SH20-9026-8
File No. S370-50

Program Product

IMS/VS Version 1 Application Programming: Designing and Coding

Program Number 5740-XX2

Release 2

SH20-9026-8
File No. S370-50

Program Product

IMS/VS Version 1 Application Programming: Designing and Coding

Program Number 5740-XX2

Release 2

IBM

This publication was produced using the
IBM Document Composition Facility
(program number 5748-XX9) and
the master was printed on the IBM 3800 Printing Subsystem.

Ninth Edition (March 1981)

This is a major revision of, and makes obsolete, SH20-9026-7.

This edition applies to Version 1, Release 2 of IMS/VS, Program Product 5740-XX2, and to any subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Amendments" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370 and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

PREFACE

This retitled edition supersedes the seventh edition, called the IMS/VS Version 1 Application Programming Reference Manual. This manual contains the application programming information from the previous edition, and additional information on designing IMS/VS application programs. The application programming information from the previous edition has been separated into a guidance part and a reference part to better support application programming.

PREREQUISITE KNOWLEDGE

Before using this manual, you should understand basic IMS/VS concepts, the IMS/VS environment, and your installation's IMS/VS system. The IMS/VS concepts explained in this manual are limited to those concepts pertinent to developing and coding IMS/VS application programs. You should also know how to use COBOL, PL/I, or assembler language.

HOW THIS BOOK IS ORGANIZED

The manual is divided into three parts. The first and second parts, "Application Design Guide" and "Application Programming Guide," guide you in designing and coding IMS/VS application programs. The third part, "For Your Reference," contains reference information about the parts of an IMS/VS application program. This part is for experienced programmers who understand IMS/VS application programming and need only to look up facts like a call format, a command code, or the meaning of a particular status code.

Part 1. Application Design Guide

This part contains the following five chapters:

- Chapter 1, "Concepts and Terminology," describes the characteristics of a hierarchic data base and explains the relationship between the programs you are designing and coding and the DL/I data base. This chapter also introduces you to the tasks you will perform in developing online and batch application programs.
- Chapter 2, "Analyzing Application Requirements," gives an overview of application design and explains two of the subtasks of application design: defining application data and designing local views.
- Chapter 3, "Understanding Online and Batch Processing," describes the types of processing available in IMS/VS in terms of the application requirements that each type answers. This chapter will help you understand characteristics of each type of processing so you can decide which type will satisfy the requirements of the application.
- Chapter 4, "Gathering Requirements for Data Base Options," describes data base options in terms of the application requirements they satisfy and how each option affects the performance and efficiency of the program. This chapter describes the information about your application that you should supply to the data base administrator. This information will be helpful to the DBA in choosing a data base design that will meet the requirements of all applications.
- Chapter 5, "Gathering Requirements for Data Communications Options," describes data communications options in terms of the application requirements each satisfies. This information will help you understand these options and will enable you to

gather information that will be useful to the DB/DC system administrator.

Part 2. Application Programming Guide

This part contains the following seven chapters:

- Chapter 6, "Structuring the DL/I Portion of a Program," contains guidance information to help you structure programs using DL/I calls, SSAs, command codes, status codes, and other tools and techniques. This chapter and the next contain all the information you need to structure and code batch application programs. Chapters 8, 9, and 10 contain additional information on structuring and coding MPPs, BMPs, and Fast Path programs.
- Chapter 7, "Coding the DL/I Portion of a Program," will guide you in coding batch programs and the DL/I portions of online programs, according to the design decisions you have made.
- Chapter 8, "Structuring a Message Processing Program," explains what your program must do to process messages. It tells how to structure an MPP using the tools and techniques described in Chapter 6 and those introduced in this chapter.
- Chapter 9, "Coding a Message Processing Program," explains how you code an MPP once you have a detailed design of it.
- Chapter 10, "Structuring and Coding a Batch Message Program," describes the types of BMPs that you can write, and tells what tools they can use.
- Chapter 11, "Testing an Application Program," gives you guidance for testing DL/I call sequences, explains what you need to test a program, and describes the tools available to help you test. This chapter also describes the actions you can take in isolating a problem when your program does not execute correctly.
- Chapter 12, "Documenting an Application Program," explains why you should document your programs and suggests parts of the application development process to document.

