On an INSERT command, the value in the size field must be at least the length specified as a minimum value during DBD generation. If the value is less than that, DL/I will transfer the amount you specified, but will store it in the data base as a segment with a length equal to the minimum value. The size field value remains as you specified it.

Since the segment size field is actually part of the segment, all starting positions must be relative to the first position of the variable length segment, not the start of your data.

You must identify each variable length segment each time you request data transfer for it. You do this by coding

```
VARIABLE
```

before the associated SEGMENT option naming the segment.

If you request data transfer but do not specify VARIABLE, DL/I assumes the segment is fixed length.

Do not code VARIABLE without specifying an associated INTO or FROM option.

The first two bytes of the segment I/O area contain the segment length except in the case of concatenated segments (see “OFFSET”). A concatenated segment is composed of two segments, of which only the second may be variable length. In this case, VARIABLE must be specified for the concatenated segment.

## 8. FIRST and LAST Options

There are two options that affect which occurrence of a segment DL/I will operate on. They are the FIRST and LAST options, and they are mutually exclusive. You code them like this:

```
{FIRST}
{LAST }
```

The one that you choose must be coded before the associated SEGMENT option.

### FIRST

The FIRST option tells DL/I to start with the first occurrence of the segment type named in the SEGMENT option, under its parent, to satisfy this level of the command. By using the FIRST option, it is possible to either back up to the first occurrence of the segment type on which position is established or to back up to the first occurrence of a segment defined earlier in the hierarchy, but in the same path as the one on which position is established. The FIRST option overrides the setting of the position pointer for the associated segment. FIRST is ignored if used at the root level.

When coded in INSERT commands, FIRST applies only to segments having a nonunique sequence field, or no sequence field and RULES=(,HERE) specified during DBD generation. In the latter case, the rule is overridden by the FIRST option.

You can code FIRST in commands with the GET NEXT, GET NEXT IN PARENT, and INSERT functions.

### LAST

The LAST option tells DL/I to start at the current pointer and find the last occurrence of the segment type named in the SEGMENT option, under its parent, to satisfy this level of the command. If you have specified qualified segment selection, the segment operated on is the last one of that type that satisfies the qualification. If segment selection is unqualified, the last occurrence of the segment type under its parent is used. LAST is ignored if used at the root level.

When coded in INSERT commands, LAST applies only to segments having a nonunique sequence field, or no sequence field and RULES=(,FIRST) or RULES=(,HERE) specified during DBD generation. In the latter case, the rule is overridden by the LAST option.

You can code LAST in commands with the GET UNIQUE, GET NEXT, GET NEXT IN PARENT, and INSERT functions.

### Examples of FIRST and LAST

Figure 2-3 on page 2-14 shows a portion of a data base to illustrate the following examples of the use of FIRST and LAST.

For the FIRST example, assume that the position pointer is pointing as shown by “current position” in the figure. Execution of the command:

```
EXEC DLI GET NEXT
         FIRST SEGMENT(ITEM) INTO(ITEMAREA);
```

will retrieve the first ITEM segment under ORDER number 6789 (ITEM 120).

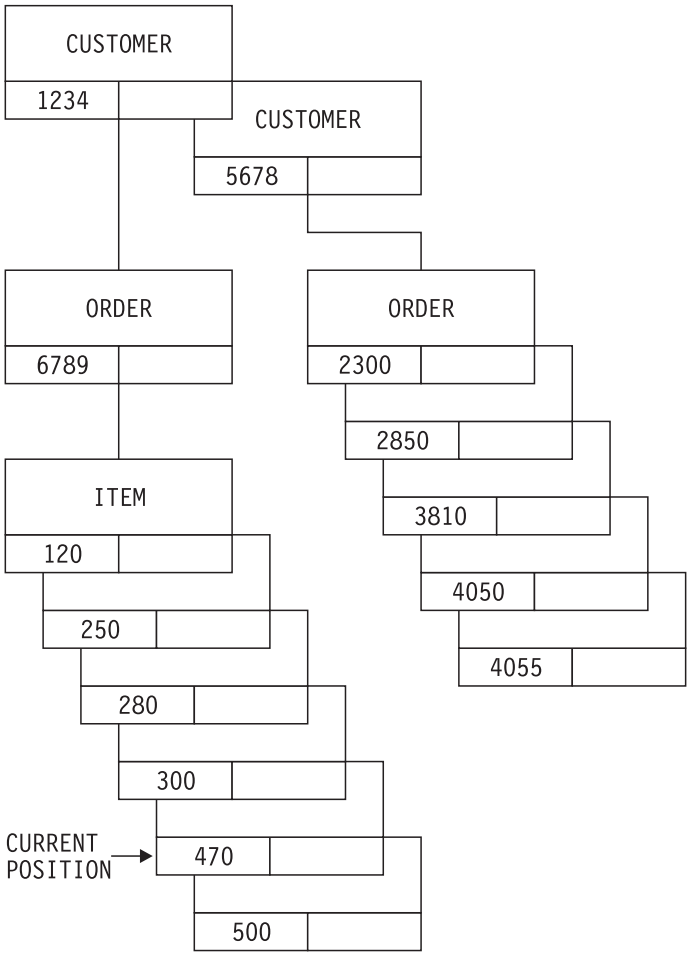



*Figure 2-3. Data Base to Illustrate FIRST and LAST examples*

Following are two examples using the LAST option:

```
EXEC DLI GET UNIQUE USING PCB(1)
         SEGMENT(CUSTOMER) WHERE(CUSTKEY=V5678)
         LAST SEGMENT(ORDER) INTO(ORDAREA);
```

will retrieve the last ORDER under CUSTOMER number 5678 (ORDER number 4055).

## 9. LOCKED

LOCKED is meaningful only when there can be contention for a single occurrence of a segment. That can happen only in an online or MPS batch environment when Program Isolation is active. (See *DL/I DOS/VS Data Base Administration* for a description of Program Isolation.)

The LOCKED option tells DL/I that your application program needs to work with a segment and that no other tasks can be allowed to modify that segment until it has finished. This means that you can retrieve segments using the LOCKED option, then retrieve them again later, knowing that they have not been altered by another application. The “LOCK,” once established, remains in effect until the application terminates or issues a CHECKPOINT or TERMINATE command. LOCKED can be specified only if INTO is also specified for the associated segment.

You can code LOCKED in commands with the GET UNIQUE, GET NEXT, and GET NEXT IN PARENT functions. LOCKED must be coded after the associated SEGMENT option.

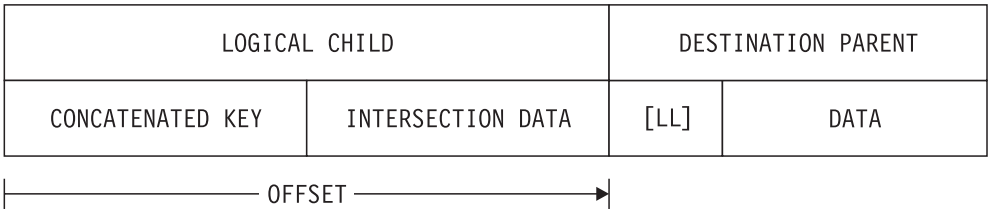
The LOCKED function activates the enqueing function that tells other online or MPS batch PSBs that are attempting to get into your PSBs data base to line up and wait. In order to avoid the delay that occurs when the other PSBs have to wait, it is recommended that you issue a TERMINATE command (or a CHECKPOINT command in MPS batch) as soon as possible to end the data base LOCKED function.

You should also be aware that DL/I invokes a lockout function when enqueing is active. The lockout function will detect when your PSB is LOCKED on data base B and you are waiting to get into data base A while another PSB (in an MPS batch or CICS/VS online environment) is LOCKED on data base A and is waiting to get into data base B. In this instance, each PSB would be enqueued and holding its own data base while waiting for the other PSB to terminate and release its data base. Lockout will cause the last PSB enqueued to terminate and an appropriate message to be returned to the user of the terminated job.

## 10. OFFSET

A logical data structure can be constructed from two or more physical data bases. In this case, the physical data bases are linked together into a logical data base through concatenated segments. A concatenated segment consists of two related segment types from two different physical data bases. The first of these segment types is called a logical child segment and the second a destination parent segment.

Whenever a logical child segment is accessed in a logical data base, its data is concatenated with the data from its destination parent segment in a single segment I/O area, like this:




where LL is the length field of the destination parent segment if it is a variable length segment.

When data is transferred for a concatenated segment with a variable length destination parent, you must specify in the command the offset to the destination parent within the I/O area. The offset is the combined length of the logical child's concatenated key and intersection data. This is required to allow DL/I to locate the length field (LL) to be used in calculating the actual length of the concatenated segment. You code the OFFSET option after the INTO or FROM for the concatenated segment, like this:

```
OFFSET(expression)
```

“expression” in COBOL is any valid integer variable or integer constant.
“expression” in PL/I is any valid expression that converts to the integer data type.

The value of “expression” must equal the combined length of the logical child's concatenated key and intersection data—thus pointing to the LL field of the destination parent.

OFFSET is to be coded only when VARIABLE and either INTO or FROM are coded for the associated segment. OFFSET must be coded after the associated SEGMENT option.

## 11. Specifying the PSB

In an online application program, the Program Specification Block (PSB) to be used for subsequent data base operations must be scheduled to give you access to the data base. You identify the PSB by naming it in the PSB option after the SCHEDULE function in a SCHEDULE command. It looks like this:

```
PSB(name)
```

where “name” is the name of a PSB defined during PSB generation. It must not be a variable. Because it is a constant, it need not be enclosed in quotes.

## 12. Specifying the Checkpoint Id

When you use the CHECKPOINT command to cause DL/I to commit all data base changes made to that point by your application program, you can specify, with the ID argument, the checkpoint identifier that you want associated with that checkpoint.

You code the checkpoint identifier like this:

```
ID(expression)
```

where “expression” is any eight-character value enclosed in single quotes, or the name of an eight-character value defined in your program.

---

# Syntax of the Command Language

This section provides, in Figure 2-4 on page 2-20, a chart summarizing the syntax of all the DL/I HLPI commands. The information for each command appears in a row that extends across both pages. The three GET commands are grouped together at the top; the remaining data base function commands are grouped next; and the CHECKPOINT, SCHEDULE, and TERMINATE commands are grouped at the bottom.

The syntax for each individual command appears under the name of that command in Chapter 4, with any notes that apply to that command. Notes that apply in general appear below.

***Notes:***

1. Wherever the word “name” appears in the syntax of a command, the following rules apply:
   - You must code an identifier consisting of alphabetic and numeric characters only, with the first character being alphabetic; or a literal string constant enclosed in single quotes. Both are interpreted as being a literal string.
   - A variable can be specified for “name” only on a LOAD command. In this case, the identifier must be enclosed in a double set of parentheses to

indicate that it represents a variable rather than a constant. For example: SEGMENT((VARNAME)).

2. Wherever the word “expression” (abbreviated as “exp”) appears in the syntax of a command (except the CHECKPOINT command), it is to be replaced by an expression in the host language or by a positive integer value itself. “expression” in COBOL is any valid integer variable or integer constant. “expression” in PL/I is any valid expression that converts to the integer data type. If a variable is specified for “expression,” it should be declared as halfword binary.

   For CHECKPOINT, “expression” should convert to an eight-byte character string.

3. Wherever the word “reference” (abbreviated as “ref”) appears in the syntax of a command, it is to be replaced by a name previously defined in the host language of the application program.

4. The following examples illustrate the acceptable methods of specifying arguments for the various command options.

   - SEGMENT(name)

     For GET, INSERT, REPLACE, DELETE, and LOAD:

     ```
     SEGMENT(ARTIST)
     ```

     *or*

     ```
     SEGMENT('ARTIST')
     ```

     where ARTIST is the name of a segment type defined to DL/I during data base generation. It is not defined in the application program.

     For LOAD only, this additional form is allowed:

     ```
     SEGMENT((SEGNAME))
     ```

     where SEGNAME is defined in the application program as either

     ```
     77 SEGNAME PIC X(8) VALUE 'ARTIST'.      (COBOL)
     ```

     *or*

     ```
     DECLARE SEGNAME CHAR(8) INIT('ARTIST');  (PL/I)
     ```

   - INTO(reference) or FROM(reference)

     For GET commands:

     ```
     INTO(INAREA)
     ```

     where INAREA is defined in the application program as either

     ```
     77 INAREA PIC X(nn).      (COBOL)
     ```

     *or*

     ```
     DECLARE INAREA CHAR(nn);  (PL/I)
     ```

     For INSERT, REPLACE, DELETE, and LOAD:

     ```
     FROM(OUTAREA)
     ```

     where OUTAREA is defined in the application program in the same way as INAREA.

     nn is equal to the length of the segment to be processed.

- SEGLENGTH(exp), FIELDLENGTH(exp), OFFSET(exp), PCB(exp), or FEEDBACKLEN(exp)

  For data base function commands:

  ```
  SEGLENGTH(4)
  FIELDLENGTH(4)
  OFFSET(4)
  PCB(4)
  FEEDBACKLEN(4)
  ```

  *or*

  ```
  SEGLENGTH(NUM)
  FIELDLENGTH(NUM)
  OFFSET(NUM)
  PCB(NUM)
  FEEDBACKLEN(NUM)
  ```

  where NUM is defined in the application program as either

  ```
  77 NUM COMP PIC S9999 VALUE IS +4.  (COBOL)
  ```

  *or*

  ```
  DECLARE NUM FIXED BIN(15) INIT(4);  (PL/I)
  ```

- WHERE(name op reference {AND|OR}name op reference“...)
  FIELDLENGTH(expression ,expression“...)“

  For GET and INSERT commands:

  ```
  WHERE (TYPENO<C800 AND TYPENO>C700 AND DESCR=JOBS
  OR MONTHS>C100) FIELDLENGTH (3,3,8,3)
  ```

  where TYPENO, DESCR, and MONTHS are the names of fields defined to DL/I during data base generation. They are not defined in the application program.

  C800, C700, JOBS, and C100 are defined in the application program as either:

  ```
  (COBOL)
        77  C800   PIC S999 COMP   VALUE IS 800.
        77  C700   PIC S999 COMP   VALUE IS 700.
        77  C100   PIC S999 COMP   VALUE IS 100.
        77  JOBS   PIC X(nn)       VALUE 'COMIC'.
  ```

  *or*

  ```
  (PL/I)
        DECLARE  C800   FIXED BIN(11) INIT (800);
        DECLARE  C700   FIXED BIN(11) INIT (700);
        DECLARE  C100   FIXED BIN(11) INIT (100);
        DECLARE  JOBS   CHAR(nn)      INIT ('COMIC');
  ```

  nn is equal to the length of the field as defined to DL/I. In the above example, “(nn)” is “(8).”

  FIELDLENGTH is a one-for-one integer description of the length of field of C800, C700, JOBS, and C100.

- PSB(name)

  For the SCHEDULE command:

  ```
  PSB(PSB01)
  ```

  *or*

  ```
  PSB('PSB01')
  ```

  where PSB01 is the name of a PSB defined to DL/I during PSB generation.

- ID(expression)

  For the CHECKPOINT command:

  ```
  ID(CHKPID)
  ```

  *or*

  ```
  ID('CHKP1000')
  ```

  where CHKPID is defined in the application program as either

  ```
  77 CHKPID PIC X(8) VALUE 'CHKP1000'.     (COBOL)
  ```

  *or*

  ```
  DECLARE CHKPID CHAR(8) INIT('CHKP1000'); (PL/I)
  ```

- KEYFEEDBACK(reference)

  Using the KEYFEEDBACK keyword:

  ```
  KEYFEEDBACK(SKILLKEY)
  ```

  where SKILLKEY is defined in the application program as either:

  ```
  01  SKILLKEY PIC X(nn).          (COBOL)
  ```

  *or*

  ```
  DCL SKILLKEY CHAR(nn);           (PL/I)
  ```

  nn is less than or equal to the length of the concatenated key as defined to DL/I.

| Trigger | Function | PCB Option | Key Feedback Option |
|---|---|---|---|
| {EXECUTE} DLI<br>{EXEC } | {GET NEXT}<br>{GN } | [USING PCB(exp)] | [KEYFEEDBACK(ref) [FEEDBACKLEN(exp)]] |
| {EXECUTE} DLI<br>{EXEC } | {GET NEXT IN PARENT}<br>{GNP } | | [KEYFEEDBACK(ref) [FEEDBACKLEN(exp)]] |
| {EXECUTE} DLI<br>{EXEC } | {GET UNIQUE}<br>{GU } | [USING PCB(exp)] | [KEYFEEDBACK(ref) [FEEDBACKLEN(exp)]] |
| | | | |
| {EXECUTE} DLI<br>{EXEC } | {INSERT}<br>{ISRT } | [USING PCB(exp)] | |
| {EXECUTE} DLI<br>{EXEC } | {REPLACE}<br>{REPL } | [USING PCB(exp)] | |
| {EXECUTE} DLI<br>{EXEC } | {DELETE}<br>{DLET } | [USING PCB(exp)] | |
| {EXECUTE} DLI<br>{EXEC } | LOAD | [USING PCB(exp)] | |
| | | | |
| {EXECUTE} DLI<br>{EXEC } | {CHECKPOINT}<br>{CHKP } | | |
| {EXECUTE} DLI<br>{EXEC } | {SCHEDULE}<br>{SCHD } | | |
| {EXECUTE} DLI<br>{EXEC } | {TERMINATE}<br>{TERM } | | |

*Figure 2-4 (Part 1 of 2). Syntax Summary Chart*

| Parent Segment(s) (0-14) | Object Segment | Delimiter |
|---|---|---|
| [[{FIRST}][VARIABLE] SEGMENT(name)<br>{LAST }<br>[INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]]<br>[WHERE(name op ref[{AND}name op ref]...)<br>{OR }<br>[FIELDLENGTH(exp[,exp]...)]]]... | [[{FIRST}][VARIABLE] SEGMENT(name)]<br>{LAST}<br>INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]<br>[WHERE(name op ref[{AND}name op ref]...)<br>{OR }<br>[FIELDLENGTH(exp[,exp]...)]]... | {END-EXEC}<br>{; } |
| [[{FIRST}][VARIABLE] SEGMENT(name)<br>{LAST }<br>[INTO(ref) [LOCKED][OFFSET(exp)][SEGLENGTH(exp)]]<br>[WHERE(name op ref[{AND}name op ref]...)<br>{OR }<br>[FIELDLENGTH(exp[,exp]...)]]]... | [[{FIRST}][VARIABLE] SEGMENT(name)]<br>{LAST }<br>INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]<br>[WHERE(name op ref[{AND}name op ref]...)<br>{OR }<br>[FIELDLENGTH(exp[,exp]...)]]... | {END-EXEC}<br>{; } |
| [[LAST][VARIABLE] SEGMENT(name)<br>[INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]]<br>[WHERE(name op ref[{AND}name op ref]...)<br>{OR }<br>[FIELDLENGTH(exp[,exp]...)]]]... | [LAST][VARIABLE] SEGMENT(name)<br>INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]<br>[WHERE(name op ref[{AND}name op ref]...)<br>{OR }<br>[FIELDLENGTH(exp[,exp]...)]]... | {END-EXEC}<br>{; } |
| | | |
| [[{FIRST}][VARIABLE] SEGMENT(name)<br>{LAST }<br>[FROM(ref) [SEGLENGTH(exp)]]<br>[WHERE(name op ref[{AND}name op ref]...)<br>{OR }<br>[FIELDLENGTH(exp[,exp]...)]]]... | [{FIRST}][VARIABLE] SEGMENT(name)<br>{LAST }<br>FROM(ref)[OFFSET(exp)][SEGLENGTH(exp)] | {END-EXEC}<br>{; } |
| [[VARIABLE] SEGMENT(name)<br>FROM(ref)[OFFSET(exp)][SEGLENGTH(exp)]]... | [VARIABLE] SEGMENT(name)<br>FROM(ref)[OFFSET(exp)][SEGLENGTH(exp)] | {END-EXEC}<br>{; } |
| | [VARIABLE] SEGMENT(name)<br>FROM(ref)[SEGLENGTH(exp)] | {END-EXEC}<br>{; } |
| | [VARIABLE] SEGMENT(name)<br>FROM(ref)[SEGLENGTH(exp)] | {END-EXEC}<br>{; } |
| | | |
| ID(exp) | | {END-EXEC}<br>{; } |
| PSB(name) | | {END-EXEC}<br>{; } |
| | | {END-EXEC}<br>{; } |

*Figure 2-4 (Part 2 of 2). Syntax Summary Chart*

# Chapter 3. DL/I Application Program

The material in this chapter is designed to help you through the process of creating an application program that uses DL/I to access data bases. The chapter is divided into four major sections that correspond to phases in the production of a program:

- planning
- writing
- executing
- debugging

## Planning Your Program

### A Checklist

Things you will have to consider include:

- The functions your program is to perform and the data with which it will work.
- The names of the segments and fields that you will be using in your program.
- How many PCBs you will be using.
- The name of the PSB you will use. If you are writing an online program, there may be more than one.
- The segments that your program is sensitive to that are of variable length.
- The access method you will be using for each data base, because there are certain access method restrictions and requirements.

Most of this information will be obtained from those responsible for data base administration in your installation. It is important that DBA be involved in the planning stage of your application program. In addition to providing you with information, it is a function of DBA to perform all the steps necessary to prepare DL/I to work properly with your program. DBA will also have to be involved in making the decisions concerning the possible use of other DL/I functions, such as multiple positioning, as discussed later in this chapter, in the section “Other Available DL/I Functions.”

### General Considerations and Restrictions

#### Data Base Processing Methods

Data Base Administration will provide you with information on how your data base can be processed.

Sequential Processing Only (HSAM)

- Root segments are stored in ascending order.

- Direct processing of root segments is possible, but expensive in terms of resource consumption, and should be avoided.

Sequential or Direct Processing (HISAM and HIDAM)

- Sequential, skip sequential, or direct processing of root segments can all be done with good performance.

Direct Processing Only (HDAM)

- Root segments are stored randomly.
- Direct processing of root segments is very efficient.
- Sequential processing of root segments in key sequence order *cannot* be done.
- Root segments can, however, be processed sequentially in the order in which they are physically stored (which is different from key sequence).

## Data Base Processing Considerations

Considerations that you should take into account in your planning, in respect to each one of the major DL/I functions, are described here under the heading of the function.

## Loading

After a data base has been created as a DBA function, it must be loaded with the initial data before it can be used. This loading is done by a batch application program that uses the DL/I HLPI LOAD command to take data from the designated segment I/O areas, after it has been built there, and load it into the data base in the correct location. No DL/I HLPI commands other than LOAD can be used in this program. The PSB for this program must have been generated with PROCOPT=L. The LOAD command cannot be used in MPS batch and online programs.

If the access method specified for this data base is simple HSAM, HSAM, simple HISAM, HISAM, or HIDAM; you must presort the data base records in ascending order of the value of the key field of the root segments, and load them into the data base in this order. If the access method is HSAM, HISAM, or HIDAM, and the data base record is composed of more than the root segment, you must see that all of the segments within the data base record have been presorted in order by their hierarchical relationship and the value of their key fields, so that they can be loaded into the data base in their hierarchical order. If the access method is HDAM, presorting of root segments is optional, but dependent segments do have to be in hierarchical order.

If the segments to be loaded have been sorted into hierarchical sequence, the load program can:

- Read a segment to be loaded into a segment data area.
- Determine the name of the segment.
- Execute a LOAD command specifying the segment name and data area.
- Check the status code.
- If no error, continue the process until all segments are loaded.

Alternatively, once the name of the segment has been determined, the load program could:

- Move the segment name into a variable defined in the program.

- Execute a LOAD command specifying the variable name rather than the segment name in the SEGMENT option, as in this example:

  ```
  EXECUTE DLI LOAD SEGMENT((SEGNAME)) FROM(SEGDATA);
  ```

  where SEGNAME is the variable (identified by double parentheses) containing the name of the segment to be loaded. The area identified by FROM must be at least as large as the largest segment to be loaded from it.

- Continue as specified above.