Part 3. For Your Reference

This part contains reference information about the parts of an IMS/VS application program such as entry statements, DL/I calls, system service calls, SSAs, command codes, PCBs, and status codes, as they apply to COBOL, PL/I, and assembler language.

Appendixes

The appendixes contain the following:

- Appendix A is a sample batch application program. This program is written in COBOL and accesses the sample data base that is part of the IMS/VS Primer function.
- Appendix B is a sample batch message (BMP) program. This program is written in COBOL and accesses the sample data base that is part of the IMS/VS Primer function.
- Appendix C is a sample message processing program (MPP). This program is written in PL/I and accesses the sample data base that is part of the IMS/VS Primer function.
- Appendix D is a sample conversational MPP program. This program is written in COBOL and accesses the sample data base that is part of the IMS/VS Primer function.
- Appendix E is a sample status code error routine. This program is written in assembler language and accesses the sample data base that is part of the IMS/VS Primer function.

- Appendix F contains reference information on the formats and usage of the DL/I Test Program control statements.

PREREQUISITE PUBLICATION

- The IMS/VS General Information Manual, GH20-1260 introduces IMS/VS. You can use this manual to acquaint yourself with IMS/VS functions, the hardware and software products prerequisite to using IMS/VS, and the IMS/VS facilities that help satisfy application requirements.

RELATED PUBLICATIONS

- The IMS/VS Data Base Administration Guide, SH20-9025, contains guidance information on planning, designing, implementing, monitoring and tuning, and controlling a data base in an IMS/VS system. It gives the characteristics of the various kinds of IMS/VS data bases and the design considerations of each.
- The IMS/VS System Administration Guide, SH20-9178, contains guidance information on establishing a data base/data communication (DB/DC) system. This book explains how to control the content of the IMS/VS system definition and establish operating procedures. This manual also has information on monitoring the performance of the IMS/VS system, coding IMS/VS execution JCL, and preparing IMS/VS system definition macros for application and tuning requirements.
- IMS/VS Utilities Reference Manual, SH20-9029, describes the function and use of IMS/VS utilities. It contains reference information necessary for those installing an application system and planning operational procedures.
- IMS/VS Installation Guide, SH20-9081, contains IMS/VS reference information for all aspects of IMS/VS installation and system definition.
- IMS/VS System Programming Reference Manual, SH20-9027, contains reference information useful when tuning the IMS/VS system or for coding exit routines.
- IMS/VS Message Format Service User's Guide, SH20-9053, contains both design and reference information for formatting messages to and from terminals.
- IMS/VS Fast Path General Information Manual,GH20-9069, describes the IMS/VS Fast Path feature and provides information for evaluating the use of this feature.
- IMS/VS Primer, SH20-9145, describes the subset of IMS/VS functions available under the IMS/VS Primer function. The Primer function enables a first-time IMS/VS user to design and install an application within a simpler system.
- DB/DC Data Dictionary General Information Manual, GH20-9104, contains introductory information on the DB/DC Data Dictionary and the hardware and software products prerequisite to using it.
- DB/DC Data Dictionary Applications Guide, SH20-9173, explains how you define subjects to the Data Dictionary.
- BTS II Batch Terminal Simulator II Program Description/Operations Manual, SH20-1844, tells you how to use BTS II to test application programs.

For installations with IMS/VS DB and CICS/OS/VS:

- CICS/VS General Information Manual, GC33-0066, contains a general introduction to CICS/VS, sample applications, and machine and program requirements.
- CICS/VS Application Programmer's Reference Manual, SC33-0079, contains information about requesting Data Language I (DL/I) services from a CICS/VS application program.

SUMMARY OF AMENDMENTS

VERSION 1, RELEASE 2, MARCH 1981

NEW PROGRAMMING FACILITIES

- Data sharing provides control for application programs in two or more IMS/VS systems to access IMS/VS data bases concurrently. The IMS/VS systems can be in one processor, or they can be in separate processors. Application programs that share data bases with application programs in other IMS/VS systems should issue checkpoint calls frequently; if they don't, they can tie up portions of the data base, preventing other application programs from accessing some data.
- There are two new processing options that can be used with the GO processing option for application programs: T and N. If an application program with the processing option GO tries to retrieve a segment containing an invalid pointer, IMS/VS terminates the program abnormally. If the application program uses the T or N option with GO (GOT or GON), IMS/VS returns control to the application program and returns a GG status code. In addition, the T processing option causes IMS/VS to retry the call before returning control to the program.