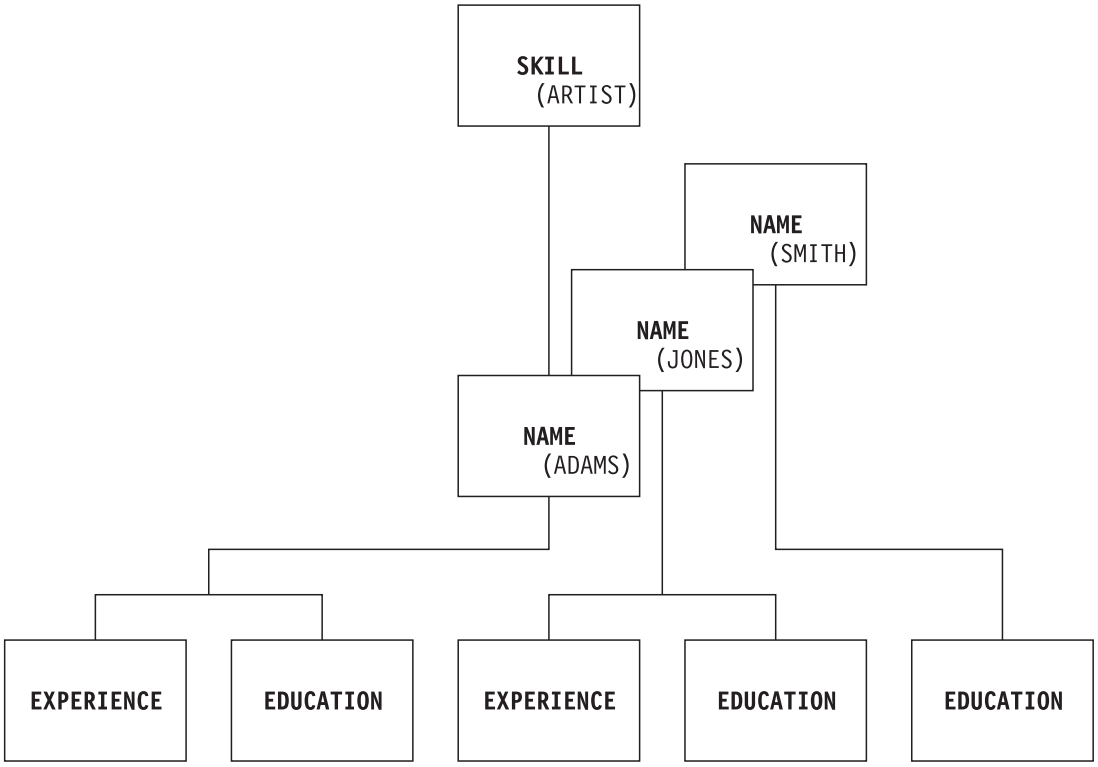



*Figure 3-1. Logical Data Base Record Structure*

## Retrieving

You must be able to retrieve from a data base any segment to which your application program is sensitive. The DL/I HLPI provides three command functions for you to use in performing retrieval. They are: GET UNIQUE, GET NEXT, and GET NEXT IN PARENT. GET UNIQUE is for use when you want to retrieve a specific segment by direct reference into the data base. When you want to retrieve segments sequentially, GET NEXT provides that function. Usually, you would want to use GET NEXT after a previous GET UNIQUE or GET NEXT has established position within the data base. However, GET NEXT can be used without position having been previously established. In this case, the GET NEXT request is satisfied by proceeding from the beginning of the data base. The GET NEXT IN PARENT command makes it possible for you to sequentially retrieve all segments subordinate to a chosen parent segment. For example, from the skills inventory data base shown in Figure 3-1, you could retrieve all experience and education segments for a given skill code and employee name. Parentage must have been previously established to a unique name segment by a previous GET UNIQUE or GET NEXT command. Once all the experience and education segments for the given skill code and employee name have been retrieved by a series of GET NEXT

IN PARENT commands, execution of another GET NEXT IN PARENT will result in status code GE being returned to your program. GE indicates that all segments subordinate to the given skill code and employee name have been retrieved.

In addition to directly retrieving unique segments and sequentially retrieving segments, you can also skip sequentially from one segment to another of the same type. For instance, assume that you need to retrieve all name segments for a particular skill segment. It is not necessary to retrieve the segments subordinate to each name segment (experience and education). The procedure is to retrieve the first name segment with a GET UNIQUE command that looks like this in a PL/I program (assuming that “SKILVAR” is defined in your program with the value ‘ARTIST’):

```
EXEC DLI GET UNIQUE SEGMENT(SKILL) WHERE(SKILCODE=SKILVAR)
         SEGMENT(NAME) INTO(NAMEIO);
```

Then, looping through the following GET NEXT command will retrieve all NAME segments whose skill is artist.

```
EXEC DLI GET NEXT SEGMENT(SKILL) WHERE(SKILCODE=SKILVAR)
         SEGMENT(NAME) INTO(NAMEIO);
```

Executing this command after the last NAME segment under ARTIST has been retrieved will result in status code GE.

The WHERE clauses above can be expanded using the Boolean AND and OR operators. This allows for more specific segment selectivity. Up to 11 of these operators may be used in any single WHERE clause. This provides for up to 12 qualification conditions. If COBOL is the programming language, then the WHERE clause must be followed by a FIELDLENGTH clause specifying the length of field in each “reference” in the WHERE clause. If PL/I is used, the FIELDLENGTH clause is optional. See “Qualified Segment Selection” in Chapter 2 of this manual for the use of Boolean operators in the WHERE clause.

```
EXEC DLI GET NEXT SEGMENT(SKILL)
         WHERE (SKILCODE>SKILVAR OR SKILCODE<SKILPROG
             AND SKILEVEL=LVLOFSKL)
         FIELDLENGTH(10,10,6)
         SEGMENT(NAME) INTO(NAMEI0);
```

In this example, ARTIST is resident in the variable 10 byte field SKILVAR, PROGRAMMER is resident in the variable 10 byte field SKILPROG, and EXPERT is resident in the variable 6 byte field LVLOFSKL.

## Updating

You must be able to modify the data contained in the data base in a given segment. The DL/I HLPI command REPLACE provides this function. When you code a REPLACE command in your program, you must be sure that one of the three types of GET command is executed, to retrieve the segment that you intend to update, before the REPLACE is executed. There can be no intervening commands of any type, using the same PCB, to this data base. If there has been an intervening command, the REPLACE command will be rejected. The key field, if any, of the segment to be updated *must not* be modified.

As an example of updating, let's change the data in the skill segment of ARTIST from COMMERCIAL to COMMERCIAL CARTOON. The GET command to retrieve the ARTIST segment would look like this in a PL/I program (assuming that SKILVAR is defined in your program with the value ‘ARTIST’):

```
EXEC DLI GET UNIQUE SEGMENT(SKILL) INTO(SKILLIO)
         WHERE(SKILCODE=SKILVAR);
```

The I/O area named SKILLIO will then contain the contents of the SKILL segment:

| | |
|---|---|
| ARTIST | COMMERCIAL |

Your program can now modify the data in the I/O area to look like this:

| | |
|---|---|
| ARTIST | COMMERCIAL CARTOON |

The command to replace the SKILL segment in the data base with the new data looks like:

```
EXEC DLI REPLACE SEGMENT(SKILL) FROM(SKILLIO);
```

Note that HSAM data bases can only be updated by copying the entire data base from the input file to the output file; omitting, modifying, or inserting segments as required. The PSB for such a program must have two PCBs; one with PROCOPT=G for reading the input file, and one with PROCOPT=L or LS for the output file.

## Deleting

You must be able to delete an entire segment from the data base. The DL/I HLPI command DELETE provides this function. Before a DELETE command, you must code one of the three types of GET command, to retrieve the segment you intend to delete. There can be no intervening command of any type, using the same PCB, to this data base. If there has been an intervening command, the DELETE command will be rejected.

It is important for you to remember that the deletion of a parent segment results in the deletion of *all* segments physically subordinate to the deleted segment, whether your program is sensitive to them or not.

If a GET UNIQUE command tries to retrieve a particular segment immediately after it was deleted, a status code GE is returned, indicating that the segment was not found.

As an example, let's delete the SKILL segment containing the key and data fields for artist. The command to retrieve the segment looks like this in a PL/I program:

```
EXEC DLI GET UNIQUE SEGMENT(SKILL) INTO(SKILLIO)
         WHERE(SKILCODE=SKILVAR);
```

The SKILLIO area will contain:

| | |
|---|---|
| ARTIST | COMMERCIAL CARTOON |

Now, when the command

```
EXEC DLI DELETE SEGMENT(SKILL) FROM(SKILLIO);
```

is executed, the ARTIST segment and all its dependent segments will be deleted. That means that NAME segment ADAMS, EXPERIEN segment ADAMS, and EDUCAT segment ADAMS are deleted; as well as NAME segment JONES and its dependent EXPERIEN and EDUCAT segments, and NAME segment SMITH and its

dependent EDUCAT segment. Notice that deleting one SKILL segment deleted every segment shown in Figure 3-1 on page 3-3.

### Inserting

You must be able to insert a new segment into an existing data base. The DL/I HLPI command INSERT provides this function.

Remember that you cannot add a dependent segment unless all parent segments in the complete hierarchical path to it already exist in the data base. For example, in Figure 3-1 on page 3-3, no EXPERIEN segment subordinate to a particular NAME segment can be added to the data base unless the NAME segment already exists. Violation of this rule results in a GE (segment not found) status code.

The key value of a segment determines where it will be inserted in the data base. Segments without key fields are inserted according to a rule established by DBA when the DBD for the data base was defined. The possible rules are: FIRST, LAST, and HERE. Segment types with non-unique keys (the same key value occurring in more than one segment of the same type) are also inserted according to a FIRST, LAST, or HERE rule established by DBA when the DBD was defined. These rules are discussed in *DL/I DOS/VS Data Base Administration.*

Further information on rules governing the use of the INSERT command is provided under “INSERT” in Chapter 4.

## Restrictions

***On Use of COMREG:*** Because bytes l6 through l9 of the communication region are used by DL/I, you must not use them in your application program.

***On Use of Overlay Programming:*** DL/I does not support the use of overlay structures for application programs executing under its control. Although the COBOL SORT verb automatically produces an overlay structure, this restriction does not apply if the job control statements used to translate, compile, and link-edit your program are as shown in the COBOL example for a batch or MPS batch program in the section “Compilation and Link-Editing” in this chapter.

The use of “PL/I-SORT” programs, using the sort program product, is not affected by this restriction, provided that an overlay structure is not explicitly specified.

***On Use of Set Exit Abnormal (STXIT AB) Linkage:*** For batch applications you have the option, through the use of the user program switch indicator (UPSI), of permitting STXIT AB linkage to pass control to DL/I prior to abnormal termination, so that a controlled shutdown can occur. The DL/I system log and DL/I data bases are closed and a storage dump is provided. However, non-DL/I files are not closed; that is your responsibility.

If your COBOL application program is executing under DL/I control, any attempt by your program to execute the COBOL debug function can cause unpredictable results. Therefore, you should not use any COBOL debug function (any COBOL option that makes use of a STXIT routine) if DL/I STXIT is used. Refer to your COBOL publications for options that use STXIT linkages.

***On Mixed Use of Interfaces:*** Two DL/I interfaces are available for use in writing application programs: the DL/I High Level Programming Interface and the DL/I CALL interface. Either can be used in an application program written for the batch,

MPS batch, or CICS/VS online environment. Also, within the online environment, either interface can be used with statements in the CICS/VS macro level or the CICS/VS command level interface. However, statements in the two DL/I interfaces must not both appear in the same program; nor should both interfaces be used in different programs that exchange control during the life of a given task. This means that if control is passed to programs other than the one that initially acquired the PSB, these programs should access the DL/I data base using the same interface as the original program.

***On Host Language Use and Features:*** Certain features of the PL/I Optimizer and ANS COBOL languages can not be used in application programs designed for use with CICS/VS. These restrictions and other techniques are described in the chapter “Programming Techniques and Restrictions” in the *CICS/VS Application Programmer's Reference Manual (Command Level).*.

***On Use of Reserved Keywords and Labels:*** Two pairs of keywords are reserved for use as triggers by the translator and can be used only within DL/I HLPI commands. They are:

- EXECUTE DLI
- EXEC DLI

The translator generates labels for the DIB, DL/I default values, and its own internal variables. To avoid creating duplicately defined symbols, you must not define the following:

- Labels beginning with the characters “DLZ”
- Labels beginning with the characters “DIB”
- Labels beginning with the characters “DFH”

The *CICS/VS Application Programmer's Reference Manual (Command Level)* lists CICS reserved labels.

## Online Considerations and Restrictions

You must be familiar with Customer Information Control System/Virtual Storage (CICS/VS) programming fundamentals before attempting to plan an online DL/I application program. The prerequisite CICS/VS publications are listed in the preface of this manual.

DL/I data bases are accessed in a CICS/VS online environment using the same DL/I HLPI commands as in the batch and MPS batch environments.

## MPS Batch Considerations and Restrictions

Before planning an application program to be run in a Multiple Partition Support (MPS) environment, you should be familiar with the MPS considerations and restrictions described in *DL/I DOS/VS Data Base Administration.* Also, when using CICS/VS Intersystem Communication support, DL/I application programs can access a data base that is resident on another CICS/VS system. The applicaiton program, except in the following situation, need not be aware of where the data base is located. If your MPS batch application program is to run on a system where Intersystem Communication support is active, it must not issue SCHEDULE commands. Specific considerations and restrictions applying to the writing of your MPS batch program are listed under “MPS Considerations” in the section “Writing Your Program” later in this chapter.

## DL/I Programming Techniques and Suggestions

The following items are programming techniques and suggestions that may be of help in the planning stage of creating your DL/I application program.

1. As far as possible, try to construct the logic of your program in a manner that is not highly dependent on the hierarchical structure of the data base.
2. Remember that deletion of a parent segment also deletes all of its children with the same command, even though your program may not be sensitive to them. If any information is required from those children, it must be retrieved from them before the parent is deleted. DELETE is the only command that can affect multiple segments without specifying parent segments.
3. The LOAD command is used to initially load a data base, and is used only for that purpose. The “loading” of additional segment types in an existing data base is done through the INSERT command. All programs after the first are actually add-type programs and their planning and use should be coordinated and reviewed by DBA to ensure that they perform adequately.

## Error Checking

The CICS/VS EXEC translator scans each of the DL/I HLPI commands in your program and reports any syntax errors. Your programming productivity is increased through the detection of these errors before you actually compile your program.

During execution of your program, DL/I checks for errors that cannot be detected by the translator and, depending on their type, reports them in the DIB as status codes, or as system messages.

Errors detected by DL/I and reported as status codes are handled in two different ways. The first group is returned to you in the status code field of the DIB (DIBSTAT). You test for them in your program with code immediately following each DL/I HLPI command and handle them in your program or call a generalized handling routine, depending on the procedures established for your installation.

The other status code errors cannot be readily corrected under program control. In a CICS/VS environment, they will result in a task ABEND. In the batch and MPS batch environments, they will result in a program ABEND.

In the CICS/VS online environment, you can intercept status code abends by coding EXEC CICS HANDLE ABEND at the beginning of your program. You can access the abend code in your ABEND exit routine by coding EXEC CICS ASSIGN ABCODE. The ABEND code that indicates HLPI status code abends is DHxx, where “xx” is the corresponding DL/I status code. Your ABEND exit routine can then attempt to correct the error and continue processing. Abends cannot be intercepted in the batch and MPS batch environments.

## Writing Your Program

This section will help you in the actual writing of your DL/I application program. In general, the information applies to all of the possible operating environments. Where this is not true, the exceptions are noted.

The detailed description of how to code and use each of the DL/I HLPI commands, including the command syntax, is found in Chapter 4.

## Entry to Batch and MPS Batch Programs

In the batch and MPS batch environments, when VSE gives control to DL/I, the DL/I control program passes control on to your application program through an entry point defined in your program. The method of defining the entry point is different for each host language.

*COBOL:* The following statement must be the first in the procedure division of the main procedure:

```
ENTRY 'DLITCBL'.
```

*PL/I:* There is no special requirement.

## DIB

Each time your program executes a DL/I HLPI command, DL/I returns a status code, and other information, to your program through the DL/I Interface Block (DIB). The status code should be checked to ensure that the function was performed as expected.

There is one DIB provided for each external procedure. The contents of the DIB, at any given moment, reflects the status of the last DL/I HLPI command executed in that procedure. DIB information required by an external procedure that has not issued a DL/I HLPI command must be passed to that procedure by your application program.

Labels that you can use to access the variables in the DIB are automatically generated in your program by the translator. (These labels are reserved and you must not redefine them in your program.)

The way the DIB variables are defined for each host language is shown below.

For COBOL:

```
DIBVER     PICTURE X(2)
DIBSTAT    PICTURE X(2)
DIBSEGM    PICTURE X(8)
DIBFLAG    PICTURE X(1)
DIBSEGLV   PICTURE X(2)
DIBKFBL    PIC S9(4) COMP
```

For PL/I:

```
DIBVER    CHAR(2)
DIBSTAT   CHAR(2)
DIBSEGM   CHAR(8)
DIBFLAG   CHAR(1)
DIBSEGLV  CHAR(2)
DIBKFBL   FIXED BIN(15)
```

- DIBVER is the version of the translator used to translate the application program.
- DIBSTAT is the DL/I status code.
- DIBSEGM is the name of the lowest level segment retrieved. DIBSEGM should be ignored following a CHECKPOINT, SCHEDULE, or TERMINATE command.
- DIBFLAG is a flag indicating that an online task had to wait for a resource owned by an MPS batch task (DIBFLAG = X‘FF’).

- DIBSEGLV is the hierarchical level of the lowest level segment retrieved. DIBSEGLV should be ignored following a CHECKPOINT, SCHEDULE, or TERMINATE command.
- DIBKFBL is the actual length of the concatenated key in the PCB when KEYFEEDBACK is specified on the DL/I command.

## Status Codes

After processing a given DL/I HLPI command, control is returned to your application program at the next sequential instruction following the command. So that you can check that the command has completed successfully, and has performed the operation you intended on the data you specified, DL/I returns a two-character status code in the DIB, as mentioned under “DIB” above.

The first thing that you should do in your program after each command is to test DIBSTAT for the various status codes and take appropriate action. The status codes that could be returned in DIBSTAT are GA, GB, GE, GK, II, LB, NE, TG, and ƀƀ.

The status codes that are returned for each command are listed with the individual commands in Chapter 4. A complete list of status codes is shown as a table in Figure 3-5 on page 3-18, later in this chapter.

## Using DIBKFBL

If DIBKFBL is greater than the FEEDBACKLEN parameter, the concatenated key is truncated when it is moved into the user area. However, under PL/I, FEEDBACKLEN can be allowed to default to the length of the KEYFEEDBACK area.

## Obtaining the PSB (Online Only)

The following information applies only in the CICS/VS online environment.

### SCHEDULE command

Before any DL/I data bases can be accessed in an online program, your program must initiate the scheduling of a PSB. You do this with the DL/I HLPI SCHEDULE command.

## Releasing the PSB (Online Only)

The following information applies only in the CICS/VS online environment.

### TERMINATE Command

You use the DL/I HLPI TERMINATE command to indicate to DL/I that all modifications made to the data bases by the transaction to this point are committed and cannot be backed out, and that you are releasing the PSB for use by another task.

# Terminating the Program

## Batch and MPS Batch

In the batch and MPS batch environments, at the completion of the execution of your program, control must be passed back to DL/I. This is done by coding an exit statement in your program that will look like this, depending on the host language you are using:

COBOL

```
GOBACK.
```

PL/I

```
RETURN;
```

The GOBACK or RETURN statement will return control to DL/I. After all DL/I resources are released and the data bases are closed, DL/I returns control to VSE.

**Note:** STOP RUN can not be used in a COBOL program as an exit statement, since control would not be returned to DL/I to allow it to release its resources and close the data bases and log. However, it can be coded after GOBACK. This prevents the compiler from giving a warning message and automatically generating a STOP RUN.

## Online

In the online environment, control is returned to DL/I by coding a CICS/VS command or macro statement, as shown below:

| | Command | Macro |
|---|---|---|
| COBOL | EXEC CICS RETURN END-EXEC. | DFHPC TYPE=RETURN |
| PL/I | EXEC CICS RETURN; | DFHPC TYPE=RETURN |

# Techniques and Suggestions

The following items are programming techniques and suggestions that may be of help in the writing stage of creating your DL/I application program.

1. In general, use qualified segments wherever possible.
2. Do not omit parent segments in a command specifying multiple parent segments if it can be avoided. This promotes flexibility and control as the application and the data base grow or change:
   - If the hierarchical structure of the data base is changed, your specification of all parent segments will ensure the integrity of your program's access to the newly structured data base.
   - If you add the use of the multiple positioning feature to your program at a later date, specification of all parent segments would be required then.
   - Complete segment specification is a sound programming practice from a documentation and debugging point of view.
3. It is possible to specify segments in a GET NEXT command in such a way that DL/I would be forced to search to the end of the data base without retrieving any segment. This is especially likely during program testing. If a large, multivolume data base is being accessed, significant amounts of processing time could be wasted. To prevent this possibility, use GET UNIQUE or GET NEXT IN PARENT commands wherever practical.

4. When specifying multiple parent segments in a command, try to specify the root segment, qualified on the key field and using the “equal” operator, wherever possible. This will prevent unnecessary and time consuming searching of the entire data base. The same applies when secondary indexing is used, except that qualification should be on the indexed field instead of the key field.
5. The use of GET UNIQUE commands (rather than GET NEXT) is likely to provide more flexibility for future application program and data base changes.
6. The GET UNIQUE command can be used to process a data base sequentially, although it usually requires more processing time than comparable GET NEXT commands.
7. A repeated GET NEXT command with one unqualified segment retrieves all occurrences of that segment type in the data base until the end is reached.
8. Remember that parentage cannot be set or reset with a GET NEXT IN PARENT command.
9. Any field defined to DL/I can be used for segment selection. However, qualification of root segments with non-key fields should be avoided for performance reasons since the data base must be scanned sequentially to satisfy these requests.
10. After an unsuccessful GET NEXT command, the current position depends on several factors. It is good practice to establish a known position with a GET UNIQUE command in this case.
11. It is also good practice to include the SEGMENT option for intermediate levels when using GET UNIQUE commands.
12. Of all the commands that should be fully qualified, it is most important with the INSERT command. If not fully qualified, the INSERT command could insert the segment in a position different from the one that you intended.

## MPS Batch Considerations

If any online tasks must wait for a resource owned by an MPS batch task, the MPS task will be informed of this fact on the next and all subsequent commands until a DL/I checkpoint is executed. This condition is flagged by the setting of the DIB flag byte (DIBFLAG) to X‘FF’. (Note that making the resource available through an action, other than a checkpoint, does not change the flag byte. The flag indicates that a wait was required at some point, not necessarily that a task is currently waiting.)

When using the MPS Restart facility, a VSE checkpoint must be coded before each DL/I CHECKPOINT command in the application program. The two checkpoints together are referred to as a “combined checkpoint.” No other DL/I commands may be issued between the VSE checkpoint and the DL/I CHECKPOINT command.

## Programming Examples

This section provides a number of examples showing how the DL/I HLPI commands are used to perform data base operations. Since it is assumed that you are familiar with the host language you have chosen for your application program, only those parts of the examples that illustrate DL/I functions are shown. Not all possible combinations of DL/I commands are included. Some of the examples are taken from, or based on, the sample programs distributed with DL/I. (These programs are described in the *DL/I DOS/VS Guide for New Users.*) Figure 3-2 on page 3-14 shows the physical and logical data bases involved.

Each example is shown as it would appear in each of the supported host languages.

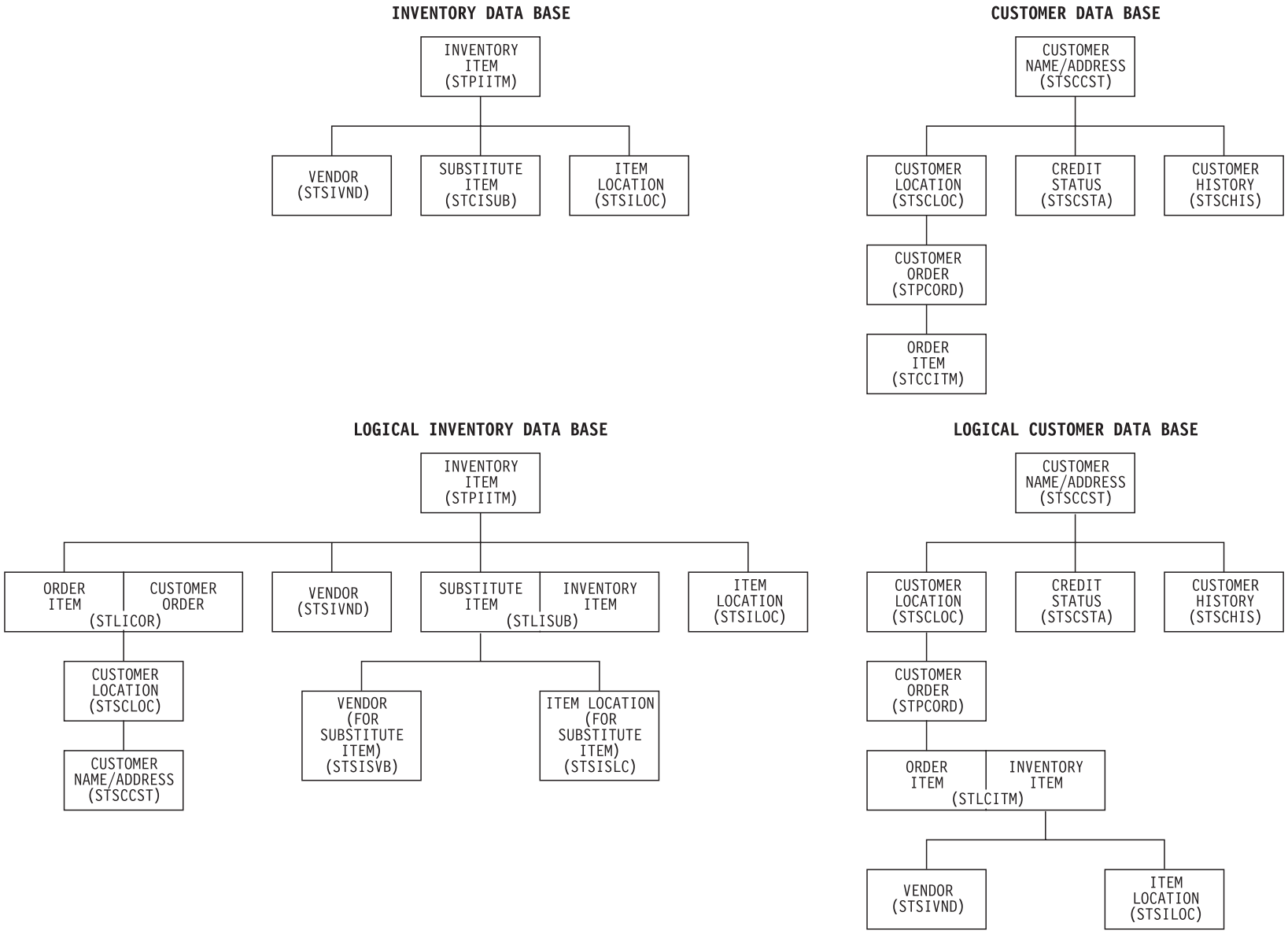



*Figure 3-2. Inventory and Customer Data Bases*

## LOAD Command Examples

The examples of the use of the DL/I HLPI LOAD command (Figure 3-3 on page 3-16 and Figure 3-4 on page 3-17) are taken from a batch program (DLZCBL10 for COBOL or DLZPLI10 for PL/I) that loads the inventory data base and the customer data base. The segments for these data bases are defined in DBDs named STDIDBP and STDCDBP, respectively. PCBs for the two data bases are in a PSB named STBICLD.

The 80-byte input data records for the two data bases are separated by a single record having “CUSTOMER” in the first eight bytes, like this:

```
 ....INVENTORY DATA....
 ....INVENTORY DATA....
        •
        •
CUSTOMER
 ....CUSTOMER DATA....
 ....CUSTOMER DATA....
        •
        •
```

```
*PROCESS XOPTS(DLI)...;
DLZPLI10: PROCEDURE OPTIONS(MAIN);
          •
          •
DCL   SW_CUSTOMER  FIXED BIN (8) INIT (0);
DCL   INV_REC      FIXED DEC (5) INIT (0);
DCL   CUST_REC     FIXED DEC (5) INIT (0);
DCL 1 SEG_NAME     CHAR (8) INIT ('        ');
DCL 1 WIO,
      3 IO_LEFT    CHAR (64),
      3 IO_RIGHT   CHAR (80);
DCL   IO           CHAR (144) DEFINED WIO POSITION (1);
DCL 1 IO_VAR,
      3 SEG_LGT    BIN (15,0),
      3 IO1        CHAR (128);
DCL 1 WMSGE_LINE,
      3 MSGE1      CHAR (6),
      3 MSGE2      CHAR (60);
          •
          •
SEG_LGT=MAX_LGT;                        /*SET MAXIMUM VARIABLE LENGTH */
DO;                                     /* LOAD DATA BASES            */
  IF SW_CUSTOMER = OFF THEN             /* IS THIS CUSTOMER DATA?     */
    DO;                                 /* NO.  LOAD INVENTORY DATA   */
      EXEC DLI                          /* LOAD COMMAND               */
           LOAD USING PCB(1)
           SEGMENT((SEG_NAME)) FROM(IO);
      IF DIBSTAT = '  ' THEN            /* WAS THERE A DL/I ERROR?    */
        INV_REC = INV_REC + ONE;        /* NO. COUNT INVENTORY RECORD */
      ELSE                              /* UNEXPECTED DL/I ERROR      */
        DO;
          MSGE1 = DIBSTAT;
          MSGE2 = 'UNEXPECTED DLI STATUS CODE-INV';
          CALL END_NOK;
        END;
    END;                                /* END -- LOAD INVENTORY DATA */
  ELSE
    DO;                                 /* LOAD CUSTOMER DATA         */
      IF SEG_NAME = 'STSCHIS' THEN      /* VARIABLE LENGTH SEGMENT?   */
        DO;                             /* YES.  PUT DATA IN I/O AREA */
          IO1 = SUBSTR(IO,1,SEG_LGT-TWO);
          EXEC DLI                      /* LOAD COMMAND               */
               LOAD USING PCB(2)
               VARIABLE SEGMENT(STSCHIS) FROM(IO_VAR);
        END;                            /* END -- VARIABLE SEGMENT    */
      ELSE                              /* NOT THE VARIABLE SEGMENT   */
        EXEC DLI                        /* LOAD COMMAND               */
             LOAD USING PCB(2)
             SEGMENT((SEG_NAME)) FROM(IO);
      IF DIBSTAT = '  ' THEN            /* WAS THERE A DL/I ERROR?    */
        CUST_REC = CUST_REC + ONE;      /* NO. COUNT CUSTOMER RECORD  */
      ELSE
        DO;                             /* UNEXPECTED DL/I ERROR      */
          MSGE1 = DIBSTAT;
          MSGE2 = 'UNEXPECTED DLI STATUS CODE-CUST';
          CALL END_NOK;
        END;
    END;                                /* END -- LOAD CUSTOMER DATA  */
END;                                    /* END -- LOAD DATA BASES     */
          •
          •
RETURN;
          •
          •
END_NOK: PROCEDURE;
          •                              /* DISPLAY FAILURE MESSAGE    */
          •
END;
END DLZPLI10;
```

*Figure 3-3. PL/I Batch Program Using LOAD Command*

```
CBL XOPTS(DLI)...
IDENTIFICATION DIVISION.
PROGRAM ID. DLZCBL10.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
          •
          •
77 SW-CUSTOMER          PIC 9 VALUE 0.
77 INV-REC              PIC 9(5) VALUE 0 COMP-3.
77 CUST-REC             PIC 9(5) VALUE 0 COMP-3.
77 SEG-NAME             PIC X(8).
01 IO.
   02 IO-LEFT           PIC X(64).
   02 IO-RIGHT          PIC X(80).
01 IO-VAR.
   02 SEG-LGT           PIC 9(4) COMP.
   02 IO1               PIC X(142).
01 MSGE-LINE.
   02 MSGE1             PIC X(6).
   02 MSGE2             PIC X(60).
          •
          •
PROCEDURE DIVISION.
ENTRY 'DLITCBL'.
          •
          •
     IF SW-CUSTOMER = 1
          GO TO LOAD-CUSTOMER-DB
     ELSE NEXT SENTENCE.
*
* LOAD INVENTORY DATA BASE
*
 LOAD-INVENTORY-DB.
     EXEC DLI
          LOAD USING PCB(1)
          SEGMENT((SEG-NAME)) FROM(IO) SEGLENGTH(144)
     END-EXEC.
     IF DIBSTAT = '  ' ADD 1 TO INV-REC
          GO TO READ-CARD
     ELSE MOVE DIBSTAT TO MSGE1
          MOVE 'UNEXPECTED DLI STATUS CODE-INV' TO MSGE2
          PERFORM PRINT-MESSAGE
          GO TO END-NOK.
*
* LOAD CUSTOMER DATA BASE
*
 LOAD-CUSTOMER-DB.
     IF SEG-NAME = 'STSCHIS'
         MOVE IO TO IO1
         EXEC DLI
              LOAD USING PCB(2)
              SEGMENT((SEG-NAME)) FROM(IO-VAR) SEGLENGTH(144)
         END-EXEC.
     ELSE
         EXEC DLI
              LOAD USING PCB(2)
              VARIABLE SEGMENT(STSCHIS) FROM(IO) SEGLENGTH(144)
         END-EXEC.
     IF DIBSTAT = '  ' ADD 1 TO CUST-REC
         GO TO READ-CARD
     ELSE MOVE DIBSTAT TO MSGE1
          MOVE 'UNEXPECTED DLI STATUS CODE-CUST' TO MSGE2
          PERFORM PRINT-MESSAGE
          GO TO END-NOK.
          •
          •
 END-NOK.
* DISPLAY FAILURE MESSAGE
          •
          •
 GOBACK.
```

*Figure 3-4. COBOL Batch Program Using LOAD Command*

## GET Command Examples

The examples of the use of the DL/I HLPI GET UNIQUE, GET NEXT, and GET NEXT IN PARENT commands are based on the online sample application programs DLZPLI30 and DLZCBL30. These programs perform various DL/I functions on the data bases loaded by the DLZPLI10 and DLZCBL10 sample programs used in the LOAD command examples above. Figure 3-5 gives PL/I examples. Figure 3-6 on page 3-20 gives COBOL examples.

```
*PROCESS XOPTS(CICS,DLI)...;
DLZPLI30: PROCEDURE OPTIONS(MAIN);
    •
    •
DCL  OFF   BIT (1) INIT (0) STATIC;
DCL  LISCUS_CALLED BIT (1) INIT (0);
DCL 1 SAVE_AREAS,
      3 LOCSAV     CHAR (6),
      3 NUMSAV     CHAR (6),
      3 NUMBAK     CHAR (6),
      3 ITMSAV     CHAR (2);
DCL 1 FILL_ORDSAV,
      3 ......;
DCL 1 ORDSAV       CHAR (12) DEFINED FILL_ORDSAV POSITION (1);
DCL 1 STSCCST,
      3 ......;
DCL 1 STSCLOC,
      3 ......;
DCL 1 STPCORD,
      3 ......;
DCL 1 STLCITM,
     3 ......;
          •
          •
EXEC CICS                              /* HANDLE ABEND COMMAND      */
     HANDLE ABEND PROGRAM('DECODE');
          •
          •
EXEC DLI                               /* GET UNIQUE COMMAND        */
     GET UNIQUE USING PCB(1)
     SEGMENT(STSCCST) INTO(STSCCST);
IF DIBSTAT ¬= '  ' THEN                 /* WAS THERE A DL/I ERROR?   */
     CALL REALER;                      /* YES. CALL ERROR ROUTINE   */
ELSE                                   /* NO. CONTINUE              */
          •
          •
EXEC DLI                               /* GET NEXT COMMAND          */
     GET NEXT USING PCB(1)
     SEGMENT(STSCCST) INTO(STSCCST);
IF DIBSTAT ¬= '  ' &                    /* UNEXPECTED DL/I ERROR?    */
   DIBSTAT ¬= 'GB' THEN                /* YES. CALL ERROR ROUTINE   */
     CALL REALER;
ELSE                                   /* NO. CONTINUE              */
          •
          •
EXEC DLI                               /* GET UNIQUE COMMAND        */
     GET UNIQUE USING PCB(2)
     KEYFEEDBACK(NUMBAK) FEEDBACKLEN(6)/*KEY FEEDBACK NAME AND LENGTH*/

     SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
     SEGMENT(STSCLOC) INTO(STSCLOC);
IF DIBSTAT ¬= '  ' THEN                 /* UNEXPECTED DL/I ERROR?    */
   IF DIBSTAT = 'GE' &                 /* SEGMENT NOT FOUND AND NOT */
      LISCUS_CALLED = OFF THEN         /* A RECURSIVE CALL          */
          •
          •
```

*Figure 3-5 (Part 1 of 2). PL/I Online Program Using GET Commands*

```
EXEC DLI                              /* GET NEXT COMMAND          */
     GET NEXT USING PCB(2)
     SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
     SEGMENT(STSCLOC) INTO(STSCLOC);
IF DIBSTAT ¬= '  ' &                 /* UNEXPECTED DL/I ERROR?    */
   DIBSTAT ¬= 'GE' THEN               /* YES. CALL ERROR ROUTINE   */
     CALL REALER;
ELSE                                  /* NO. CONTINUE              */
          •
          •
EXEC DLI                              /* GET UNIQUE COMMAND        */
     GET UNIQUE USING PCB(2)
     SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
     SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV) INTO(STSCLOC)
     SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
     SEGMENT(STLCITM) INTO(STLCITM);
IF DIBSTAT ¬= '  ' THEN              /* WAS THERE A DL/I ERROR?   */
   CALL REALER;                       /* YES. CALL ERROR ROUTINE   */
ELSE                                  /* NO. CONTINUE              */
          •
          •
EXEC DLI                              /* GET NEXT COMMAND          */
     GET NEXT USING PCB(2)
     SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
     SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV) INTO(STSCLOC)
     SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
     SEGMENT(STLCITM) INTO(STLCITM);
IF DIBSTAT ¬= '  ' &                 /* UNEXPECTED DL/I ERROR?    */
   DIBSTAT ¬= 'GE' THEN               /* YES. CALL ERROR ROUTINE   */
     CALL REALER;
ELSE                                  /* NO. CONTINUE              */
          •
          •
EXEC DLI                              /* GET NEXT IN PARENT COMMAND */
     GET NEXT IN PARENT USING PCB(2)
     SEGMENT(STLCITM) INTO(STLCITM);
IF DIBSTAT ¬= '  ' &                 /* UNEXPECTED DL/I ERROR?    */
   DIBSTAT ¬= 'GE' THEN               /* YES. CALL ERROR ROUTINE   */
     CALL REALER;
ELSE                                  /* NO. CONTINUE              */
          •
          •
REALER: PROCEDURE;                    /* HANDLE UNEXPECTED DL/I ERRORS*/
          •
          •
END REALER;
END DLZPLI30;
```

*Figure 3-5 (Part 2 of 2). PL/I Online Program Using GET Commands*

```
 CBL XOPTS(CICS,DLI)...
 ID DIVISION.
 PROGRAM-ID. DLZCBL30.
           •
           •
01  SAVE-AREAS.
    02  LOCSAV   PIC X(6).
    02  ORDSAV.
        03 .......
    02  NUMSAV   PIC X(6).
    02  NUMBAK   PIC X(6).
    02  ITMSAV   PIC XX.
01  STSCCST.
    03 .......
01  STSCLOC.
    03 .......
01  STPCORD.
    03 .......
01  STLCITM.
    03 .......
           •
           •
* HANDLE ABEND COMMAND
*
 EXEC CICS
      HANDLE ABEND PROGRAM('DECODE')
 END=EXEC.
           •
           •
* GET UNIQUE COMMAND
*
 EXEC DLI
      GET UNIQUE USING PCB(1)
      SEGMENT(STSCCST) INTO(STSCCST) SEGLENGTH(31)
 END-EXEC.
 IF DIBSTAT NOT = '  ' GO TO REALER.
           •
           •
* GET NEXT COMMAND
*
 EXEC DLI
      GET NEXT USING PCB(1)
      SEGMENT(STSCCST) INTO(STSCCST) SEGLENGTH(31)
 END-EXEC.
 IF DIBSTAT = '  ' GO TO LSTCUS.
 IF DIBSTAT NOT = 'GB' GO TO REALER.
           •
           •
* GET UNIQUE COMMAND
*
 EXEC DLI
      GET UNIQUE USING PCB(2)
      KEYFEEDBACK(NUMBAK) FEEDBACKLEN(6)
      SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
      FIELDLENGTH(6) SEGLENGTH(106)
      SEGMENT(STSCLOC) INTO(STSCLOC) SEGLENGTH(106)
 END-EXEC.
 IF DIBSTAT = '  ' NEXT SENTENCE
    ELSE IF DIBSTAT = 'GE' GO TO LISCUS
         ELSE GO TO REALER.
           •
           •
* GET NEXT COMMAND
*
 EXEC DLI
      GET NEXT USING PCB(2)
      SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
      FIELDLENGTH(6) SEGLENGTH(106)
      SEGMENT(STSCLOC) INTO(STSCLOC) SEGLENGTH(106)
 END-EXEC.
 IF DIBSTAT = '  ' GO TO LSTLOC
    ELSE IF DIBSTAT NOT = 'GE' GO TO REALER.
           •
           •
```

*Figure 3-6 (Part 1 of 2). COBOL Online Program Using GET Commands*

```
* GET UNIQUE COMMAND
*
 EXEC DLI
      GET UNIQUE USING PCB(2)
      SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
      FIELDLENGTH(6) SEGLENGTH(106)
      SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV) INTO(STSCLOC)
      FIELDLENGTH(6) SEGLENGTH(106)
      SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
      FIELDLENGTH(12) SEGLENGTH(55)
      SEGMENT(STLCITM) INTO(STLCITM) SEGLENGTH(94)
 END-EXEC.
 IF DIBSTAT NOT = '  ' GO TO REALER.
           •
           •
* GET NEXT COMMAND
*
 EXEC DLI
      GET NEXT USING PCB(2)
      SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) INTO(STSCCST)
      FIELDLENGTH(6) SEGLENGTH(106)
      SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV) INTO(STSCLOC)
      FIELDLENGTH(6) SEGLENGTH(106)
      SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
      FIELDLENGTH(12) SEGLENGTH(55)
      SEGMENT(STLCITM) INTO(STLCITM) SEGLENGTH(94)
 END-EXEC.
 IF DIBSTAT = '  ' GO TO LSTITM.
 IF DIBSTAT NOT = 'GE' GO TO REALER.
          •
          •
* GET NEXT IN PARENT COMMAND
*
 EXEC DLI
      GET NEXT IN PARENT USING PCB(2)
      SEGMENT(STLCITM) INTO(STLCITM) SEGLENGTH(94)
 END-EXEC.
 IF DIBSTAT = '  ' GO TO MORITM.
 IF DIBSTAT NOT = 'GE' GO TO REALER.
          •
          •
 REALER.
*
* HANDLE UNEXPECTED DL/I ERRORS
*
         •
         •
```

*Figure 3-6 (Part 2 of 2). COBOL Online Program Using GET Commands*

## INSERT, REPLACE, and DELETE Command Examples

The examples of the use of the DL/I HLPI INSERT, REPLACE, and DELETE commands are taken from, or based on, the online sample application programs DLZPLI30 and DLZCBL30. These programs perform various DL/I functions on the data bases loaded by the DLZPLI10 and DLZCBL10 sample programs used in the LOAD command examples above. Figure 3-7 gives PL/I examples. Figure 3-8 on page 3-24 gives COBOL examples.

```
*PROCESS XOPTS(CICS,DLI)...;
DLZPLI30: PROCEDURE OPTIONS(MAIN);
          •
          •
DCL 1 SAVE_AREAS,
      3 LOCSAV     CHAR (6),
      3 NUMSAV     CHAR (6),
      3 ITMSAV     CHAR (2);
DCL 1 FILL_ORDSAV,
      3 ......;
DCL 1 ORDSAV       CHAR (12) DEFINED FILL_ORDSAV POSITION (1);
DCL 1 STSCCST,
      3 ......;
DCL 1 STSCLOC,
      3 ......;
DCL 1 STPCORD,
      3 ......;
DCL 1 STLCITM,
      3 ......;
          •
          •
EXEC CICS
     HANDLE ABEND PROGRAM('DECODE');   /* HANDLE ABEND COMMAND      */
          •
          •
EXEC DLI                               /* INSERT COMMAND            */
     INSERT USING PCB(1)
     SEGMENT(STLCITM) FROM(STLCITM);
IF DIBSTAT ¬= '  ' THEN                 /* WAS THERE A DL/I ERROR?   */
   CALL REALER;                        /* YES. CALL ERROR ROUTINE   */
ELSE                                   /* NO. CONTINUE              */
          •
          •
EXEC DLI                               /* INSERT COMMAND            */
     INSERT USING PCB(1)
     SEGMENT(STSCCST) FROM(STSCCST)
     SEGMENT(STSCLOC) FROM(STSCLOC)
     SEGMENT(STPCORD) FROM(STPCORD);
IF DIBSTAT ¬= '  ' THEN                 /* WAS THERE A DL/I ERROR?   */
   CALL REALER;                        /* YES. CALL ERROR ROUTINE   */
ELSE                                   /* NO. CONTINUE              */
          •
          •
EXEC DLI                               /* INSERT COMMAND            */
     INSERT USING PCB(1)
     SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV)
     SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV)
     SEGMENT(STPCORD) FROM(STPCORD);
IF DIBSTAT ¬= '  ' THEN                 /* WAS THERE A DL/I ERROR?   */
   CALL REALER;                        /* YES. CALL ERROR ROUTINE   */
ELSE                                   /* NO. CONTINUE              */
          •
          •
```

*Figure 3-7 (Part 1 of 2). PL/I Online Program Using INSERT, REPLACE, and DELETE Commands*

```
EXEC DLI                              /* GET UNIQUE COMMAND FOR    */
     GET UNIQUE USING PCB(1)          /* FOLLOWING REPLACE COMMAND */
     SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV)
     SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV)
     SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
     SEGMENT(STLCITM) WHERE(STQCILI=ITMSAV) INTO(STLCITM);
          •
          •
EXEC DLI                              /* REPLACE COMMAND           */
     REPLACE USING PCB(1)
     SEGMENT(STPCORD) FROM(STPCORD)
     SEGMENT(STLCITM) FROM(STLCITM);
IF DIBSTAT ¬= '  ' THEN               /* WAS THERE A DL/I ERROR?   */
   CALL REALER;                       /* YES. CALL ERROR ROUTINE   */
ELSE                                  /* NO. CONTINUE              */
          •
          •
EXEC DLI                              /* GET UNIQUE COMMAND FOR    */
     GET UNIQUE USING PCB(1)          /* FOLLOWING DELETE COMMAND  */
     SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV)
     SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV)
     SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
     SEGMENT(STLCITM) WHERE(STQCILI=ITMSAV) INTO(STLCITM);
          •
          •
EXEC DLI                              /* DELETE COMMAND. ALSO      */
     DELETE USING PCB(1)              /* DELETES STLCITM SEGMENT   */
     SEGMENT(STPCORD) FROM(STPCORD);
IF DIBSTAT ¬= '  ' THEN               /* WAS THERE A DL/I ERROR?   */
   CALL REALER;                       /* YES. CALL ERROR ROUTINE   */
ELSE                                  /* NO. CONTINUE              */
          •
          •
REALER: PROCEDURE;                    /* HANDLE UNEXPECTED DL/I ERRORS*/
          •
          •
END REALER;
END DLZPLI30;
```