VERSION 1, RELEASE 1.6, JULY 1980

NEW PROGRAMMING FACILITIES

- Intersystem Communication, or ISC, is a part of Multiple Systems Coupling (MSC) that makes communication sessions between IMS/VS and other subsystems (such as CICS/VS, a user-written system, or another IMS/VS system) possible.
- MSC directed routing makes it possible for an application program to specify the multiple system name (MSNAME) and destination within that system for a message in another system. The application program can receive the MSNAME of the system that originally scheduled it.
- Application programs may bypass MFS editing or basic editing when communicating via 3270 or SLU 2 devices. This bypass makes it possible to leave the screen in an unprotected mode so that IMS/VS can send output to the device without requiring input from the device. The application program can also control the locking and unlocking of the keyboard.

MAJOR EDITORIAL CHANGES

This book is a complete reorganization of the IMS/VS Version 1 Application Programming Reference Manual. The revised book contains information on application design that is, for the most part, new to IMS/VS publications, and it contains information on application programming from the APRM.

Part 1 is a guide to designing IMS/VS applications. This new information has been added to help you in designing IMS/VS applications.

Parts 2 and 3 contain information on application programming from the APRM. This information has been separated into guide and

reference information. Part 2 is a guide to application programming; it explains the steps of structuring and coding IMS/VS application programs. Part 2 is for those who are not thoroughly familiar with IMS/VS, particularly people who have little or no experience with IMS/VS. Part 3 contains reference information concerning application programming. This information is intended for experienced IMS/VS application programmers who understand the concepts of IMS/VS application programming, and need only to look up a specific piece of information.

For a chapter by chapter description of the revised book, see the Preface at the beginning of this book.

VERSION 1, RELEASE 1.5, DECEMBER 1978

FIELD LEVEL SENSITIVITY

Changes have been made to reflect the use of field level sensitivity.

VERSION 1, RELEASE 1.5, SEPTEMBER 1978

NEW PROGRAMMING FACILITIES

- An application program can issue IMS/VS operator commands, and a user-written exit routine can monitor resource activity, by using the Automated Operator Interface (AOI).
- Expanded security facilities, including the presence of the user identification in the I/O PCB.
- Access to Fast Path and IMS/VS data bases from both Fast Path and IMS/VS application programs and other enhancements for integrated support.
- Support for direct dependent segment types for DEDBs.
- An IMS/VS Primer function which makes it easier for new IMS/VS users to get IMS/VS and their initial applications up running. This function runs under OS/VS1 and OS/VS2 MVS.

OTHER CHANGES

- The rollback (ROLB) call can be issued by IMS/VS application programs to undo data base changes without subsequent abnormal termination.

Service Changes

- Further system service call clarification.
- Correction to PL/I conversational program example.

RELEASE 1.4, FEBRUARY 1978

TECHNICAL CHANGES

- The DL/I call trace facility has been added to the book. This facility traces and records all DL/I calls issued by an application program, making it possible to duplicate the conditions that caused a program failure. The trace output can be used as input to the DL/I test prgram, DFSDDLT0.

OTHER CHANGES

Service Changes

- COBOL and PL/I examples have been updated.
- The explanations of the DL/I system service checkpoint, restart, log, and statistics calls have been clarified.
- The example of the independent AND Boolean operator has been expanded.
- A single use of the MFS MOD parameter is allowed for data communication insert and purge calls. Clarification in the use of the data communication change call has been added.
- DL/I test program changes:
  - The DATA statement maximum segment size has been corrected.
  - The operation of the SNAP call has been clarified.
- DL/I status code changes:
  - The detailed descriptions of status codes AM, DJ, and IX were expanded.
  - A description for status code XX was added.
- The majority of the Boolean operator information given in Chapter 2 has been moved to the end of Chapter 3.