*Figure 3-7 (Part 2 of 2). PL/I Online Program Using INSERT, REPLACE, and DELETE Commands*

```
 CBL XOPTS(CICS,DLI)...
 ID DIVISION.
 PROGRAM-ID. DLZCBL30.
           •
           •
01  SAVE-AREAS.
    02  LOCSAV   PIC X(6).
    02  ORDSAV.
        03 .......
    02  NUMSAV   PIC X(6).
    02  ITMSAV   PIC XX.
01  STSCCST.
    03 .......
01  STSCLOC.
    03 .......
01  STPCORD.
    03 .......
01  STLCITM.
    03 .......
           •
           •
* HANDLE ABEND COMMAND
*
 EXEC CICS
      HANDLE ABEND PROGRAM('DECODE')
 END-EXEC.
           •
           •
* INSERT COMMAND
*
 EXEC DLI
      INSERT USING PCB(1)
      SEGMENT(STLCITM) FROM(STLCITM) SEGLENGTH(94)
 END-EXEC.
 IF DIBSTAT NOT = '  ' GO TO REALER.
           •
           •
* INSERT COMMAND
*
 EXEC DLI
      INSERT USING PCB(1)
      SEGMENT(STSCCST) FROM(STSCCST) SEGLENGTH(106)
      SEGMENT(STSCLOC) FROM(STSCLOC) SEGLENGTH(106)
      SEGMENT(STPCORD) FROM(STPCORD) SEGLENGTH(55)
 END-EXEC.
 IF DIBSTAT NOT = '  ' GO TO REALER.
           •
           •
* INSERT COMMAND
*
 EXEC DLI
      INSERT USING PCB(1)
      SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) FIELDLENGTH(6)
      SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV) FIELDLENGTH(6)
      SEGMENT(STPCORD) FROM(STPCORD) SEGLENGTH(55)
 END-EXEC.
 IF DIBSTAT NOT = '  ' GO TO REALER.
           •
           •
* GET UNIQUE COMMAND FOR FOLLOWING REPLACE COMMAND
*
 EXEC DLI
      GET UNIQUE USING PCB(1)
      SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) FIELDLENGTH(6)
      SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV) FIELDLENGTH(6)
      SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
      FIELDLENGTH(12) SEGLENGTH(55)
      SEGMENT(STLCITM) WHERE(STQCILI=ITMSAV) INTO(STLCITM)
      FIELDLENGTH(2) SEGLENGTH(94)
 END-EXEC.
           •
           •
```

*Figure 3-8 (Part 1 of 2). COBOL Online Program Using INSERT, REPLACE, and DELETE Commands*

```
* REPLACE COMMAND
*
 EXEC DLI
      REPLACE USING PCB(1)
      SEGMENT(STPCORD) FROM(STPCORD) SEGLENGTH(55)
      SEGMENT(STLCITM) FROM(STLCITM) SEGLENGTH(94)
 END-EXEC.
 IF DIBSTAT NOT = '  ' GO TO REALER.
           •
           •
* GET UNIQUE COMMAND FOR FOLLOWING DELETE COMMAND
*
 EXEC DLI
      GET UNIQUE USING PCB(1)
      SEGMENT(STSCCST) WHERE(STQCCNO=NUMSAV) FIELDLENGTH(6)
      SEGMENT(STSCLOC) WHERE(STQCLNO=LOCSAV) FIELDLENGTH(6)
      SEGMENT(STPCORD) WHERE(STQCODN=ORDSAV) INTO(STPCORD)
      FIELDLENGTH(12) SEGLENGTH(55)
      SEGMENT(STLCITM) WHERE(STQCILI=ITMSAV) INTO(STLCITM)
      FIELDLENGTH(2) SEGLENGTH(94)
 END-EXEC.
           •
           •
* DELETE COMMAND
*
 EXEC DLI
      DELETE USING PCB(1)
      SEGMENT(STPCORD) FROM(STPCORD) SEGLENGTH(55)
 END-EXEC.
 IF DIBSTAT NOT = '  ' GO TO REALER.
           •
           •
 REALER.
*
* HANDLE UNEXPECTED DL/I ERRORS
*
           •
           •
```

*Figure 3-8 (Part 2 of 2). COBOL Online Program Using INSERT, REPLACE, and DELETE Commands*

## SCHEDULE, TERMINATE, and CHECKPOINT Command Examples

The examples of the use of the DL/I HLPI SCHEDULE, TERMINATE, and CHECKPOINT commands are taken from, or based on, the online sample application programs DLZPLI30 and DLZCBL30. These programs perform various DL/I functions on the data bases loaded by the DLZPLI10 and DLZCBL10 sample programs used in the LOAD command examples above. Figure 3-9 gives PL/I examples. Figure 3-10 on page 3-27 gives COBOL examples.

```
*PROCESS XOPTS(CICS,DLI)...;
DLZPLI30: PROCEDURE OPTIONS(MAIN);
          •
          •
EXEC CICS
     HANDLE ABEND PROGRAM('DECODE');    /* HANDLE ABEND COMMAND       */
          •
          •
EXEC DLI                                /* SCHEDULE COMMAND           */
     SCHEDULE PSB(STBCUSR);
          •
          •
EXEC DLI                                /* CHECKPOINT COMMAND         */
     CHECKPOINT ID('CHCKPT01');
          •
          •
EXEC DLI                                /* TERMINATE COMMAND          */
     TERMINATE;
          •
          •
EXEC DLI                                /* SCHEDULE COMMAND           */
     SCHEDULE PSB(STBCUSU);
          •
          •
EXEC DLI                                /* TERMINATE COMMAND          */
     TERMINATE;
          •
          •
END DLZPLI30;
```

*Figure 3-9. PL/I Online Program Using SCHEDULE, TERMINATE, and CHECKPOINT Commands*

```
CBL XOPTS(CICS,DLI)...
ID DIVISION.
PROGRAM-ID. DLZCBL30.
          •
          •
EXEC CICS
     HANDLE ABEND PROGRAM('DECODE')
END-EXEC.
          •
          •
EXEC DLI
     SCHEDULE PSB(CBBCUSR)
END-EXEC.
          •
          •
EXEC DLI
     CHECKPOINT ID('CHCKPT01')
END-EXEC.
          •
          •
EXEC DLI
     TERMINATE
END-EXEC.
          •
          •
EXEC DLI
     SCHEDULE PSB(CBBCUSU)
END-EXEC.
          •
          •
EXEC DLI
     TERMINATE
END-EXEC.
          •
          •
```

*Figure 3-10. COBOL Online Program Using SCHEDULE, TERMINATE, and CHECKPOINT Commands*

## HANDLE ABEND Command Examples

In the CICS/VS online environment, you can intercept DL/I abends by coding an EXEC CICS HANDLE ABEND command at the beginning of your program (see Figure 3-5 on page 3-18 through Figure 3-10). This command causes all task abends to be intercepted and control passed to the specified abend processing program on task abend.

Figure 3-11 on page 3-28 gives an example of an abend processing program using COBOL and Figure 3-12 on page 3-29 gives an example using PL/I.

These examples of an abend processing program select DL/I HLPI abend codes, which begin with 'DHxx', and use the DL/I status code (the 'xx' in the abend code) to select a message that describes the cause of the abend.

```
CBL APOST,LIB,SUPMAP,CLIST,SXREF,XOPTS(CICS,DLI)
      ID DIVISION.
      PROGRAM-ID. DECODE.
      REMARKS. INTERCEPTS DL/I ABENDS AND ISSUES MESSAGE.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      INPUT-OUTPUT SECTION.
      DATA DIVISION.
      FILE SECTION.
      WORKING-STORAGE SECTION.
      01  MSGAB         PIC X(72) VALUE ' NO SEGMENT I/O AREA       '.
      01  MSGAC         PIC X(72) VALUE ' HIERARCHICAL ERROR        '.
      01  MSGAD         PIC X(72) VALUE ' INVALID FUNCTION SPECIFIED'.
      01  MSGAH         PIC X(72) VALUE ' NO SEGMENT SPECIFIED      '.
            •
            •
            •
      01  MSGXH         PIC X(72) VALUE ' LOGGING NOT ACTIVE        '.
      01  MSGXX         PIC X(72) VALUE ' NO TEXT FOR THIS ABCODE   '.
      01  MSG           PIC X(72) VALUE '                           '.
      01  MSGABEND.
          02  FILLER    PIC X(19) VALUE ' DL/I ABEND CODE = '.
          02  STATCD    PIC X(2) VALUE SPACES.
          02  FILLER    PIC X(51) VALUE '                   '.
      01  ABCODE.
          02  DLICODE   PIC X(2) VALUE SPACES.
          02  DLISTAT   PIC X(2) VALUE SPACES.
      PROCEDURE DIVISION.
          EXEC CICS ASSIGN ABCODE (ABCODE) END-EXEC.
          IF DLICODE = 'DH'
              MOVE DLISTAT TO STATCD
              EXEC CICS SEND TEXT FROM(MSGABEND) LENGTH(72)
                    ACCUM END-EXEC
              GO TO DHAB
          ELSE GO TO END-RUT.
      DHAB.
          IF DLISTAT = 'AB'
              MOVE MSGAB TO MSG
              GO TO END-RUT.
      DHAC.
          IF DLISTAT = 'AC'
              MOVE MSGAC TO MSG
              GO TO END-RUT.
      DHAD.
          IF DLISTAT = 'AD'
              MOVE MSGAD TO MSG
              GO TO END-RUT.
      DHAH.
          IF DLISTAT = 'AH'
              MOVE MSGAH TO MSG
              GO TO END-RUT.
            •
            •
            •
      DHXH.
          IF DLISTAT = 'XH'
              MOVE MSGXH TO MSG
              GO TO END-RUT.
      DHXX.
          MOVE MSGXX TO MSG.
      END-RUT.
          EXEC CICS SEND TEXT FROM(MSG) LENGTH(72)
                    ACCUM END-EXEC.
          EXEC CICS SEND PAGE END-EXEC.
          EXEC CICS RETURN END-EXEC.
          STOP RUN.
```

*Figure 3-11. COBOL Online HANDLE ABEND Program*

```
* PROCESS MACRO,WORKFILE(3330),AG,ESD,MAP,STG,SIZE(MAX),NIS,
     OFFSET,LC(60),XOPTS(CICS,DLI,LC(70));
/*********************************************************************/
/*   NAME:      DECODE                                               */
/*                                                                   */
/*   FUNCTION:  INTERCEPT HLPI ABENDS AND ISSUE A DESCRIPTIVE        */
/*              MESSAGE                                              */
/*                                                                   */
/*********************************************************************/
%SKIP(2);
DECODE: PROCEDURE OPTIONS(MAIN);
%SKIP(2);
/*********************************************************************/
/*          D E S C R I P T I V E   S T A T E M E N T S              */
/*********************************************************************/
%SKIP(2);
DCL 1 MSGAB CHAR(42) INIT(' NO SEGMENT I/O AREA IN COMMAND');
DCL 1 MSGAC CHAR(42) INIT(' HIERARCHICAL ERROR IN COMMAND');
DCL 1 MSGAD CHAR(42) INIT(' INVALID FUNCTION SPECIFIED');
DCL 1 MSGAH CHAR(42) INIT(' COMMAND DOES NOT HAVE SEGMENT KEYWORD');
       •
       •
       •
DCL 1 MSGXH CHAR(42) INIT(' LOGGING NOT ACTIVE DURING CKPT');
DCL 1 MSG__ CHAR(42) INIT(' NO TEXT FOR THIS STATUS CODE');
DCL 1 MSG   CHAR(72) INIT(' ');
DCL 1 MSGABEND,
      3 PT1   CHAR(33) INIT(' DL/I ABEND INTERCEPTED - CODE = ')
      3 MSTAT CHAR(2),
      3 PT3   CHAR(37) INIT(' ');
DCL 1 ABCODE,
      3 DLICODE CHAR(2),
      3 DLISTAT CHAR(2);
%SKIP(2);
DCL STG BUILTIN;
DCL ADDR BUILTIN;
DCL LOW BUILTIN;
%PAGE;
/*********************************************************************/
/*              P R O C E S S   S T A T E M E N T S                  */
/*********************************************************************/
%SKIP(2);
EXEC CICS ASSIGN ABCODE (ABCODE);
IF DLICODE = 'DH' THEN
DO;
  MSTAT = DLISTAT;
  EXEC CICS SEND TEXT FROM(MSGABEND) LENGTH(72) ACCUM;
  SELECT (DLISTAT);
    WHEN ('AB') MSG = MSGAB;
    WHEN ('AC') MSG = MSGAC;
    WHEN ('AD') MSG = MSGAD;
    WHEN ('AH') MSG = MSGAH;
        •
        •
        •
    WHEN ('XH') MSG = MSGXH;
    OTHERWISE   MSG = MSG__;
  END;
    EXEC CICS SEND TEXT FROM(MSG) LENGTH(72) ACCUM;
END;
EXEC CICS SEND PAGE;
EXEC CICS RETURN;
END DECODE;
```

*Figure 3-12. PL/I Online HANDLE ABEND Program*

## CHECKPOINT Command Examples Using MPS Restart

Figure 3-13 on page 3-31 gives COBOL examples of checkpoints in an MPS batch program running with MPS Restart active. PL/I examples are given in Figure 3-14 on page 3-32. When using MPS Restart, a VSE checkpoint must be coded before each DL/I checkpoint in the application program. The two checkpoints together are referred to as a “combined checkpoint.” No other DL/I commands may be issued between the VSE checkpoint and the DL/I CHKP command.

```
 CBL XOPTS(CICS,DLI)...
 ID DIVISION.
 PROGRAM-ID.DLZCBLMP.
           •
           •
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CHKPT-MSGS ASSIGN TO SYS101-UR-3203-S.
 I-O-CONTROL.
* TAKE A VSE CHKPT ON EVERY WRITE TO CHKPT-MSGS
*
     RERUN ON SYS100-UT-2400-S
         EVERY 1 RECORDS OF CHKPT-MSGS.
 DATA DIVISION.
 FILE SECTION.
 FD CHKPT-MSGS
    LABEL RECORD OMITTED
    RECORDING IS F
    RECORD CONTINAS 72 CHARACTERS
    DATA RECORD IS MSG-LINE.
 01  MSG-LINE.
     02  FILLER   PIC X.
     02  ALL-71   PIC X(71).
           •
           •
 WORKING-STORAGE SECTION.
 01  INITMSG       PIC X(72) VALUE ' READY FOR VSE CHECKPOINTS '.
 01  CHKPTMSG      PIC X(72) VALUE ' TAKING A VSE CHECKPOINT'.
 01  CHKPTIO       PIC X(8)  VALUE ' DL/I CHKP'.
           •
           •
 PROCEDURE DIVISION.
      OPEN OUTPUT CHKPT-MSGS.
* INITIALIZE CHECKPOINT MESSAGE FILE
*
      WRITE MSG-LINE FROM INITMSG AFTER POSITIONING 2.
           •
           •
* TAKE AN IMPLICIT VSE CHECKPOINT
*
      WRITE MSG-LINE FROM CHKPTMSG AFTER POSITIONING 2.
* TAKE A DL/I CHECKPOINT
*
      EXEC DLI
           CHECKPOINT ID(CHKPTID)
      END-EXEC.
           •
           •
```

**Note:** In this example, a print file is defined so that a VSE checkpoint is issued each time a message is written to it. The initial message is necessary because COBOL does not start issuing VSE checkpoints until the write statement after the first write occurs. COBOL will also issue a VSE checkpoint when the print file is closed a the end of the program, but this additional checkpoint will not affect the function of MPS Restart. Since the COBOL-VSE checkpoint interface is implicit and the VSE checkpoint ID is not available to the application program, it cannot be used as the checkpoint ID on the DL/I CHKP command.

*Figure 3-13. COBOL Example of a Combined Checkpoint in an MPS Batch Program Using MPS Restart*

```
*PROCESS XOPTS(CICS,DLI)...;
DLZPLIMP: PROCEDURE OPTIONS(MAIN);
          .
          .
DECLARE CHKPTID CHAR(8);
DECLARE RETCODE FIXED BIN(31);
          .
          .
CALL PLICKPT('',CHKPTID,'SYS100,2400',RETCODE);  /* ISSUE A VSE CHKPT          */
IF RETCODE > 4 THEN                              /* VSE CHKPT ERROR            */
DO;
    PUT EDIT ('ERROR DURING CHECKPOINT. RETCODE=',RETCODE) (A,F(2));
    STOP;                                        /* ABNORMAL END               */
END;
IF RETCODE = 4 THEN                              /* VSE RESTART OCCURRED       */
    PUT EDIT('RESTARTED AT CHECKPOINT #',CHKPTID)(A);
EXEC DLI
     CHECKPOINT ID(CHKPTID);            /* ISSUE A DL/I CHKP WITH VSE CHKPT ID */
          .
          .
END DLZPLIMP;
```

**Note:** In the above program example, the VSE checkpoint ID (CHKPTID) is used as the checkpoint ID for the DL/I CHKP command. This provides a cross reference between the normal checkpoint messages issued to SYSLOG as the result of taking VSE and DL/I checkpoints. While this is the recommended procedure for PL/I programs, it is not mandatory when using the MPS Restart facility.

*Figure 3-14. PL/I Example of Combined Checkpoint in an MPS Batch Program Using MPS Restart*

## Executing Your Program

This section will present information that will help you in preparing your DL/I application program for execution. Examples of the necessary steps and the job control statements needed to perform them are included. Where particular information does not apply to one or more of the possible operating environments, the exceptions will be noted.

## Translation

### Translator

The DL/I HLPI commands that you coded in your application program must be translated into calls to DL/I in the host language of your application program.

The translation is done by the CICS/VS EXEC translator. The translator is executed as a separate job step before compilation of your program. CICS/VS supplies cataloged procedures you can use to set up the translation step. They are described in the *CICS/DOS/VS Installation and Operations Guide*.

The translator generates an initialization call at the beginning of each external procedure. It also generates, for each external procedure, the labels for the DIB fields and other fields that may be needed by statements subsequently generated.

The translator searches your code for triggers corresponding to the XOPTS options specified on the PROCESS (PL/I) or CBL (COBOL) statements at the beginning of your program. (See *CICS/VS Application Programmer's Reference Manual (Command Level)* for details on translator options.)

Possible triggers are:

```
{EXECUTE CICS}     {EXECUTE DLI}
{EXEC    CICS}     {EXEC    DLI}
```

Either type of command, or both, may be present in an online program.

When the translator recognizes a DL/I HLPI command, it scans the command looking for proper syntax. If an error is found, a message is printed. The translator proceeds to generate one or more CALL statements to DL/I. The translator replaces the command you coded in your program with the calls and parameter lists.

## Compilation and Link-editing

The output of the translator must be compiled by the appropriate host language compiler and link-edited into a VSE core image library for execution as a separate job. DL/I application programs cannot be executed in a compile-link-go environment, since they run as a subprogram of the DL/I initialization program.

The appropriate DL/I or CICS/VS interface module must be link-edited with your program. The module to be included is determined by the host language and the operating environment.

Batch and MPS batch programs must also have an entry statement included in the input to the linkage editor.

The following examples illustrate job control statements needed for various combinations of environment and host language. Further examples and use of translator cataloged procedures appear in the *CICS/DOS/VS Installation and Operations Guide*.

*For COBOL:*
*Online*

```
// JOB COBSAMPL
// DLBL IJSYSPH,'COBOL TRANSLATION',yy/ddd
// EXTENT SYSPCH,balance of extent information
ASSGN SYSPCH,DISK,VOL=volid,SHR
// EXEC DFHECP1$,SIZE=...
 CBL LIB,XOPTS(CICS,DLI)
       •
       •
SOURCE DECK
       •
       •
/*
CLOSE SYSPCH,PUNCH
// DLBL IJSYSIN,'COBOL TRANSLATION',yy/ddd
// EXTENT SYSIPT
ASSGN SYSIPT,DISK,VOL=volid,SHR
// OPTION SYM,ERRS,NODECK,CATAL
   PHASE COBSAMPL,*
   INCLUDE DFHECI
// EXEC FCOBOL,SIZE=...
// EXEC LNKEDT
/&
// JOB RESET
CLOSE SYSIPT,00C
/&
```

*Batch and MPS batch*

```
// JOB COBSAMPL
// DLBL IJSYSPH,'COBOL TRANSLATION',yy/ddd
// EXTENT SYSPCH,balance of extent information
ASSGN SYSPCH,DISK,VOL=volid,SHR
// EXEC DFHECP1$,SIZE=...
 CBL LIB,XOPTS(DLI)
       •
       •
SOURCE DECK
       •
       •
/*
CLOSE SYSPCH,PUNCH
// DLBL IJSYSIN,'COBOL TRANSLATION',yy/ddd
// EXTENT SYSIPT
ASSGN SYSIPT,DISK,VOL=volid,SHR
// OPTION SYM,ERRS,NODECK,CATAL
   PHASE COBSAMPL,*
   INCLUDE DLZLICBL
   INCLUDE DLZBPJRA
// EXEC FCOBOL,SIZE=...
   ENTRY CBLCALLA
// EXEC LNKEDT
/&
// JOB RESET
CLOSE SYSIPT,00C
/&
```

*For PL/I:*
*Online*

```
// JOB PLISAMPL
// DLBL IJSYSPH,'PL/I TRANSLATION',yy/ddd
// EXTENT SYSPCH,balance of extent information
ASSGN SYSPCH,DISK,VOL=volid,SHR
// EXEC DFHEPP1$,SIZE=...
*PROCESS INCLUDE,XOPTS(CICS,DLI);
       •
       •
SOURCE DECK
       •
       •
/*
CLOSE SYSPCH,PUNCH
// DLBL IJSYSIN,'PL/I TRANSLATION',yy/ddd
// EXTENT SYSIPT
ASSGN SYSIPT,DISK,VOL=volid,SHR
// OPTION CATAL
   PHASE PLISAMPL,*
   INCLUDE DFHPL1I
// EXEC PLIOPT,SIZE=...
// EXEC LNKEDT
/&
// JOB RESET
CLOSE SYSIPT,00C
/&
```

### *Batch and MPS batch*

```
// JOB PLISAMPL
// DLBL IJSYSPH,'PL/I TRANSLATION',yy/ddd
// EXTENT SYSPCH,balance of extent information
ASSGN SYSPCH,DISK,VOL=volid,SHR
// EXEC DFHEPP1$,SIZE=...
*PROCESS INCLUDE,XOPTS(DLI);
       •
       •
SOURCE DECK
       •
       •
/*
CLOSE SYSPCH,PUNCH
// DLBL IJSYSIN,'PL/I TRANSLATION',yy/ddd
// EXTENT SYSIPT
ASSGN SYSIPT,DISK,VOL=volid,SHR
// OPTION CATAL
   PHASE PLISAMPL,*
// EXEC PLIOPT,SIZE=...
   INCLUDE DLZLIPLI
   INCLUDE IBMBPJRA
   ENTRY DLZLIPLI
// EXEC LNKEDT
/&
// JOB RESET
CLOSE SYSIPT,00C
/&
```

# Execution

## Online

An online DL/I application is executed by entering its assigned transaction ID through a terminal, as with any other online task.

Assignment of the transaction ID and other CICS/VS table requirements are described in *DL/I DOS/VS Data Base Administration.*

## Batch and MPS Batch

Once your compiled application program is link-edited into a VSE core image library with the appropriate modules, as shown in the job control statements above, it is ready for execution as a subprogram under the DL/I initialization program. The EXEC statement in your job stream names the DL/I initialization module rather than your application program.

## Parameter Statement

The name of your application program to be executed and the PSB it uses are identified in a parameter statement that follows the EXEC statement in the job control statement stream.

The format of the DL/I parameter statement, beginning in column one, looks like this:

```
{DLI},progname,psbname[,{buff}]
{DLR}                    {1   }
               [,HDBFR=({bufno}[,dbdname1,dbdname2,...])][,...]
                        {32  }
               [,HSBFR=({indno},{ksdsbuf},[{esdsbuf}],dbdname3)][,...]
                        {3   } {2   }    {2      }
                [,TRACE=modname][,ASLOG=YES]
                [,LOG=({TAPE }),{PAUSE  })]
                       {DISK1} {NOPAUSE}
                       {DISK2}
```

Parameters can be entered from SYSIPT or SYSLOG. However, continuation statements, if required, can be entered only from SYSIPT. Continuation statements are not permitted from SYSLOG.

Continuation is indicated by a nonblank character in column 72 of the statement being continued. The parameter statement can be stopped in or before column 71, and continued in a continuation statement.

**DLI**

The DLI function code is required for batch DL/I programs or MPS batch DL/I programs that do not use the MPS Restart facility.

**DLR**

The DLR function code is required for MPS batch programs using the MPS Restart facility. If DLR is specified as the function code for a non-MPS batch program, it will be treated exactly as if DLI had been specified.

**Note:** The only valid parameters for the DLR funciton code are PROGNAME and PSBNAME.

**progname**
specifies a one to eight alphameric character name of the application program or utility to be executed.

**psbname**
specifies a one to seven alphameric character name of the PSB as named in the PSB generation.

**buff**
specifies the number (1 to 255) of data base subpools required for this execution; if omitted, 1 is assumed. If no buffer pool control options are specified, a subpool consists of 32 fixed-length buffers. The buffer size is generally consistent with the VSE/VSAM data base control interval size and may be 512 or any multiple of 512 bytes. The buffer size value is determined at DL/I system initialization and is based on the value specified in BFRPOOL, the number of data bases, and size of the VSE/VSAM control intervals. A data base is assigned a subpool containing buffers that are equal to or greater in size than the size of the data base control interval. See *DL/I DOS/VS Data Base Administration* for buffer pool information.

**HDBFR**
describes one DL/I subpool. See *DL/I DOS/VS Data Base Administration* for buffer pool information.

- bufno specifies the number (2 to 32) of buffers to be allocated for this subpool. If omitted for a specific subpool, 32 is assumed. A specification exceeding 2 digits will cause an abnormal termination.
- dbdname1, dbdname2,...specify the names of DBDs that are to be allocated to this subpool. If no dbdnames are specified, this subpool is used for DMBs not explicitly assigned; the parentheses around the number of buffers are still required. The DBD name used should be the physical DBD even though a logical DBD is being used.

**HSBFR**
defines VSE/VSAM buffer allocation for HISAM, SHISAM, and INDEX data bases. See *DL/I DOS/VS Data Base Administration* for buffer pool information.

- indno specifies the number of index buffers for a KSDS; if omitted, 3 is assumed. A specification of 1 or 2 digits is permitted. A specification exceeding 2 digits will cause an abnormal termination.
- ksdsbuf specifies the number of data buffers for a KSDS; if omitted, 2 is assumed. A specification of 1 or 2 digits is permitted. A specification exceeding 2 digits will cause an abnormal termination.
- esdsbuf specifies the number of data buffers for the ESDS (applies to HISAM only); if omitted, 2 is assumed. A specification of 1 or 2 digits is permitted. A specification exceeding 2 digits will cause an abnormal termination.
- dbdname3 is the name of the HISAM, SHISAM, or INDEX DBD referenced by the application program.

**TRACE**
indicates that tracing is to be active during this execution. See the *DL/I DOS/VS Diagnostic Guide* for details on tracing.

**ASLOG=YES**
specifies that asynchronous logging is to be used. See *DL/I DOS/VS Data Base Administration* for asynchronous logging information.

**LOG**
specifies the type of logging to be used.

**TAPE**
indicates the log records are to be written to a tape device. It is the default if the LOG parameter is omitted.

**DISK1**
indicates the log records are to be written on one disk extent with the filename DSKLOG1.

**DISK2**
indicates that the log records are to be written on two disk extents. If one disk extent becomes full, the extent is closed and the other extent is used. DSKLOG1 is used first; then DSKLOG2. If DSKLOG2 becomes full, logging will switch back to DSKLOG1 and continue to repeat the sequence.

**PAUSE**
indicates that before reusing the only disk extent (DISK1) or before switching to the next extent (DISK2), the operator is notified and the partition waits for the operator's reply. PAUSE is the default if the second option in the LOG parameter is omitted.

**NOPAUSE**
indicates that reusing a log extent or switching log extents is done without notifying the operator.

**Note:** The UPSI byte (bit 6=0) must be set to indicate DL/I logging is required. If anything other than the above parameters are specified, an error message is issued and the job is canceled.

## UPSI

You can control certain execution-time functions through use of the UPSI byte settings that you can set by including this statement in the job control stream:

```
// UPSI xxxxxxxx
```

where the meanings of the settings of these bits are as described below for each operating environment.

**Batch**

```
Bit 0  =   0  Read parameter information via SYSIPT.
       =   1  Read parameter information via SYSLOG.

Bits 1-4      Available for use by the application program.

Bit 5  =   0  Storage dump on set exit (STXIT) abnormal
              termination if STXIT active (that is, bit 7 = 0).
       =   1  No storage dump on set exit (STXIT) abnormal
              termination.

Bit 6  =   0  All data base modifications written to the DL/I
              system log.
       =   1  DL/I system log function inactive.

Bit 7  =   0  Set exit (STXIT) linkage to DL/I for abnormal
              task termination.
       =   1  STXIT inactive.
```

MPS batch

```
Bit 0  =   0  Read parameter information via SYSIPT.
       =   1  Read parameter information via SYSLOG.

Bits 1-4      Available for use by the application program.

Bit 5  =   0  Storage dump on set exit (STXIT) abnormal
              task termination.
       =   1  No storage dump on set exit (STXIT) abnormal task
              termination.

Bits 6-7      Not used for MPS.  Data base logging, normally
              controlled by UPSI bit 6, is controlled in the
              CICS/VS partition under MPS operation.  STXIT
              linkage to DL/I for abnormal task termination,
              normally controlled by UPSI bit 7, is always
              active under MPS operation.
```

*Online* The UPSI byte is set at system initialization as a system programming function.

If you are unsure of the significance of these functions, the system programming or data base administration personnel in your installation can provide more information.

## Job Control Statements

The following information tells you how to set up the job control statements for the execution of your program.

***Batch:*** If data base changes are to be logged, either disk (batch environment only) or tape logging must be specified on the DL/I parameter statement with the LOG parameter. If the LOG parameter is omitted and UPSI byte bit 6=0, the default is tape logging.

If tape logging is used, ASSGN and TLBL statements as shown below are required. The log tape must have a standard label.

```
//  ASSGN  SYS011,cuu
//  TLBL   LOGOUT
```

If disk logging is used, the DLBL statement as shown below is required. The log file must have been previously defined with a DEFINE command because this is a VSAM file.

```
//  DLBL     {DSKLOG1},'cluster-name',,VSAM
             {DSKLOG2}
```

The execution job stream must contain DLBL or TLBL statements that define the data base(s) to be processed. ASSGN and EXTENT statements are also required for SHSAM and HSAM data bases. When initially loading a data base, additional DLBL and EXTENT statements may also be required for system work files. Consult data base administration for the details.

The EXEC statement specifies the DL/I initialization program and the SIZE parameter. Typically you will require a 512K virtual partition for execution with a SIZE parameter of 256K. See *VSE/Advanced Functions System Control Statements*, for details.

```
//  EXEC     DLZRRC00,SIZE=xxxK
```

Shown below are the execution job control statements for a program INVUPDT with a PSB of INVMSTR. It is assumed that the updates to the data base will be logged and that HISAM is the access method for the data base.

The number of DLBL statements varies depending on the number of data bases accessed and the DL/I access method used. See *DL/I DOS/VS Resource Definition and Utilities* for more details. The information in the DLBL statements defining the data bases must be the same as assigned in the DBDs for those data bases by DBA.

The UPSI statement is optional and when set to all zeros, as shown, can be omitted.

If the application program does retrievals only, or if the UPSI byte is used to turn off data base logging, no log tape or disk is required.

```
//  JOB     UPDATE
//  UPSI    00000000
//  ASSGN   SYS011,182
//  TLBL    LOGOUT
//  DLBL    INVPRT1,'INVENTORY',99/365,VSAM
//  DLBL    INVPRTZ,'INVENTORY-OFLOW',99/365,VSAM
//  EXEC    DLZRRC00,SIZE=...
DLI,INVUPDT,INVMSTR
            •
            •
            DATA CARDS IF REQUIRED
            •
            •
/*
/&
```

***MPS Batch:*** The EXEC statement specifies the DL/I initialization program and the SIZE parameter. Typically you will require a 512K virtual partition for execution with a SIZE parameter of 256K. See *VSE/Advanced Functions System Control Statements*, for details.

```
//  EXEC     DLZMPI00,SIZE=xxxK
```

Shown below are the execution job control statements for a program INVUPDT with a PSB of INVMSTR. For the MPS environment, data base logging is controlled in the CICS/VS partition.

The UPSI statement is optional and when set to all zeros, as shown, can be omitted.

```
//  JOB     UPDATE
//  UPSI    00000000
//  EXEC    DLZMPI00,SIZE=...
DLI,INVUPDT,INVMSTR
            •
            •
            DATA CARDS IF REQUIRED
            •
            •
/*
/&
```

***MPS Batch Using MPS Restart:*** Shown below are the execution job control statements for the same program using MPS Restart. Included in this example are statements which assign a tape to contain certain checkpoint records written by VSE checkpoints.

```
//  JOB     UPDATE
//  MTC     REW,280
//  ASSGN   SYS100,280
//  EXEC    DLZMPI00,SIZE=...
DLR,INVUPDT,INVMSTR
            •
            •
            DATA CARDS IF REQUIRED
            •
            •
/*
/&
```

The MPS Restart facility is invoked for an MPS batch job by using the DLR function code in the parameter input to DL/I. This function code (see above example) replaces the DLI function code used for normal batch and MPS batch jobs. The DLR function code must be used when the job is first started and not just when it is restricted.

***Restarting an MPS Batch Program Using MPS Restart:*** The following steps are required to restart an MPS batch program after a failure:

1. Get the VSE checkpoint ID from the SYSLOG message.
   a. If the individual MPS batch job failed, a message containing the correct checkpoint ID for restart is issued by DL/I at the time of failure.
   b. If there was a system failure, the message is issued when MPS is started again in the online partition.
2. Use the VSE checkpoint ID on the VSE RSTRT job control statement. The RSTRT statement is used instead of the EXEC statement when the job is resubmitted for execution.

The job control statements in the following example will restart the program in the previous job control example from checkpoint 0010. Note that the jobname must be the same on the restart job as it was on the job that failed.

```
//  JOB     UPDATE
//  MTC     REW,280
//  ASSGN   SYS100,280
//  RSTRT   SYS100,0010
DLR,INVUPDT,INVMSTR
            •
            •
            DATA CARDS IF REQUIRED
            •
            •
/*
/&
```

For additional information on the function, use, and restrictions of the VSE checkpoint/restart facility, see the *VSE/Advanced Functions Application Programming: User's Guide*.

**Restart Considerations:**

- If an MPS batch program using MPS Restart does not issue a combined checkpoint before a failure, it must be started over from the beginning using the EXEC job control statement rather than the RSTRT statement. For an individual job failure, this is indicated in the message issued at the time of failure. For a system failure, no message is issued for such jobs when MPS is started again in the online partition.
- The VSE restart facility requires the jobname to be the same on a restart job as it was on the job that failed. It also requires that the VSE partition start and end at the same addresses as when the job failed.
- During a VSE restart, a data check (tape checkpoint files) or end-of-file (disk checkpoint files) may occur if the checkpoint ID specified is greater than the actual number of checkpoints taken before the failure.
- On a restart, parameter input is ignored by DL/I, since the parameters were already read and saved when the job first started. However, if the parameter input statement was included on SYSIPT (instead of having been entered from SYSLOG) when the job first started, it is important that one also be included when the job is restarted. This is because DL/I will attempt to position SYSIPT past the parameter input statement when the job is restarted.

## Debugging Your Program

This section presents information that will help you in the task of debugging your DL/I application program. In general, the information applies to all of the possible operating environments. Where this is not true, the exceptions will be noted.

## Problem Determination

The following is a brief discussion of steps that you, as an application programmer, can take when your program fails to run, abnormally terminates, or gives incorrect results.

### Initialization Errors

Before your program receives control, DL/I must have correctly loaded and initialized a nucleus and control blocks. If you suspect a problem in this area, consult your system programming or data base administration functions. Aids are available to them that will help to determine if a problem does exist and to isolate it. Check to see whether there have been any recent changes to DBDs, PSBs, and the control blocks generated from them.

### Execution Errors

If initialization errors do not seem to be present, you should check the following:

1. The output from the translator.
   - All error messages should be resolved.
2. The output from the compiler.
   - All error messages should be resolved.
3. The output from the linkage editor.
   - Are all external references resolved?
   - Have all necessary modules been included?
   - Is the correct entry point specified?
   - For online programs, check that the first module in the link-edit map is DFHEPI (PL/I) or DFHECI (COBOL).
4. Your job control statements.
   - Is the information correct that describes the files that contain the data bases? See data base administration.
   - Are you using the UPSI bit settings correctly?
   - Have you included the SIZE parameter in the EXEC statement and is its value large enough to include your program?
   - Have you included a DL/I parameter statement in the correct format?
5. Your program.
   - Are the literals you are using for arguments in DL/I commands producing the results you expect?
   - If you need help in producing and interpreting a dump, see your system programmer.
   - Make full use of the information in the DIB if your program is producing incorrect results. For more detailed information about the status codes, see the status code summary below.

# Execution Time Debugging Aids

## Status Codes

After processing a DL/I HLPI command, control is returned to your program at the next sequential instruction following the command, unless its execution caused an abend. DL/I places a status code in the status code field of the DIB (DIBSTAT) to indicate the result of the execution of the command.

Figure 3-15 on page 3-46 provides a list of DL/I status codes and is given as a quick reference. These status codes are discussed in detail in *DL/I DOS/VS Messages and Codes.*

Two categories of status codes are listed in Figure 3-15 on page 3-46. The starred status codes will be returned in DIBSTAT. They do not necessarily indicate errors, since the results of the command may be perfectly valid even though the data operated on may not be what you expected. You can test for these codes in your program and proceed accordingly.

All of the other status codes indicate conditions that would cause your program to abend. In that case, the status code would be returned in message DLZ037I.

**Note:** While DL/I is performing index maintenance, it makes internal DL/I calls. If an error occurs in one of these calls, DL/I replaces the first character of the status code returned for that call with "N." This serves as notice to you that an error has occurred, but not in one of the commands executed in your program.

## Key Feedback Length

If KEYFEEDBACK was specified, you can check DIBKFBL against the feedback length specified (FEEDBACKLEN) or defaulted (the KEYFEEDBACK area length) under PL/I. If truncation occurred, use the key feedback area to verify that you are following the right path.

## Abnormal Termination Messages

DL/I also issues execution time error messages. For an explanation of these messages and the required action, consult the *DL/I DOS/VS Messages and Codes.*

## PL/I Diagnostic Information

Your debugging job can be made easier, when the host language of your application program is PL/I, by making use of diagnostic information supplied by both PL/I and DL/I. In batch and MPS batch applications, when a program check is detected during application program execution, a STXIT PC routine will be given control if you have requested STXIT support of DL/I (UPSI bit 7 = 0 for batch, and always for MPS batch).

## CICS/VS Execution Diagnostic Facility (EDF)

If your application program runs in the CICS/VS online environment, the CICS/VS Execution Diagnostic Facility (EDF) is available for your use. This interactive debug tool allows you to see, in EXEC command format, what the program is doing at execution time. Use of EDF should enhance your programming productivity for programs using DL/I HLPI and CICS/VS commands.

**Note:** EDF requires the entry DLZHLPI in the CICS/VS Program Processing Table (PPT) to process the HLPI commands.

***EDF:***

- Displays each command or selected commands and options before the command is executed.
- Allows temporary modification of command arguments before execution.
- Displays each command or selected commands and options after the command is executed.
- Displays, and allows modification to, DIB fields and the program's working storage (including I/O areas) on request.
- Allows redisplay of the last ten commands executed.

For more information on EDF, refer to the *CICS/VS Application Programmer's Reference Manual (Command Level).*

| STATUS CODE | COMMANDS | | | | | | | | | | | COMMAND COMPLETED | ERROR IN CMD or CONVERSION | I/O or SYSTEM ERROR | DESCRIPTION |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | GET UNIQUE | GET NEXT | GET NEXT IN PARENT | DELETE | REPLACE | LOAD | INSERT | CHECKPOINT | SCHEDULE | TERMINATE | | | | | |
| AB | X | X | X | X | X | X | X | | | | | | X | | SEGMENT I/O AREA REQUIRED, NONE SPECIFIED IN COMMAND |
| AC | X | X | X | | | X | X | | | | | | X | | HIERARCHICAL ERROR IN SEGMENT SELECTION |
| AD | | | | | | X | | | X | X | | | X | | INVALID FUNCTION PARAMETER |
| AH | | | | | | X | X | | | | | | X | | COMMAND REQUIRES SEGMENT SELECTION, NONE PROVIDED |
| AI | X | X | X | X | X | X | X | | | | | | | X | DATA MANAGEMENT OPEN ERROR |
| AJ | X | X | X | X | X | X | X | | | | | | X | | INVALID QUALIFIED SEGMENT SELECTION FORMAT |
| AK | X | X | X | | | X | X | | | | | | X | | INVALID FIELD NAME IN COMMAND |
| AM | X | X | X | X | X | X | X | | | | | | X | | COMMAND FUNCTION NOT COMPATIBLE WITH PROCESSING OPTION OR SEGMENT OR PATH SENSITIVITY |
| AO | X | X | X | X | X | X | X | | | | | | | X | I/O ERROR |
| DA | | | | | X | | | | | | | | X | | SEGMENT KEY FIELD HAS BEEN CHANGED |
| DJ | | | | X | X | | | | | | | | X | | NO PRECEDING SUCCESSFUL GET COMMAND |
| DX | | | | X | | | | | | | | | X | | VIOLATED DELETE RULE |
| GA | | ★ | ★ | | | | | | | | | ★ | | | CROSSED HIERARCHICAL BOUNDARY INTO HIGHER LEVEL (RETURNED ONLY ON COMMANDS WITHOUT SEGMENT SELECTION) |
| GB | | ★ | | | | | | | | | | | | | END OF DATA SET, LAST SEGMENT REACHED |
| GE | ★ | ★ | ★ | | | | ★ | | | | | | | | SEGMENT OR PARENT SEGMENT NOT FOUND |
| GK | | ★ | ★ | | | | | | | | | ★ | | | DIFFERENT SEGMENT TYPE AT SAME LEVEL RETURNED (RETURNED ON UNQUALIFIED COMMANDS ONLY) |
| GP | | | X | | | | | | | | | | X | | A GNP COMMAND AND NO PARENT ESTABLISHED, OR REQUESTED SEGMENT LEVEL NOT LOWER THAN PARENT LEVEL |
| II | | | | | | | ★ | | | | | | | | SEGMENT TO INSERT ALREADY EXISTS IN DATA BASE OR IS NON-UNIQUE |
| IX | | | | | | | X | | | | | | X | | VIOLATED INSERT RULE |
| KA | X | X | X | X | X | X | X | | | | | | X | | NUMERIC TRUNCATION ERROR DURING CONVERSION |
| KB | X | X | X | X | X | X | X | | | | | | X | | CHARACTER TRUNCATION ERROR DURING CONVERSION |
| KC | X | X | X | X | X | X | X | | | | | | X | | INVALID PACKED/ZONED DECIMAL CHARACTER DURING CONVERSION |
| KD | X | X | X | X | X | X | X | | | | | | X | | TYPE CONFLICT DURING CONVERSION |
| KE | | | | | X | | | | | | | | X | | REPLACE VIOLATION |
| LB | | | | | | ★ | | | | | | | | | SEGMENT TO INSERT ALREADY EXISTS IN DATA BASE OR IS NON-UNIQUE |
| LC | | | | | | X | | | | | | | | | KEY FIELD OF SEGMENTS OUT OF SEQUENCE |
| LD | | | | | | X | | | | | | | | | NO PARENT FOR THIS SEGMENT HAS BEEN LOADED |
| LE | | | | | | X | | | | | | | | | SEQUENCE OF SIBLING SEGMENT NOT THE SAME AS DBD SEQUENCE |
| NA | | | | | X | | | | | | | | X | | DATA IN SEARCH OR SUBSEQUENCE FIELD HAS BEEN CHANGED |
| NE | | | | ★ | ★ | | ★ | | | | | ★ | | ★ | INDEX MAINTENANCE CANNOT FIND SEGMENT |
| NI | | | | X | X | X | X | | | | | | | X | INDEX MAINTENANCE UNABLE TO OPEN INDEX DATA BASE |
| | | | | | X | X | X | | | | | | X | | DUPLICATE KEY FOUND FOR INDEX DATA BASE |
| NO | | | | X | X | X | X | | | | | | | X | I/O ERROR |
| | | | | | | X | | | | | | | | | LOADING DUPLICATE SECONDARY INDEX POINTER SEGMENT |
| RX | | | | | X | | | | | | | | X | | VIOLATED REPLACE RULE |
| TA | | | | | | | | | X | | | | | X | PSB NOT IN DIRECTORY |

★ Indicates status code returned in DIB.

X Indicates status code that could be expected as an error situation.

*Figure 3-15 (Part 1 of 2). DL/I Status Codes*

| STATUS CODE | COMMANDS | | | | | | | | | | COMMAND COMPLETED | ERROR IN CMD or CONVERSION | I/O or SYSTEM ERROR | DESCRIPTION |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | GET UNIQUE | GET NEXT | GET NEXT IN PARENT | DELETE | REPLACE | LOAD | INSERT | CHECKPOINT | SCHEDULE | TERMINATE | | | | |
| TC | | | | | | | | | X | | | X | | TASK ALREADY SCHEDULED |
| TE | | | | | | | | | X | | | | X | PSB INITIALIZATION ERROR |
| TF | | | | | | | | | X | | | | X | PSB NOT AUTHORIZED |
| TG | | | | | | | | | | ★ | | | | TERMINATE ATTEMPTED WHEN PSB NOT SCHEDULED |
| TH | X | X | X | X | X | X | X | X | | | | X | | DATA BASE COMMAND ATTEMPTED WHEN PSB NOT SCHEDULED |
| TI | | | | | | | X | | | | | X | | INVALID PATH INSERT |
| TJ | X | X | X | X | X | X | X | X | X | X | | | X | DL/I NOT ACTIVE |
| TK | | | | | | | | | X | | | | X | DATA BASE NOT ACTIVE |
| TL | | | | | | | | | X | | | | X | SCHEDULING CONFLICT WITH MPS TASK |
| TN | X | X | X | X | X | X | X | X | X | X | | X | X | INVALID SYSTEM DIB ADDRESS |
| TO | | | | | X | | | | | | | X | | PATH REPLACE ERROR |
| TP | X | X | X | X | X | X | X | X | | | | X | | INVALID PCB INDEX |
| V1 | | | | | X | X | X | | | | | X | | INVALID LENGTH FOR VARIABLE LENGTH SEGMENT |
| V2 | X | X | X | X | X | X | X | | | | | X | | SEGLENGTH MISSING OR INVALID |
| V3 | X | X | X | | | | X | | | | | X | | FIELD LENGTH MISSING OR INVALID |
| V4 | X | X | X | X | X | X | X | | | | | X | | INVALID LENGTH FOR VARIABLE LENGTH SEGMENT |
| V5 | X | X | X | | X | | X | | | | | X | | INVALID OFFSET |
| V8 | X | X | X | | | | | | | | | X | | KEY FEEDBACK LENGTH MISSING OR INVALID |
| XD | | | | | | | | X | | | | | | ERROR DURING DATA BASE BUFFER WRITE OUT |
| XH | | | | | | | | X | | | | | | DATA BASE LOGGING NOT ACTIVE |
| XR | | | | | | | | X | | | | | | ERROR DURING CHECKPOINT PROCESSING FOR MPS RESTART |
| bb | ★ | ★ | ★ | ★ | ★ | ★ | ★ | ★ | ★ | ★ | ★ | | | COMMAND COMPLETED SUCCESSFULLY |

★ Indicates status code returned in DIB.

X Indicates status code that could be expected as an error situation.

For more information on the DL/I Status Codes, refer to DL/I DOS/VS Messages and Codes.

*Figure 3-15 (Part 2 of 2). DL/I Status Codes*

## Other Available DL/I Functions

There are several optional functions available to provide you with more powerful and sophisticated techniques for organizing and processing data bases. However, most of these require that you be an experienced DL/I user in order to take full advantage of them. They have both advantages and disadvantages that should be carefully evaluated before deciding to use them in your application, since your program, other applications, and the overall DL/I system may be affected. Most of these functions are handled as a DBA function, but one, multiple positioning, is an application programming function and is discussed here. Use of multiple positioning requires earlier planning for PSB generation, so you must consult with DBA in the planning stage.

## Multiple Positioning

DL/I provides two alternative methods of maintaining the current position in the data base. These options are called *"single"* and *"multiple"* positioning. The choice is specified during PSB generation.

When *single positioning* is specified for a PCB, DL/I maintains only one position in that data base for that PCB. This is the position that is used in attempting to satisfy all subsequent GET NEXT commands.

If *multiple positioning* is specified, DL/I maintains a unique position in each hierarchical path in the data base.

With single positioning, whenever a segment is obtained, no position is maintained for its dependent segments or any other segments on the same level. Also, no position is maintained under "not-found" conditions. With multiple positioning, whenever a segment is obtained, no position is maintained for its dependent segments, but position for the segments at the same level is maintained. The control blocks are the same in each case (multiple positioning does not require more storage). There is no significant performance difference, even though in some cases multiple positioning requires slightly more processing time.

DL/I attempts to satisfy GET NEXT commands from the existing position by analyzing segments in a forward direction only. Since multiple positioning allows position to be maintained at each level in all hierarchical paths under the current parent position, rather than at each level in only one hierarchical path, the GET NEXT command is satisfied using the existing position established on the path for which the GET NEXT command is qualified. If the GET NEXT command is not qualified, DL/I uses the position established on a path by the prior command.

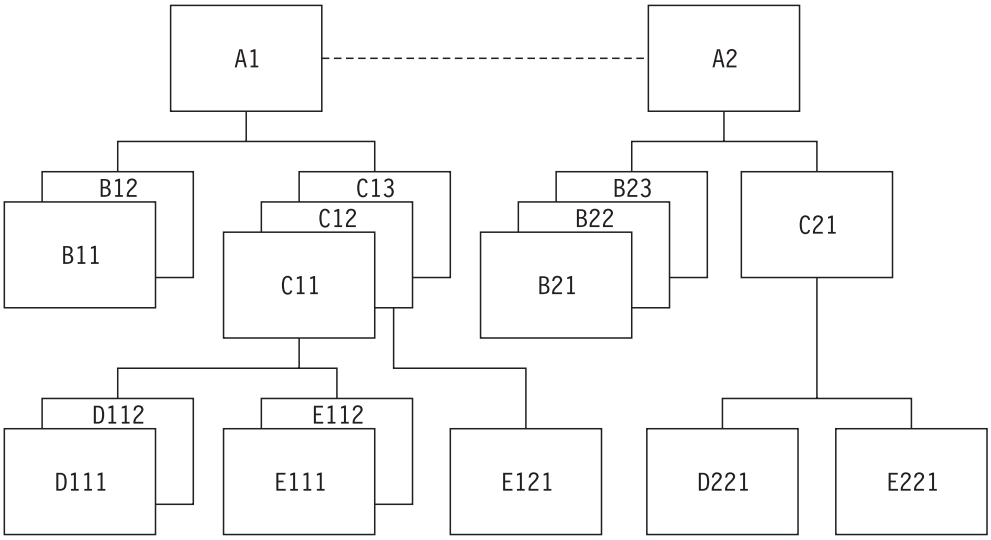



*Figure 3-16. Assumed Data Base to Illustrate Single and Multiple Positioning*

The only effect multiple positioning has on GET UNIQUE and INSERT commands occurs when these commands have missing segment selection specifications in the hierarchical path. The missing levels are completed by the system according to the rules for GET commands.

DELETE and REPLACE commands are not affected by the choice between single and multiple positioning. Rather, any effect is on the GET commands, as described above, since a GET command must be issued prior to a DELETE or REPLACE command.

The following examples compare the results of single and multiple positioning, using the data base in Figure 3-16.

| Command Sequence | Result with Single Positioning | Result with Multiple Positioning |
|---|---|---|
| **Example 1:** | | |
| GET UNIQUE A (KEY=A1) | A1 | A1 |
| GET NEXT B | B11 | B11 |
| GET NEXT C | C11 | C11 |
| GET NEXT B | B21 | B12 |
| GET NEXT C | C21 | C12 |
| **Example 2:** | | |
| GET UNIQUE A (KEY=A1) | A1 | A1 |
| GET NEXT C | C11 | C11 |
| GET NEXT B | B21 | B11 |
| GET NEXT B | B22 | B12 |
| GET NEXT C | C21 | C12 |
| **Example 3:** | | |
| GET UNIQUE A (KEY=A1) | A1 | A1 |
| GET NEXT B | B11 | B11 |
| GET NEXT C | C11 | C11 |
| GET NEXT D | D111 | D111 |
| GET NEXT E | E111 | E111 |
| GET NEXT B | B21 | B12 |
| GET NEXT D | D221 | D112 |
| GET NEXT C | C under next A | C12 |
| GET NEXT E | E under next A | E121 |

Multiple positioning may be useful to you in your application program in two general types of situations.

1. *Increased Data Independence.* Multiple positioning allows you to develop application programs using GET NEXT or GET NEXT IN PARENT commands, and INSERT and GET UNIQUE commands with missing levels; in a manner independent of the relative order of segment types defined at the same level in the data base structure. This means that if performance could be improved by changing the relative order of segment types, and all application programs that access the segment types use multiple positioning, then the change could be made with no impact on previously produced application programs. It should be noted, however, that this ability depends on the proper use of the commands relevant to multiple positioning (GET NEXT, GET NEXT IN PARENT, and incompletely specified INSERT and GET UNIQUE commands). Multiple positioning also adds to your responsibility in that you must keep track of all positions maintained by DL/I. Other alternatives exist to decrease an application program's exposure to future changes. For instance: increased use of explicit command specifications whenever possible. These alternatives may require additional application program coding. Such trade-offs must be determined by you in your own environment.
2. *Parallel Processing of Dependent Segment Types.* When your application program needs to process dependent segment occurrences in parallel (to switch alternately from one dependent segment type to another under a parent) the program may specify multiple positioning to accomplish this processing. An alternative parallel processing technique would be to give the program two or more PCBs using the same data base. Under this alternative, the program processes the data base as though it were two or more different data bases.

This approach may be more useful if the way a segment is updated depends on the analysis of other subsequent segments. The use of multiple PCBs may decrease the number of GET commands required, but may increase the number of other commands required to maintain proper positioning in the two or more data bases. Internal control block processing also increases with each added PCB. You must make the decision on whether to use multiple positioning or multiple PCBs by evaluating your own environment.

Remember that multiple positioning maintains position differently from single positioning. If an application program changes from one option to the other, you must not assume the same results will be produced. An application program must be developed for one alternative or the other.

The multiple positioning feature is intended to be used with DL/I requests that specify segment selection, thereby providing for parallel processing and increased data independence. However, retrieval commands without segment selection may also be used when multiple positioning is specified, to accomplish a sequential retrieval of segment occurrences independent of segment types, if the following considerations are observed:

1. Certain restrictions apply when GET commands without segment selection are mixed with DL/I requests that do specify segment selection in processing a single data base record.

   Example (using Figure 3-16 on page 3-49):

| **Command** | **Result with Multiple Positioning** |
|---|---|
| GET UNIQUE A (KEY=A1) | Retrieves A1 |
| GET NEXT C | Retrieves C11 |
| GET NEXT B | Retrieves B11 |
| GET NEXT B | Retrieves B12 |
| GET NEXT | Unpredictable |

   The GET NEXT commands may not attempt to retrieve occurrences of the C segment type because a position has already been established on this segment type using the multiple positioning feature. The result of the command is unpredictable.

2. When segment types have previously been processed with GET commands not specifying segment selection, a position is established on the last retrieved segment type and its parent (hierarchical path). Multiple positions are no longer maintained.

| **Command** | **Result with Multiple Positioning** |
|---|---|
| GET UNIQUE A (KEY=A1) | Retrieves A1 |
| GET NEXT C | Retrieves C11 |
| GET NEXT B | Retrieves B11 |
| GET NEXT C | Retrieves C12 |
| GET NEXT | Retrieves E121 |
| GET NEXT B | Unpredictable |

   Multiple positions on B are no longer maintained. The result of the GET NEXT B command is unpredictable.

   It should be noted that although the mixed use of GET commands, with and without segment selection, in processing a single logical data base record may

be valid for some types of parallel processing, it may decrease the degree of data independence created by the use of multiple positioning. You should carefully consider the implications of the two restrictions stated above before basing any application programming on mixed use of retrieval with and without segment selection within a single data base record. If possible, GET commands without segment selection should be limited to GET NEXT IN PARENT commands to avoid potentially inconsistent retrieval situations.

# Chapter 4. DL/I HLPI Command Reference

The material in this chapter is reference material having to do with the individual DL/I HLPI commands. It presents, for each command:

- function
- syntax
- segment selection and keys
- status codes
- position pointer
- parentage

## DL/I HLPI Functions

The following reference material provides detailed information about each of the DL/I HLPI commands under six headings:

1. *Function* tells you what the command is designed to do.
2. *Syntax* tells you how to code the command in your program.
3. *Segment selection or segment selection and keys* tells you of the effect of the inclusion or omission of qualified and unqualified segment selection specifications.
4. *Status codes* tells you which status codes are returned in the DIB by DL/I as a result of each command, and what their significance is to you. The status codes representing conditions that would cause your application to abend are listed separately. In this case, the status code would be returned in message DLZ037I.
5. *Position pointer* tells you what effect, if any, the command has on the position pointer that normally points to the next sequential segment, following the successful or unsuccessful execution of the command.
6. *Parentage* tells you what effect, if any, the successful or unsuccessful execution of the command has on the establishment of parentage. The operation of subsequent GET NEXT IN PARENT commands depends on previously established parentage.

## GET NEXT

1. *Function:* You can use this command to sequentially retrieve the next sensitive segment from a data base in a forward direction. Sequential retrieval within a data base hierarchy always proceeds from top-to-bottom, and from left-to-right.

2. *Syntax:* The syntax for the GET NEXT command looks like this:

```
  {EXECUTE} DLI  {GET NEXT}  [USING PCB(exp)]
  {EXEC   }      {GN      }
  [KEYFEEDBACK(ref) [FEEDBACKLEN(exp)]]

[[{FIRST}][VARIABLE]SEGMENT(name)
  {LAST }

  [INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]]
  [WHERE(name op ref[{AND}name op ref]...)
                     {OR }

  [FIELDLENGTH(exp[,exp]...)]]]...

[[{FIRST}][VARIABLE]SEGMENT(name)]
  {LAST }

   INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]
  [WHERE(name op ref[{AND}name op ref]...)
                     {OR }

  [FIELDLENGTH(exp[,exp]...)]]...

  {;       }
  {END-EXEC}
```

   **Note:** If a parent segment is specified, an object segment must also be specified.

3. *Segment selection:* If you do not specify segment selection, the next sequential sensitive segment is retrieved. The next sequential segment is the one that the position pointer was pointing to at the completion of the last successful command.

   **Note:** In HDAM, root segments are returned in physical sequence of data base records, rather than in root key sequence, unless the randomizing module was designed to maintain key sequence. Input or output sorting or secondary indexing can be used if key sequencing is required.

   - If you specify a single unqualified segment, the first occurrence of that segment type found by searching in a forward direction from the current position is retrieved.
   - If you specify segment selection for one or more segments, the path leading to the segment retrieved is defined by the segments. The last segment type specified is retrieved.
   - It is suggested that segment selection be specified because of documentation, control, and future change considerations. (See "Techniques and Suggestions," in Chapter 3.) However, the presence or absence of unqualified parent segments has no effect on the operation. Only qualified parent segments and the object segment are used by DL/I to determine the path and retrieve the object segment. In other words, missing or unqualified parent segments indicate that *any* correct path to the object segment will satisfy the command.

- When the object segment is qualified, it defines the segment occurrence that is to be retrieved. Qualified parent segments define the segment occurrences that are to become a part of the path to the object segment.
- Do not code VARIABLE without specifying an associated INTO or FROM option.

4. *Status codes:*

   **GA**
   A command with no segments specified has retrieved a segment that is at a higher level in the hierarchy than the previous segment retrieved. GA serves as a warning that the position in the data base has changed with respect to the path that existed previously.

   **GB**
   The end of the data base has been reached. No segment was retrieved. If, however, the GET NEXT command specified a qualified root segment with the field name referring to a key field (not a data field) and the relational operator was ¬=, =¬, ¬, or =; and the end of the data base was reached without locating the segment, the status code would be GE instead of GB. At the end of the data base, you can continue processing by issuing a GET UNIQUE or GET NEXT command. However, if the status code returned for the command that reached the end of the data base was GE, a following GET NEXT command should not specify a qualified segment, so that DL/I will begin at the start of the data base. The GET NEXT command will then start searching from the beginning of the data base for the next segment that satisfies the conditions of the command. You must remember that, unless a segment satisfying these conditions is found, the whole data base will be searched.

   **GE**
   You specified a qualified segment that was not found. This could be because the segment does not exist, because the segment specified cannot be found by searching forward from the previous established position, or because the current position pointer is pointing to a segment that is forward in the data base of any possible existence of the specified segment. This status code is also returned for the programming error of specifying higher level qualified or unqualified segments that differ from one or more of the segments in the currently established parentage path.

   **GK**
   A segment has been retrieved that has the same higher level path as the previous segment processed, but is a different type of segment at the same hierarchical level. This status code is returned only as a warning for calls with no segment specified and merely indicates that the program is working with a different segment type.

   **bb** The correct segment was retrieved.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position Pointer:* Following a GET NEXT command in which a segment was retrieved, the position pointer points to the next segment in the data base. If

the GB status code (end-of-data-base) was returned, it points to the first segment in the data base. Following an unsuccessful command, the position pointed to is unpredictable and remains so until the next successful command.

6. *Parentage:* Parentage is set to the segment retrieved, and can be used by a subsequent GET NEXT IN PARENT command. If this command was unsuccessful, the previous parentage, if any, is destroyed.

## GET NEXT IN PARENT

1. *Function:* You can use this command to sequentially retrieve the next sensitive segment from a data base in a forward direction, under the established parentage.
2. *Syntax:* The syntax for the GET NEXT IN PARENT command looks like this:

```
{EXECUTE} DLI  {GET NEXT IN PARENT}  [USING PCB(exp)]
{EXEC   }      {GNP                }
[KEYFEEDBACK(ref) [FEEDBACKLEN(exp)]]

[[{FIRST}][VARIABLE]SEGMENT(name)
 {LAST }

 [INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]]
 [WHERE(name op ref[{AND}name op ref]...)
                    {OR }

 [FIELDLENGTH(exp[,exp]...)]]...

[[{FIRST}][VARIABLE]SEGMENT(name)]
 {LAST }

  INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]
 [WHERE(name op ref[{AND}name op ref]...)
                    {OR }

 [FIELDLENGTH(exp[,exp]...)]]...

 {;       }
 {END-EXEC}
```

   **Note:** Parentage must have been established by a successful GET UNIQUE or GET NEXT command either immediately before the present one, or at some prior time, provided no other command that changes parentage has intervened.

3. *Segment selection:*
   - If you do not specify a segment, the next sequential segment under the established parentage path is retrieved. The next segment is the one that the position pointer was pointing to at the completion of the last successful command.
   - If you specify a single unqualified segment, the first occurrence of that segment type found by searching in a forward direction within the established parentage path is retrieved.
   - When you specify unqualified parent segments in a command with multiple parent segments, they establish the first occurrence of the associated segment type as a part of the path.
   - If you omit parent segments between the previously established parentage and the object segment, DL/I generates unqualified segment selection for

these segments. This has the effect of establishing the first occurrence of those parent segments as a part of the path.

- When the object segment is qualified, it defines the segment occurrence that is to be retrieved. Qualified parent segments define the segment occurrences that are to become a part of the path to the object segment.
- Do not code VARIABLE without specifying an associated INTO or FROM option.

4. *Status codes:*

   **GA**
   A command with no segments specified has retrieved a segment that is at a higher level in the hierarchy than the previous segment retrieved, but still below the parentage segment previously established. This code serves as a warning that the position of the data base has changed with respect to the path that previously existed from the established parentage to the last segment retrieved.

   **GE**
   A qualified or unqualified segment within the previously established parentage was not found by searching forward from the established position, or the segments beneath the established parentage have been exhausted without locating the specified segment.

   **GK**
   A segment has been retrieved that is within the previously established parentage and has the same higher level path as the previously processed segment, but is a different type of segment at the same hierarchical level. This status code is returned only for commands with no segment specified. The code merely indicates that the program is now working with a different segment type.

   **bb** The correct segment was retrieved.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position Pointer:* If a segment was retrieved, the position pointer points to the next segment in the data base. If the GE (no-segment-found) status code was returned, it points to the segment that caused the no-segment-found condition to be recognized. This is either a segment with a higher key than was specified in a qualified segment, or is the first segment located outside of the established parentage in a forward direction.

6. *Parentage:* Parentage is not changed or affected.

## GET UNIQUE

1. *Function:* You can use this command to randomly retrieve any sensitive segment in the data base. The GET UNIQUE command is the only command that has this random access capability. Because of it, GET UNIQUE can be used to reposition the position pointer to any sensitive segment.

2. *Syntax:* The syntax for the GET UNIQUE command looks like this:

```
{EXECUTE} DLI {GET UNIQUE} [USING PCB(exp)]
{EXEC   }     {GU        }
 [KEYFEEDBACK(ref) [FEEDBACKLEN(exp)]]

[[LAST][VARIABLE]SEGMENT(name)
 [INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]]
  [WHERE(name op ref[{AND}name op ref]...)
                     {OR }

  [FIELDLENGTH(exp[,exp]...)]]]...

 [LAST][VARIABLE]SEGMENT(name)
  INTO(ref)[LOCKED][OFFSET(exp)][SEGLENGTH(exp)]
  [WHERE(name op ref[{AND}name op ref]...)
                     {OR }

  [FIELDLENGTH(exp[,exp]...)]]...

 {;       }
 {END-EXEC}
```

   **Note:** To establish position at the beginning of the data base, issue a GET UNIQUE command specifying unqualified segment selection for the root segment type.

3. *Segment selection:*

   - You can specify any number of segment types up to the number of hierarchical levels defined in the PCB.
   - If you specify one unqualified segment type, the object segment under the first root segment in the data base is retrieved. If you specify multiple segments, the root segment identified by the first segment becomes the parent root segment for the child subsequently retrieved.
   - If you specify multiple segments, the object segment retrieved has the path defined by the parent segments.
   - If you specify an unqualified segment, the first occurrence of that segment type is retrieved.
   - If you omit one or more parent segments, DL/I will generate implied segment selection for the missing levels.
     - If the previous command established the same path as the path to the object segment of this command, unspecified segments in the path will be qualified as in the previous command.
     - If the previous command established a different path than the path to the object segment in this command, unspecified segments in the path will be unqualified.
   - Do not code VARIABLE without specifying an associated INTO or FROM option.

4. *Status codes:*

   **GE**
   No segment was found that satisfies the segment selection specified.

   **bb** The correct segment was retrieved.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position pointer:* After a successful GET UNIQUE command, the position pointer points to the next logical segment in the data base. If the call was unsuccessful, the position pointer is set to the position beyond the segment that caused the GE status, and remains so until the next successful execution of a command that sets the pointer. For this reason, if the status code is GE, you may want to cause your program to execute such a command to reestablish position.

6. *Parentage:* Parentage is set to the segment retrieved, and can be used by a subsequent GET NEXT IN PARENT command. If this command was unsuccessful, the previous parentage, if any, is destroyed.

## INSERT

1. *Function:* You can use this command to add a new segment to an existing data base; either randomly or sequentially, after you have built it in an I/O area. The PSB generated by DBA determines that the command is to add data to an existing data base, rather than to load a new one.

2. *Syntax:* The syntax for the INSERT command looks like this:

```
  {EXECUTE} DLI  {INSERT}  [USING PCB(exp)]
  {EXEC   }      {ISRT  }

[[{FIRST}][VARIABLE]SEGMENT(name)
  {LAST }

  [FROM(ref)[SEGLENGTH(exp)]]
  [WHERE(name op ref[{AND}name op ref]...)
                     {OR }

  [FIELDLENGTH(exp[,exp]...)]]]...

 [{FIRST}][VARIABLE]SEGMENT(name)
  {LAST }

   FROM(ref)[OFFSET(exp)][SEGLENGTH(exp)]

  {;       }
  {END-EXEC}
```

   **Notes:**

   a. FROM and WHERE must not be specified for the same parent segment.

   b. For a path call INSERT command, data must be transferred (using the FROM option) for all segments between the highest level parent segment with FROM specified, and the object segment.

3. *Segment Selection and Keys:*

   **Note:** For details about logical relationships and the RULES parameter, which are mentioned below, refer to *DL/I DOS/VS Data Base Administration.*

   - At least unqualified segment selection must be specified for each segment being added.
   - If a complete set of qualified segments is specified for the segment being inserted, it is inserted at its designated location in the data base.
   - If multiple parent segments are specified for an insertion, they may be a mixture of qualified and unqualified segments. However, the last segment must be unqualified.
   - If unqualified parent segments are used to establish a path to an insertion, the first occurrence of the segment type satisfies the path definition.
   - If parent segments are omitted, the current position in the data base is used to develop implied segment selection. If the current position in the data base is not correct because higher level segment selection has changed the position in the data base, then implied unqualified segments are developed for the first occurrence of the segment type that falls within the newly established path.
   - Before the INSERT command is issued, the segment being inserted must be built in its I/O area, and, if it has a key, its correct key must be placed in the proper location in the I/O area.
   - If the segment being inserted is a root segment, the correct place for its insertion in the data base is determined by the key taken from its I/O area.
   - If the segment being inserted is not a root segment, but its immediate parent has just been inserted, it may be inserted as soon as it is built in the I/O area merely by specifying unqualified segment selection for it in the INSERT command.
   - When inserting a logical child segment through the physical path, the I/O area must contain the logical parent's concatenated key, followed by the logical child segment.
   - When inserting a logical child segment through the logical path, the I/O area must contain the physical parent's concatenated key, followed by any logical child data.
   - If an unqualified segment is being inserted, the position pointer must point to the right place in the data base so that the segment's logical location in the data base can be found by searching forward or backward in the current record.
   - Segments that have no keys are always inserted following the last segment of the same type unless the DBD generated for this data base states RULES=FIRST or HERE, in which case the specified rule applies. Segments having equal keys are also treated in this manner.
   - Do not code VARIABLE without specifiying an associated INTO or FROM option.

4. *Status codes:*

   **GE**
   The path specified by multiple parent segments was not found. This status code is normally associated with one or more qualified parent segments that define the path to its insertion. However, it can be returned when only unqualified segments are used, if no segment of the parent segment type exists.

   **II** The segment already exists. This code generally indicates a duplicate segment, although it could occur if an INSERT command was issued for a segment before a proper path had been established for it. The segment could possibly match a segment with the same key in another hierarchy or record. The command is not completed.

   **NE**
   During some previous insert call, an index source segment was inserted with data in search and subsequence fields equal to an already existing index source segment. An NI status code had been returned with that call.

   **bb** The segment was inserted in the proper place as specified by the given or implied segment selection specification.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position pointer:* After a successful INSERT command, the position pointer points to the segment immediately following the segment just inserted. This could be another segment of the same type if the segment was inserted in the middle of a group of the same segment types by its key, or to the next segment type in the hierarchy if the segment was inserted as the last segment of the group.

6. *Parentage:* Previously established parentage is not changed or affected, providing the segments being inserted are inserted below the lowest level parent in the established parentage path. If a segment is inserted outside the currently established path, then parentage is destroyed and must be reestablished before GET NEXT IN PARENT commands can be issued.

## REPLACE

1. *Function:* You can use this command to replace, update, or rewrite the last segment successfully retrieved from the data base by a GET UNIQUE, GET NEXT, or GET NEXT IN PARENT command using the same PCB.

2. *Syntax:* The syntax for the REPLACE command looks like this:

```
{EXECUTE} DLI {REPLACE} [USING PCB(exp)]
{EXEC   }     {REPL   }

[[VARIABLE]SEGMENT(name)
  FROM(ref)[OFFSET(exp)][SEGLENGTH(exp)]]...

 [VARIABLE]SEGMENT(name)
  FROM(ref)[OFFSET(exp)][SEGLENGTH(exp)]

{;       }
{END-EXEC}
```

   **Notes:**

   a. The segment being replaced must be successfully retrieved by a GET command before the REPLACE command is executed, or the REPLACE command will be rejected. Only one REPLACE command can be executed for each GET command.

   b. No other DL/I commands using the same PCB can be executed between the GET and the associated REPLACE command.

   c. Only segments specified in the associated GET command can be specified in the REPLACE command.

   d. Not all of the segments specified in the associated GET command have to be replaced.

   e. DIBSEGM and DIBSEGLV are not updated by the REPLACE command.

   f. Do not code VARIABLE without specifying an associated INTO or FROM option.

3. *Segment selection:* Use unqualified segment selection to specify the segment or segments to be replaced, from those just retrieved by a GET command.

4. *Status codes:*

   **NE**
   An index source segment was replaced, but a corresponding index pointer segment could not be found.

   **bb** Segment was successfully replaced.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position Pointer:* The current position pointer is not changed by either a successful or an unsuccessful REPLACE command. It continues to point to the next segment in the data base following the replaced segment.

6. *Parentage:* Parentage is not changed by a REPLACE command.

## DELETE

1. *Function:* You can use this command to delete the last segment successfully retrieved from the data base by a GET UNIQUE, GET NEXT, or GET NEXT IN PARENT command using the same PCB.

2. *Syntax:* The syntax for the DELETE command looks like this:

```
{EXECUTE} DLI {DELETE} [USING PCB(exp)]
{EXEC   }     {DLET  }

[VARIABLE]SEGMENT(name)
 FROM(ref)[SEGLENGTH(exp)]

{;       }
{END-EXEC}
```

   **Notes:**

   a. The segment being deleted must be successfully retrieved by a GET command before the DELETE command is executed, or the DELETE command will be rejected. Only one DELETE command can be executed for each GET command.

   b. No other DL/I commands using the same PCB can be executed between the GET and the associated DELETE command.

   c. This command deletes all of the deleted segment's physically dependent segments or children at all levels beneath it, *whether they are sensitive segments or not.* The previously retrieved segment and its dependent segments are removed from the data base and cannot be retrieved or used as parents again.

   d. Do not code VARIABLE without specifying an associated INTO or FROM option.

3. *Segment selection:* You must use unqualified segment selection to specify the segment to be deleted, from the segments just retrieved by a GET command.

4. *Status codes:*

   **NE**
   An attempt was made to delete an index source segment and the corresponding index pointer segment could not be found. The cause could be an error during reorganization of the data base or previous insertion of an index source segment with data in search and subsequence fields equal to an already existing index source segment.

   **bb** The segment specified, and all of its dependent segments (if any) have been successfully deleted.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position Pointer:* The current position pointer is not changed by an unsuccessful DELETE command. After a successful deletion, the pointer may continue to point to the next segment in the data base after the deleted segment or, if dependent segments were also deleted, to the segment after the last dependent segment deleted.

6. *Parentage:* Parentage is not changed by a DELETE command unless the last established parent is deleted; if so, parentage must be reestablished.

## LOAD

1. *Function:* You create a new data base by writing an application program containing LOAD commands that load segments from their segment I/O areas after being built there. The PSB generated for the program as a DBA function specifies that the program is loading a new data base, as opposed to adding to an existing one.
2. *Syntax:* The syntax for the LOAD command looks like this:

```
{EXECUTE} DLI LOAD [USING PCB(exp)]
{EXEC   }

[VARIABLE]SEGMENT(name)
 FROM(ref)[SEGLENGTH(exp)]

{;       }
{END-EXEC}
```

   **Notes:**

   a. The LOAD command can only be used in batch programs.

   b. Segment name can be a variable, in which case it must be enclosed in double parentheses. For example: SEGMENT((VARNAME)).

3. *Segment selection and keys:*

   **Note:** For details about logical relationships, which are mentioned below, refer to *DL/I DOS/VS Data Base Administration.*

   - Before your program executes a LOAD command, the segment you intend to load with that command must have been built in its I/O area. If the segment has a key, its correct value must have been placed in the proper location in the I/O area.
   - When you load a logical child segment, the I/O area must contain the logical parent's concatenated key, followed by the logical child segment to be inserted.
   - You must load all segments having keys in sequence by key value.
   - When you load segments that do not have keys, they are loaded in the data base in the order in which their LOAD commands are processed.
   - Dependent segment types, as specified by segment selection, must be loaded in hierarchical order. Their parent segments must have been loaded before you can load them.

4. *Status codes:*

   **LB**
   The segment you are trying to load already exists in the data base. You cannot load a duplicate segment. This status code only applies to segment types having key fields.

   **bb** The segment was loaded successfully.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position pointer:* When a LOAD command is executed, the position pointer points to the next available space following the last segment successfully loaded. The next segment loaded will be placed in that space.

6. *Parentage:* Since only LOAD commands can be issued to a data base being loaded, parentage is not significant, and is not set.

## CHECKPOINT

1. *Function:* The CHECKPOINT command is used to establish a point of reference, during the execution of your program, that indicates that all data base operations prior to that point were completed satisfactorily. CHECKPOINT can be issued at any appropriate points in your program as determined by you. It can be issued in batch, MPS batch, or online tasks. The exact operations performed when the CHECKPOINT is executed are different in the different environments.

## Batch

In DL/I batch application programs, the CHECKPOINT command causes a checkpoint record to be written on the DL/I log as an aid in restart processing. The data base buffers will be written to the data base and a checkpoint log record, with a user supplied unique checkpoint identification, will be written to the DL/I log.

The eight-character checkpoint identification just mentioned may be any value. However, an EBCDIC character string is recommended because the value is used in messages to the operator and/or programmer. Each time a batch application program issues a CHECKPOINT command, DL/I issues a message, DLZ105I, that contains the checkpoint identification. This id should be noted and saved, because it may be required to aid in backout and restart processing.

In case of a failure in the batch environment, the backout utility may be run to back out data base changes occurring since the last checkpoint. The utility will not back out data base changes occurring before the last checkpoint.

## MPS Batch and Online

For MPS batch and online tasks running with CICS/VS journaling active, a CHECKPOINT command is, in effect, a CICS/VS sync point call with the exception that the task's PSB scheduling status is not changed. Therefore, if an online task has a scheduled PSB in effect at the time the CHECKPOINT command is issued, a PSB SCHEDULE command is not required after it. In fact, a SCHEDULE command issued under these circumstances would cause a scheduling error.

It is recommended that DL/I logging not be used for MPS batch and online tasks. With DL/I logging active, a CHECKPOINT command causes data base buffers to be written to secondary storage and a checkpoint log record to be written to the DL/I log, as in the batch environment. However, these functions are not usable for performing backout because batch backout cannot be used in an online environment. Backout for an MPS batch or online DL/I task can only be performed using CICS/VS dynamic transaction backout, which requires that CICS/VS journaling be active.

For MPS batch tasks, the message DLZ105I is written as for a batch program. For online tasks, DLZ105I is not written. Therefore, online application programs must include their own provisions for identifying where restart is to begin following a CHECKPOINT command.

## MPS Batch Using MPS Restart

In order to provide a restart capability for MPS batch users, DL/I supports the use of VSE checkpoint/restart in conjunction with the DL/I CHECKPOINT command. A VSE checkpoint writes a copy of the partition in which the checkpoint was issued to disk or tape. The VSE RSTRT job control statement reloads this copy into the partition and passes control to the restart address that was specified when the VSE checkpoint was issued. For COBOL and PL/I programs using their respective VSE checkpoint interfaces, this corresponds to the next program statement after the one which invokes the VSE checkpoint.

MPS Restart provides the following functions:

- Combined Checkpoint Verification

  MPS Restart verifies that a VSE checkpoint is issued immediately before each DL/I checkpoint command. This is called a combined checkpoint. A VSE checkpoint may be issued in PL/I and COBOL by using the checkpoint interfaces provided by those languages. See the “Programming Examples” section of this book for examples of how to code combined checkpoints.

- MPS Batch Reinitialization

  The first DL/I command executed following a VSE restart must be a DL/I checkpoint command. This will be the normal sequence when VSE checkpoints are placed immediately before each DL/I checkpoint. The DL/I checkpoint command will automatically determine that a VSE restart has occurred and reinitialize the MPS batch environment. Following successful reinitialization, the checkpoint command will return control to the application program as if from a normal checkpoint and the program may continue processing.

- Checkpoint ID Notification and Verification

  Besides the normal SYSLOG messages issued by VSE and DL/I when checkpoints are taken, a message containing the checkpoint identifier (ID) of the last successful combined checkpoint will be issued when a failure occurs. For individual job failures, the message is issued at the time of the failure. For system-wide failures, it is issued when MPS is started again in the online partition. The checkpoint ID contained in this message must be specified as a parameter on the VSE RSTRT job control statement when restarting an MPS batch job. MPS Restart will verify whether the checkpoint ID used for restart is the correct one. If it is not, DL/I will indicate the correct checkpoint ID which must be used and will cancel the job. This allows you to restart the job again, using the correct checkpoint ID.

  For additional information on using the MPS Restart facility, see *DL/I DOS/VS Data Base Administration*.

## Restrictions on Using VSE Checkpoint/Restart

Certain restrictions exist on the use of VSE checkpoint/restart. For example, VSAM files must be closed before a VSE checkpoint is issued. DL/I data bases used by MPS programs are in the online partition and are not affected by this restriction.

Also, VSE restart cannot be used to restart programs which failed because of program logic errors. This is because a copy of the program, exactly as it was before it failed, is loaded back into the partition during a restart.

For details on these and other restrictions, see the *VSE/Advanced Functions Application Programming: Reference*, and the *VSE/Advanced Functions Application Programming: User's Guide*.

1. *Syntax:* The syntax for the CHECKPOINT command looks like this:

```
{EXECUTE} DLI {CHECKPOINT}
{EXEC   }     {CHKP      }

 ID(exp)

{;       }
{END-EXEC}
```

   **Note:** Under MPS Restart, since a DL/I checkpoint will always be issued after a VSE checkpoint has been taken, it is recommended that the VSE checkpoint ID also be used as the DL/I checkpoint ID in PL/I programs. This is not possible in COBOL since the VSE checkpoint ID is not returned to the application program as it is in PL/I. Using the VSE checkpoint ID on the DL/I CHKP command provides a cross reference between the VSE and DL/I checkpoint messages issued to SYSLOG.

2. *Segment selection:* No segments are specified in the CHECKPOINT command.

3. *Status codes:*

   **bb** A checkpoint was successfully taken.

   **Other**
   If the status code is other than a **bb**, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

4. *Position Pointer:* Current position in the data base is lost upon return from a CHECKPOINT command. If the PSB being used has more than one PCB defined, position is lost across all PCBs and not just the PCB used in the checkpoint command. Position must be reestablished (usually by a GET UNIQUE command) before continuing with data base processing after a CHECKPOINT command.

5. *Parentage:* Parentage has to be reestablished after a CHECKPOINT command.

## SCHEDULE

1. *Function:* In an online application, you must schedule a PSB before you issue any DL/I HLPI data base function commands. You use the SCHEDULE command for this purpose.
2. *Syntax:* The syntax for the SCHEDULE command looks like this:

```
{EXECUTE} DLI {SCHEDULE}
{EXEC   }     {SCHD    }

 PSB(name)

{;        }
{END-EXEC}
```

   **Note:** The SCHEDULE command can only be used in online programs.
3. *Segment selection:* No segments are specified in the SCHEDULE command.
4. *Status codes:*

   **bb** The PSB was successfully scheduled.

   **Other**
   If the status code is other than a **bb**, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.
5. *Position pointer:* Position is not established by the SCHEDULE command.
6. *Parentage:* Parentage is not established by the SCHEDULE command.

## TERMINATE

1. *Function:* You use the TERMINATE command in an online application to indicate to DL/I that all modifications made to the data bases by the transaction to this point are committed and cannot be backed out, and that you are releasing the PSB for use by another task.

   **Note:** A TERMINATE command causes a CICS/VS synchronization point. Also, a CICS/VS synchronization point causes a TERMINATE if a PSB is still scheduled by the transaction. Refer to the *CICS/VS Recovery and Restart Guide.*

2. *Syntax:* The syntax for the TERMINATE command looks like this:

```
{EXECUTE} DLI {TERMINATE}
{EXEC   }     {TERM     }

{;        }
{END-EXEC}
```

   **Note:** The TERMINATE command can only be used in online programs.

3. *Segment selection:* No segments are specified in the TERMINATE command.

4. *Status codes:*

   **TG**
   A TERMINATE command was issued when a PSB had not previously been scheduled for the task.

   **bb** The command was successfully completed.

   **Other**
   If the status code is other than one of those listed above, DL/I will report the code in message DLZ037I, then terminate your program.

   See “Status Codes” in Chapter 3 for a complete list of status codes.

5. *Position pointer:* Position is not established by the TERMINATE command.

6. *Parentage:* Parentage is not established by the TERMINATE command.

# Glossary

A number of terms and phrases used in describing DL/I DOS/VS are either new to most readers, or have new meanings. To improve the readability and your understanding of this and other DL/I DOS/VS publications, the significant and important terms are defined in this Glossary. Some of the definitions refer to the representative DL/I hierarchical structure shown in Figure X-1.

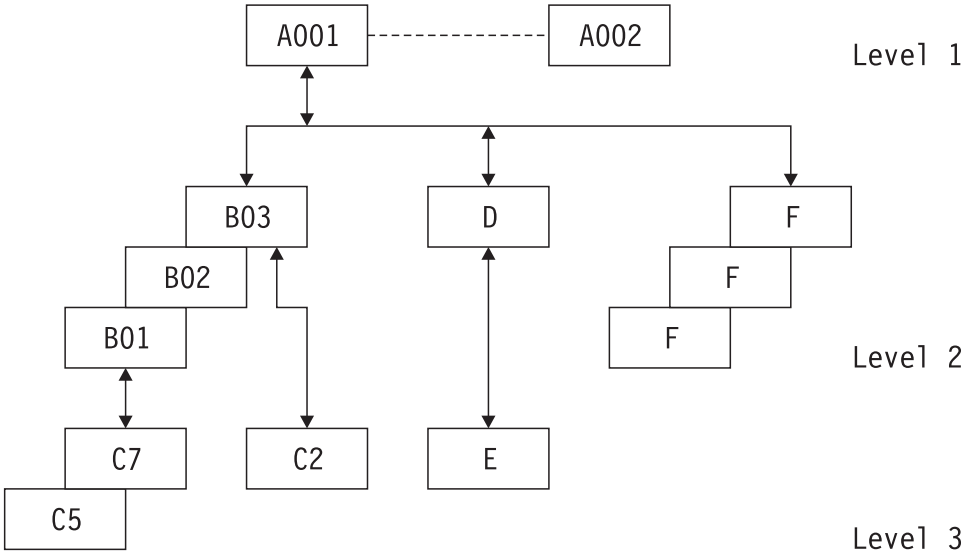



*Figure X-1. Representative DL/I Hierarchical Structure*

**ACB**. (1) Application control blocks (DL/I). (2) Access method control block (VSAM).

**ACBGEN**. Application control block generation.

**access method control block (ACB).**. A control block that links a program to a VSAM data set.

**access method services**. A multifunction utility program that defines VSAM data sets (or files) and allocates space for them, and lists data set records and catalog entries.

**ACT**. Application control table.

**addressed direct access**. In systems with VSAM, the retrieval or storage of a data record identified by its relative byte address, independent of the record's location relative to the previously retrieved or stored record. (See also *keyed direct access, addressed sequential access, keyed sequential access,* and *relative byte address.)*

**addressed sequential access**. The retrieval or storage of a VSAM data record relative to the previously retrieved or stored record. (See also *keyed sequential access, addressed direct access,* and *keyed direct access.)*

**aggregate**. See *data aggregate.*

**anchor point (AP)**. See *root anchor point.*

**application control blocks**. The control blocks created from the output of DBDGEN and PSBGEN, e.g., a DMB of an internal PSB created by the ACB utility program.

**application control block generation (ACBGEN)**. The process by which application control blocks are created.

**application control table (ACT)**. A DL/I online table describing those CICS application programs that utilize DL/I.

**argument**. (1) (ISO)[00] An independent variable. (2) (ISO)[00] Any value of an independent variable . (3) Information, such as names, constants, or variable values included within the parentheses in a DL/I command.[1]

[1] International Organization for Standardization, Technical Committee 97/Subcommittee 1.

**attribute**. A property of an entity expressing a value. Synonymous with *field*.

**backout**. The process of removing all the data base updates performed by an application program that has terminated abnormally. See also *dynamic backout.*

**batch checkpoint/restart**. The facility that enables batch processing programs to synchronize checkpoints and to be restarted at a user-specified checkpoint.

**batch processing**. A processing environment in which data base transactions requested by applications are accumulated and then processed periodically against a data base.

**Boolean operator**. (1) (ISO)[2] An operator, each of the operands of which and the result of which, take one of two values. (2) An operator that represents symbolically relationships, such as AND, OR, and NOT, between entities.

**business process**. A defined function of a business enterprise usually interrelated through information requirements with other business processes. For example, personnel management is the business process responsible for employee welfare from pre-hire through retirement. It is related to the accounting business process through payroll.

**CA**. Control area.

**call**. (1) (ISO)[2] The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (2) (ISO)[2] In computer programming, to execute a call. (3) The instruction in the COBOL, PL/I, or Assembler program that requests DL/I services. For RPG II, see *RQDLI command.* See also *command.*

**control area (CA)**. A collection of control intervals. Used by VSAM to distribute free space.

**checkpoint**. A time at which significant system information is written on the system log, and optionally, the system shut down.

**child**. Synonymous with *child segment.*

**child segment**. A segment one level below the segment which is its parent, with a direct path back up to the parent. Depending on the structure of the data base, a parent may have many children; however, a child has only one parent segment. Referring to Figure X-1 on page X-1:

- All the B, D, and F segments are children of A-001.
- C-5 and C-7 are children of B-01 (and A-001) but not children of the other B segments.
- B-02 has no children.

See also *logical child* and *physical child.*

**CI**. Control interval.

**combined checkpoint**. In MPS batch programs, a VSE checkpoint followed immediately by a DL/I CHKP command. The two checkpoints together form one logical checkpoint, and allow the use of the VSE restart facility to restart MPS batch programs.

**command**. The statement in DL/I High Level Programming Interface (HLPI) that requests services for application programs written in COBOL or PL/I. See also *call.*

**command code**. An optional addition to the SSA that provides specification of a function variation applicable to the call function.

**concatenated key**. The key constructed to access a particular segment. It consists of the key fields, including that of the root segment and successive children down to the accessed segment.

**control interval (CI)**. (1) A fixed length amount of auxiliary storage space in which VSAM stores records and distributes free space. (2) The unit of information transmitted to or from auxiliary storage by VSAM.

**data aggregate**. A group of data elements that describe a particular entity. Synonymous with *segment.* See also *data element.*

**data base (DB)**. (1) (ISO)[2] A set of data, part of the whole of another set of data, and consisting of at least one file, that is sufficient for a given purpose or for a given data processing system. (2) A collection of data records comprised of one or more data sets. (3) A collection of interrelated or independent data items stored together without unnecessary redundancy to serve one or more applications. See *physical data base* and *logical data base.*

**data base administration (DBA)**. The tasks associated with defining the rules by which data is accessed and stored. The typical tasks of data base administration are outlined in *DL/I DOS/VS Data Base Administration.*

**data base administrator (DBA)**. The person in an installation who has the responsibility (full or part time) for technically supporting the use of DL/I.

---

[2] International Organization for Standardization, Technical Committee 97/Subcommittee 1.

**data base description (DBD)**. A description of the physical characteristics of a DL/I data base. One DBD is generated and cataloged in a core image library for each data base that is used in the installation. It defines the structure, segment keys, physical organization, names, access method, devices, etc., of the data base.

**data base integrity**. The protection of data items in a data base while they are available to any application program. This includes the isolation of the effects of concurrent updates to a data base by two or more application programs.

**data base organization**. The physical arrangement of related data on a storage device. DL/I data base organizations are hierarchical direct (HD) and hierarchical sequential (HS). See *hierarchical direct organization* and *hierarchical sequential organization.*

**data base record**. A collection of DL/I data elements called segments hierarchically related to single root segments.

Referring to Figure X-1 on page X-1, A-001, B-01, C-5, C-7, B-02, B-03, C-2, D, E, F, F, F constitute a data base record.

**data base reorganization**. The process of unloading and reloading a data base to optimize physical segment adjacency, or to modify the DBD.

**data communication (DC)**. A program that provides terminal communications and automatic scheduling of application programs based on terminal input. For example, CICS/DOS/VS.

**data dictionary**. (1) A centralized repository of information about data, such as its meaning, relationship to other data, usage, and format. (2) A program to assist in effectively planning, controlling, and evaluating the collection, storage, and use of data. For example, DOS/VS DB/DC Data Dictionary.

**data element**. The smallest unit of data that can be referred to. Synonymous with *field*. See also *data aggregate.*

**data field**. Synonymous with *field.*

**data independence**. (1) The concept of separating the definitions of logical and physical data such that application programs do not depend on where or how physical units of data are stored. (2) The reduction of application program modification in data storage structure and access strategy.

**data management block (DMB)**. The data management block is created from a DBD by the application control blocks creation and maintenance utility, link edited, and cataloged in a core image library. The DMB describes all physical characteristics of a data base. Before an application program using DL/I facilities can be run, one DMB for each data base accessed, plus a PSB for the program itself, must be cataloged in a core image library. The DMBs and the associated PSB are automatically loaded into main storage from the core image library at the beginning of the application program execution (their loading is controlled by the parameter information supplied to DL/I at the beginning of program execution).

**data set**. A named organized collection of logically related records. They may be organized sequentially, as in the case of DOS/VSE SAM, or in key entry sequence, as in the case of VSE/VSAM. Synonymous with *file.*

**data set group (DSG)**. A control block linking together a data base with the data sets comprising this DL/I data base.

**DB**. Data base.

**DBA**. (1) Data base administration. (2) Data base administrator.

**DBD**. Data base description.

**DBDGEN**. Data base description generation -- the process by which a DBD is created.

**DB/DC**. Data base/data communication.

**DC**. Data communication.

**dependent segment**. A DL/I segment that relies on at least the root segment (or on another segment at a level immediately above its own) for its full hierarchical meaning. Synonymous with *child segment.*

**destination parent**. The physical or logical parent segment reached by the logical child path.

**device independence**. The concept of writing application programs such that they do not depend on the physical characteristics of the device on which data is stored.

**DIB**. DL/I interface block.

**direct access**. The retrieval or storage of a VSAM data record independent of the record's location relative to the previously retrieved or stored record. (See also *address direct access* and *keyed direct access*). Contrast with *sequential access.*

**distributed data**. The ability of DL/I application programs to access a data base that is resident on another processor.

**distributed free space**. See *free space.*

**DL/I interface block (DIB)**. Variables automatically defined in an application program using HLPI to receive information passed to the program by DL/I during execution. Contrast with *PCB mask.*

**DMB**. Data management block.

**DSG**. Data set group.

**DTF**. Define the file -- a control block that connects a program to a SAM, DAM, ISAM, or some other data set.

**dynamic backout**. A process that automatically cancels all activities performed by an application program that terminates abnormally.

**entity**. A item about which information is stored. It has properties that can be recorded. Information about an entity is a record.

**entry sequenced data set (ESDS)**. A VSAM data set whose records are physically in the same order as they were put in the data set. It is processed by addressed direct access or addressed sequential access and has no index. New records are added at the end of the data set.

**ESDS**. Entry sequenced data set.

**exclusive intent**. The scheduling intent type that prevents an application program from being scheduled concurrently with another application program. See *scheduling intent.*

**FDB**. field description block.

**field**. (1) (ISO)[3] In a record, a specified area used for a particular category of data, for example, in which a salary rate is recorded. (2) a unique or nonunique area (as defined during DBDGEN) within a segment that is the smallest unit of data that can be referred to. (3) any designated portion of a segment. (4) see also *key field.*

**field level sensitivity**. The ability of an application program to access data at the field level. See *sensitivity.*

**file**. (ISO)[3] A set of related records treated as a unit. See also *data set.*

**forward**. Movement in a direction from the beginning of the data base to the end of the data base, accessing each record in ascending root key sequence, and accessing the dependent segments of each root segment from top to bottom and from left to right. Referring to Figure X-1 on page X-1, forward accessing of all the segments shown would be in the following sequence: A-001, B-01, C-5, C-7, B-02, B-03, C-2, D, E, F, F, F, A-002.

**free space**. Space available in a VSAM data set for inserting new records. The space is distributed throughout a key sequenced data set (KSDS) or left at the end of an entry sequenced data set (ESDS). Synonymous with *distributed free space.*

**free space anchor point**. A fullword at the beginning of a control interval pointing to the first free space element in this CI.

**free space element**. In HD data bases, the portions of direct access storage not occupied by DL/I segments are called and marked as free space elements.

**FSA**. free space anchor point.

**FSE**. free space element.

**HD**. Hierarchical direct.

**HDAM**. Hierarchical direct access method.

**HIDAM**. Hierarchical indexed direct access method

**HIDAM index**. A data base that consists of logical DL/I records, each containing an image of the key field of a HIDAM root segment. A HIDAM index data base consists of one VSAM KSDS (keyed sequenced data set).

**hierarchic sequence**. The sequence of segment occurrences in a data base record defined by traversing the hierarchy from top to bottom, front to back, and left to right.

**hierarchical direct access method (HDAM)**. Provides for direct access to a DL/I data base in the HD organization. Segments are stored in VSAM control intervals and are referenced by a relative byte address. Root segments are accessed through a randomizing routine. An HDAM data base consists of one VSAM entry sequence data set (ESDS).

**hierarchical direct organization**. An organization of DL/I segments of a data base that are related by direct addresses and may be accessed through an HD randomizing routine or an index.

**hierarchical indexed direct access method (HIDAM)**. Provides for indexed access to a DL/I data base in the HD organization. Segments are stored in VSAM control intervals and are referenced by a relative byte address. Root segments are accessed through a HIDAM index data base. A HIDAM data base consists

[3] International Organization for Standardization, Technical Committee 97/Subcommittee 1.

of one VSAM Entry Sequenced Data Set (ESDS) and its associated index.

**hierarchical indexed sequential access method (HISAM)**. Provides for indexed access to a DL/I data base. A HISAM data base consists of one VSAM key sequenced data set (KSDS) and one VSAM entry sequenced data set (ESDS).

**hierarchical sequential access method (HSAM)**. The segments of a DL/I HSAM physical data base record are arranged in sequential order with the root segments followed by the dependent segments. HSAM data bases are accessed by the DOS/VSE sequential access method (SAM).

**hierarchical sequential organization**. An organization of DL/I segments of a data base that are related by physical adjacency.

**hierarchy**. (1) An arrangement of data segments beginning with the root segment and proceeding downward to dependent segments. (2) A “tree” structure.

**high level programming interface (HLPI)**. A DL/I facility providing services to application programs written in either COBOL or PL/I Optimizer language through commands.

**HISAM**. Hierarchical indexed sequential access method.

**HLPI**. High level programming interface.

**HS**. Hierarchical sequential.

**HSAM**. Hierarchical sequential access method.

**index data base**. An ordered collection of DL/I index entries (segments) consisting of a key and a pointer used by VSAM to sequence and locate the records of a key sequenced data set (KSDS). Organized as a balanced tree of levels of index.

**index data set**. Synonymous with *index data base.*

**index pointer segment**. The segment that contains the data and pointers used to index the index target segments.

**index record**. A system-created collection of VSAM index entries that are in collating sequence by the key in each of the entries.

**index segment**. The segment in the index data base that contains a pointer to the segment containing data (the indexed segment). Synonymous with *index pointer segment.*

**index set**. The set of VSAM index levels above the sequence set. An entry in a record in one of these levels contains the highest key entered in an index record in the next lower level and a pointer that indicates the record's physical location.

**index source segment**. The segment containing the data from which the indexing segment is built.

**index target segment**. The segment pointed to by a secondary index entry, that is, by an index pointer segment.

**indexed segment**. A segment that is located by an index. Synonymous with *index target segment.*

**intersection data**. Any user data in a logical child segment that does not include the logical parent's concatenated key.

**inverted file**. In information retrieval, a method of organizing a cross-index file in which a key identifies a record. The items pertinent to that key are indicated.

**key**. (1) (ISO)[4] One or more characters within a set of data that contains information about that set, including its identification. (2) The field in a segment used to store segment occurrences in sequential order. (3) A field used to search for a segment. See *primary key* and *secondary key.* (4) Synonymous with *key field* and *sequence field.*

**Note:** A segment may or may not have a key, that is, a sequence field. All root segments, except for HSAM and simple HSAM data bases, must have keys. DL/I ensures that multiple segments of the same type that have keys are maintained in strict ascending sequence by key. The key may be located anywhere within a segment; it must be in the same location in all segments of the same type within a data base. The maximum sizes for keys are 236 alphameric characters for root segments and 255 for all dependent segments. Keys provide a convenient way to retrieve a specific occurrence of a segment type, maintain the uniqueness and sequential integrity of multiples of the same segment type, and determine under which segment of a group of multiples new dependent segments are to be inserted. Keys should normally be prescribed for all segment types; the exceptions being if there will never be multiples of a particular type or if a particular segment type will never have dependents.

**key field**. The field is a segment used to store segment occurrences in sequential ascending order. A

[4] International Organization for Standardization, Technical Committee 97/Subcommittee 1.

key field is also a search field. Synonymous with *key* and *sequence field.*

**key sequenced data set (KSDS)**. A VSAM file whose records are loaded in key sequence and controlled by an index. See also *keyed direct access* and *keyed sequential access.*

**keyed direct access**. The retrieval or storage of a data record by use of an index that relates the record's key to its physical location in the VSAM data set, independent of the record's location relative to the previously retrieved or stored record. See also *addressed direct access, keyed sequential access,* and *addressed sequential access.*

**keyed sequential access**. The retrieval or storage of a VSAM data record in its collating sequence relative to the previously retrieved or stored record, by the use of an index that specifies the collating sequence of the records by key. See also *addressed sequential access, keyed direct access,* and *keyed sequential access.*

**KSDS**. Key sequenced data set.

**level**. (1) (ISO)[5] The degree of subordination of an item in a hierarchic arrangement. (2) Level is the depth in the hierarchical structure at which a segment is located. Roots are always the highest level and the segments at the bottom of the structure are the lowest level. The maximum number of levels in a DL/I data base is 15. For purposes of documentation and reference, the levels are numbered from 1 to 15, with the root segments being level number 1. Referring to Figure X-1 on page X-1:

- Three levels are shown.
- The A segments (roots) are at the highest level (Level 1).
- The C and E segments are at the lowest level (Level 3).

**local system**. (1) A specific system in a multisystem environment. Contrast with *remote system.* (2) The system in a multisystem environment on which the application program is executing. The local application may process data from data bases located on both the same (local) system and another (remote) system.

**local view**. A description of the data that an individual business process requires. See *system view.*

**logical**. When used in reference to DL/I components, logical means that the component is treated according to the rules of DL/I rather than physically as it may exist, or as it may be organized, on a physical storage device. For example, a logical DL/I record (a root segment and all of its dependent segments grouped) might be contained on several physical records or blocks on a storage device, and because of prior insertions and deletions, the segments might be in a different physical sequence than that by which they are retrieved logically for the application program by DL/I.

**logical child**. A pointer segment that establishes an access path between its physical parent and its logical parent. It is a physical child of its physical parent; it is a logical child of its logical parent. See also *logical parent* and *logical relationship.*

**logical data base**. A data base composed of one or more physical data bases representing a hierarchical structure derived from relationships between data segments that can be different from the physical structure.

**logical data base record**. (1) A set of hierarchically related segments of one or more segment types. As viewed by the application program, the logical data base record is always a hierarchic tree structure of segments. (2) All of the segments that exist hierarchically dependent on a given root segment, and that root segment.

**logical data structure**. A hierarchic structure of segments that is not based solely on the physical relationship of the segments. See also *logical relationships.*

**logical parent**. The segment a logical child points to. A logical parent segment can also be a physical parent. See also *logical child* and *logical relationship.*

**logical relationship**. A user defined path between two segments; that is, between logical parent and logical child, which is independent of any physical path. Logical relationships can be defined between segments in the same physical data base hierarchy or in different hierarchies.

**logical twins**. All occurrences of one type of logical child with a common logical parent. Contrast with *physical twin.* See also *twin segment.*

**MPS**. Multiple partition support

**MPS Restart facility**. The capability to restart an MPS batch job when a system or application program failure occurs using VSE checkpoint/restart in conjunction with the DL/I checkpoint facility.

**multiple partition support (MPS)**. Multiple partition support provides a centralized data base facility to permit multiple applications in different partitions to

---

[5] International Organization for Standardization, Technical Committee 97/Subcommittee 1.

access DL/I data bases concurrently. MPS follows normal DL/I online conventions in that two programs cannot both update the same segment type in a data base concurrently. (With program isolation, two programs can concurrently update the same segment type; however, they cannot concurrently update the same segment. See *program isolation.*) However, two or more programs can retrieve from a data base while another program updates it. If one program has exclusive use of a data base, no other program can update it or retrieve from it.

**multiple SSA**. A series of segment search arguments (SSAs) included in a DL/I call to identify a specific segment or path. See also *segment search argument.*

**object segment**. The segment at the lowest hierarchical level specified in a particular command. See also *path call.*

**online**. A operating environment in which DL/I is used with CICS/DOS/VS (or another data communication program) to permit end-users of application programs to access and store information in a data base through terminals.

**option**. A command keyword used to qualify the requested function.

**parent**. Synonymous with *parent segment.*

**parent segment**. (1) A segment that has one or more dependent segments. Contrast with *child.* (2) A parent is the opposite of a child, or dependent segment, in that dependent segments exist directly beneath it at lower levels. A parent may also itself be a child. Referring to Figure X-1 on page X-1:

- A-001 is the parent of all B, C, D, E, and F segments.
- D is a parent of E, yet a child of A.
- B-02 is not a parent.
- None of the level 3 segments are parents.

**parentage**. Establishment in a program of a particular parent as the beginning point for the use of the get next in parent (GNP) or get hold next in parent (GHNP) functions. Parentage can only be established by issuing successful GU, GHU, GN, or GHN calls, or GET UNIQUE or GET NEXT commands.

**PATH**. The chain of segments within a record that leads to the currently retrieved segment. The formal path contains only one segment occurrence from each level from the root down to the segment for which the path exists. The exact path for each retrieved segment is returned in the following fields of the PCB:

**Field 2** Segment hierarchy level indicator

**Field 6** Segment name feedback area

**Field 7** Length of key feedback area

**Field 9** Key feedback area, containing the concatenated keys in the path.

Referring to Figure X-1 on page X-1:

- The path to C-5 is A-001, B-01.
- The path to C-7 is the same as the path to C-5.
- There is no path to A-002 because it is a root segment.

**path call**. (1) The retrieval or insertion of multiple segments in a hierarchical path in a single call, by using the D command code and multiple SSAs. (2) The retrieval, replacement, or insertion of data for multiple segments in a hierarchical path in a single command, by using the FROM or INTO options specifying an I/O area for each parent segment desired. The object segment is always retrieved, replaced, or inserted.

**PCB**. Program communication block.

**PCB mask**. A skeleton data base PCB in the application program by which the program views a hierarchical structure and into which DL/I returns the results of the application's calls.

**physical child**. A segment type that is dependent on a segment type defined at the next higher level in the data base hierarchy. All segment types, except the root segment, are physical children because each is dependent on at least the root segment. See also *child segment.*

**physical data base**. An ordered set of physical data base records.

**physical data base record**. A physical set of hierarchically related segments of one or more segment types.

**physical data structure**. A hierarchy representing the arrangement of segment types in a physical data base.

**physical parent**. A segment that has a dependent segment type at the next lower level in the physical data base hierarchy. See also *parent.*

**physical segment**. The smallest unit of accessible data.

**physical twins**. All occurrences of a single physical child segment type that have the same (single occurrence) physical parent segment type. Contrast with *logical twins.* See also *twin segment.*

**PI**. Program isolation

**pointer**. A physical or symbolic identifier of a unique target.

**position pointer**. For most call functions, a position pointer exists before, during, and after the completion of the function. The pointer indicates the next segment in the data base that can be retrieved sequentially. It is normally set by the successful completion of the call function. Referring to Figure X-1 on page X-1:

- If A-001 has just been retrieved, it points to B-01.
- If a new segment C-6 has just been inserted, it points to C-7.
- If the D segment has been deleted (E will be deleted along
  with it), it points to the first F segment.
- If the last F segment has just been retrieved, it points to A-002.

During PSB generation, it is possible to specify either single or multiple positioning.

**primary key**. The data element, or combination of data elements, within a segment that uniquely identifies an occurrence of that segment. See *key* and *secondary key.*

**program communication block (PCB)**. Every data base accessed in an application program has a PCB associated with it. The PCB actually exists in DL/I and its fields are accessed by the application program by defining their names within the application program as follows:

**COBOL** The PCB names are defined in the linkage section.

**PL/I** The PCB names are defined under a pointer variable.

**Assembler** The PCB names are defined in a DSECT.

**RPG II** The PCB names are automatically generated by the translator, or may be defined by the user.

There are nine fields in a PCB:

1. Data base name
2. Segment hierarchy level indicator
3. DL/I results status code
4. DL/I processing options
5. Reserved for DL/I
6. Segment name feedback area
7. Length of key feedback area
8. Number of sensitive segments
9. Key feedback area.

**Program Isolation (PI)**. A facility that isolates all data base activity of an application program from all other application programs active in the system until that application program commits, by reaching a synchronization point, that the data it has modified or created is valid.

This concept makes it possible to dynamically backout the data base activities of an application program that terminates abnormally without affecting the integrity of the data bases controlled by DL/I. It does not affect the activity performed by other application programs processing concurrently in the system.

**program specification block (PSB)**. A PSB is generated for each application program that uses DL/I facilities. The PSB is associated with the application program for which it was generated and contains a PCB for each data base that is to be accessed by the program. Once it is generated, the PSB is cataloged in a core image library, and subsequently processed by a utility along with the associated DBDs to produce the updated PSB and DMBs; all of these are cataloged in a core image library for subsequent use by the application program during execution.

**PSB**. Program specification block

**PSBGEN**. PSB generation -- the process by which a program specification block is created.

**qualified call**. A DL/I call that contains at least one segment search argument (SSA). See also *segment search argument.*

**qualified segment selection**. The identification of a specific occurrence of a given segment type in a command, by using the WHERE option in the command for the desired segment. Contrast with *qualified SSA.*

**qualified SSA**. A qualified segment search argument contains both a segment name that identifies the specific segment type, and segment qualification that identifies the unique segment within the type for which the call function is to be performed. See also *segment search argument* and *multiple SSA.*

**RAP**. Root anchor point.

**RBA**. Relative byte address.

**read-only intent**. The scheduling intent type that allows a program to be scheduled with any number of other programs except those with exclusive intent. No updating occurs. See *scheduling intent.*

**record**. A data base record is made up of at least a unique root segment, and all of its dependent segments. See *data base record.*

**relative byte address (RBA)**. The displacement of a stored record or control interval from the beginning of the storage space allocated to the VSAM data set to which it belongs.

**remote system**. In a multisystem environment, the system containing the data base that is being used by an application program resident on another (local) system Contrast with *local system.*

**root anchor point (RAP)**. A DL/I pointer in an HDAM control interval that points to a root segment or a chain of root segments.

**root segment**. The highest level (level 1) segment in a record. A root segment must have a key unless the organization is HSAM or simple HSAM. The sequence of the root segments constitutes the fundamental sequence of the data base. There can be only one root segment per record. Dependent segments cannot exist without a parent root segment but a root segment can exist without any dependent segments.

**RQDLI COMMAND**. The instruction in the RPG II program used to request DL/I services.

**scheduling intent**. An application program attribute defined in the PSB that specifies how the program should be scheduled if multiple programs are contending for scheduling. See *exclusive intent, read-only intent,* and *update intent.*

**search field**. In a given DL/I call, a field that is referred to by one or more segment search arguments (SSAs).

**secondary index**. Secondary indexes can be used to establish alternate entries to physical or logical data bases for application programs. They can also be processed as data bases themselves. See also *secondary index data base.*

**secondary index data base**. An index used to establish accessibility to a physical or logical data base by a path different from the one provided by the data base definition. It contains index pointer segments.

**secondary key**. A data element, or combination of data elements, within a segment that identifies -- and is used to locate -- those occurrences of the segment that have a property named by the key. See *key* and *primary key.*

**segment**. A segment is a group of similar or related data that can be accessed by the application program with one I/O function call. There may be a number of segments of the same type within a record.

**segment name**. A segment name is assigned to each segment type. Segment names for the different segment types must be unique within a data base. The segment name is used by the application programmer when constructing a qualified or unqualified SSA prior to issuing a call for a specific segment. Synonymous with *segment type.*

**segment occurrence**. One instance of a set of similar segments.

**segment search argument (SSA)**. Describes the segment type, or specific segment within a segment type, that is to be operated on by a DL/I call. See also *multiple SSA, qualified SSA,* and *unqualified SSA.*

**segment selection**. The specifying of parent and object segments by name in a command. Selection may be either qualified or unqualified. Contrast with *segment search argument.*

**segment type**. A user-defined category of data. Referring to Figure X-1 on page X-1, there are six different types of segments; A through F.

Different segment types may have different lengths, but within each single type, all segments must be the same length (unless variable length segments have been specified by the DBA). Synonymous with *segment name.*

**sensitivity**. (1) A DL/I capability that ensures that only data segments or fields predefined as “sensitive” are available for use by a particular application program. The sensitivity concept also provides a degree of control over data security, inasmuch as users can be prevented from accessing particular segments or fields from a logical data base. (2) Sensitivity to the various segments and fields that constitute a data base is controlled, on a program-by-program basis, when the PSB for each program is generated. For example, a program is said to be sensitive to a segment type when it can access that segment type. When a program is not sensitive to a particular segment type, it appears to the program as if that segment type does not exist at all in the data base. Segment sensitivity applies to types of segments, not to specific segments within a type, and to all segment types in the path to the lowest level sensitive segment type.

**sequence field**. Synonymous with *Key field*.

**sequence set**. The lowest level of a VSAM index. It immediately controls the order of records in a key sequenced data set (KSDS). A sequence set entry contains the key of the highest keyed record stored in a control interval of the data set and a pointer to the control interval's physical location. A sequence set record also contains a pointer to the physical location of each free control interval in the fan-out of the record.

**sequential processing**. Processing or searching through the segments in a data base in a forward direction (see also *forward*).

**simple HISAM**. A hierarchical indexed sequential access method data base containing only one segment type.

**source segment**. A segment containing the data used to construct the secondary index pointer segment. See also *secondary index data base.*

**SSA**. Segment Search Argument

**status code**. Each DL/I request for service returns a status code that reflects the exact results of the operation. The first operation that a program should perform immediately following a DL/I request is to test the status code to ensure that the function requested was successful. Following a command, the status code is returned in the DIB at the label DIBSTAT. Following a call, the status code is returned in field 3 of the PCB.

**sync(h) point**. Synonymous with *synchronization point.*

**synchronization point**. A logical point in time during the execution of an application program where the changes made to the data bases by the program are committed and will not be backed out. Synonymous with *sync point* or synch point.

A synchronization point is created by:

- a DL/I CHECKPOINT command or CHKP call
- a DL/I TERMINATE command or TERM call
- a CICS/VS synch point request
- an end of task (online) or an end of program (MPS-batch).

**system view**. A conceptual data structure that integrates the individual data structures associated with local views into an optimum arrangement for physical implementation as a data base. See *local view.*

**transaction**. A specific set of input data that triggers the execution of a specific process or job.

**twin segments**. All child segments of the same segment type that have a particular instance of the same parent type. See also *physical twins* and *logical twins.*

**twins**. Synonymous with *twin segments.*

**unqualified call**. A DL/I call that does not contain a segment search argument.

**unqualified segment selection**. The identification of a given segment type in a command without specifying a particular occurrence of that segment type (without using the WHERE option). As a general rule, unqualified segment selection retrieves the first occurrence of the specified segment type. Contrast with *unqualified SSA.*

**unqualified SSA**. An unqualified SSA contains only a segment name that identifies the specific type of segment for which the I/O function is to be performed. As a general rule, the use of an unqualified SSA retrieves the first occurrence of the specified type of segment. See also *segment search argument.*

**update intent**. The scheduling intent type that permits application programs to be scheduled with any number of other programs except those with exclusive intent. See *scheduling intent.*

**UPSI**. User program switch indicator. A special 8 bit byte that allows each bit to be set by the user as “1” or “0.” Bits may be read by a program to determine what the user wants to do.

# Communicating Your Comments to IBM

IBM Data Language/I Disk Operating System/
Virtual Storage (DL/I DOS/VS)
Application Programming:
High Level Programming Interface
Version 1 Release 7

Publication No. SH24-5009-02

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of the book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF form and either send it postage-paid in the United States, or directly to:

  IBM Deutschland Entwicklung GmbH
  Department 3248
  Schoenaicher Strasse 220
  D-71032 Boeblingen
  Federal Republic of Germany

- If you prefer to send comments by FAX, use this number:
  - (Germany): 07031-16-3456
  - (Other countries): (+49)+7031-16-3456

- If you prefer to send comments electronically, use this network ID:

  INTERNET: s390id@de.ibm.com

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies.

# Readers' Comments — We'd Like to Hear from You

**IBM Data Language/I Disk Operating System/
Virtual Storage (DL/I DOS/VS)
Application Programming:
High Level Programming Interface
Version 1 Release 7**

**Publication No. SH24-5009-02**

**Overall, how satisfied are you with the information in this book?**

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|---|---|---|---|---|---|
| Overall satisfaction | □ | □ | □ | □ | □ |

**How satisfied are you that the information in this book is:**

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|---|---|---|---|---|---|
| Accurate | □ | □ | □ | □ | □ |
| Complete | □ | □ | □ | □ | □ |
| Easy to find | □ | □ | □ | □ | □ |
| Easy to understand | □ | □ | □ | □ | □ |
| Well organized | □ | □ | □ | □ | □ |
| Applicable to your tasks | □ | □ | □ | □ | □ |

**Please tell us how we can improve this book:**

Thank you for your responses. May we contact you? □ Yes □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

Company or Organization _____

Phone No. _____

**Readers' Comments — We'd Like to Hear from You**
SH24-5009-02

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

Fold and Tape | **Please do not staple** | Fold and Tape



Cut or Fold
Along Line

File Number: S370/4300-50
Program Number: 5746-XX1

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SH24-5009-02