VERSION 1, RELEASE 1.4

NEW PROGRAMMING FEATURE

The Fast Path feature provides data base and data communication facilities for applications requiring high transaction rates but needing only simple data base structures. The Fast Path feature uses functions of the Data Communication feature and operates with existing telecommunication networks.

Fast Path provides two new types of data bases that are accessed with standard DL/I calls and, optionally, with Fast Path DL/I calls. The feature includes a message-handling facility to expedite the processing of Fast Path messages.

Four new DL/I calls exist for use in Fast Path applications: the field (FLD), rollback (ROLB), synchronize (SYNC), and position (POS) calls. These calls and all status codes associated with them have been added to the manual. Also added are notes on the use of those IMS/VS DL/I calls applicable in the Fast Path environments.

OTHER CHANGES

- Fast Path application programming is addressed in a new Chapter 6.
- A section on determining data base position after a GE status code has been added to the manual.

VERSION 1, RELEASE 1.2

TECHNICAL CHANGES

This release reflects technical changes to this publication in support of the following new feature and devices:

- Multiple Systems Coupling Feature
- 3767 Communication Terminal
- 3779 Data Coummunication System

OTHER CHANGES

- A symbolic all interface for the extended checkpoint/restart facility has been added. With this facility, ANS COBOL and PL/I application programs can now issue extended CHKP and XRST DL/I calls and also CHKP DL/I calls that specify OS checkpoints.
- Updates have been made to PL/I information, and a revised example is included for the PI/I Optimizing Compiler.
- Chapter 7 of this edition describes the "DL/I Test Program" and "Message Processing Region Simulation." This information was formerly in Appendix C of the IMS/VS Utilities Reference Manual and Appendix B of the IMS/VS System/Application Design Guide, respectively.

VERSION 1, MODIFICATION LEVEL 1.1.

- Support has been added for the 3740 Data Entry System. IMS/VS supports the 3741 Data Station, Model 2, and the 3741 Programmable Work Station, Model 4, attached on a switched line using BTAM.
- The restriction against the Utility Control Facility (UCF) has been lifted.

VERSION 1, MODIFICATION LEVEL 1

The following new and/or enhanced IMS/VS functions have been added:

- Generalized Sequential Access Method (GSAM)
- Expanded restart (restart call), get System Contents Directory Call, and statistics call
- Response alternate PCBs
- Fixed-length SPAs
- Program isolation
- Application program output limits
- Message Format Service (MFS) support for additional terminals

Note: Information in this manual about the Utility Control Facility (UCF) is for planning purposes only until that facility becomes available.

VERSION 1, MODIFICATION LEVEL 0.1

- Support for the IBM 2260 Display Station, Model 1 and 1, and for the IBM 2265 Display Station, Model 1

CONTENTS

FIGURES

PART 1. APPLICATION DESIGN GUIDE

This part of the book gives you an introduction to IMS/VS, and covers the decisions that you have to make when you are designing an IMS/VS application. The introductory material and the tasks that this part covers are:

- Concepts and Terminology
- Analyzing Application Requirements
- Understanding Online and Batch Processing
- Gathering Requirements for Data Base Options
- Gathering Requirements for Data Communications Options

CHAPTER 1. CONCEPTS AND TERMINOLOGY

This chapter is an introduction to IMS/VS, and to designing and coding IMS/VS application programs. The first section explains some basic concepts about processing a data base, and the second section gives an overview of the tasks covered in this book:

- **Processing Information in a Data Base**

  This section explains the concepts and terms that you need to understand before reading the chapters that follow.

- **A Look at the Tasks Ahead of You**

  This section describes what you do to design and code IMS/VS application programs.

PROCESSING INFORMATION IN A DATA BASE

Before explaining what data base records look like and how you process them, this section describes what makes storing data in a data base different from other ways of storing data.

COMPARING WAYS TO STORE DATA

The advantage of storing and processing data in a data base is that all of the data appears only once, and that each program has to process only the data that it needs. One way to understand this is to compare three ways of storing data: in separate files, in a combined file, and in a data base.

Storing Data in Separate Files

If you keep separate files of data for each part of your organization, you can make sure that each program uses only the data it needs, but you have to store a lot of the data in several places at once. The problem with this is that redundant data takes up space that could be used for something else.

For example, suppose that a medical clinic keeps separate files for each of its departments, such as the clinic department, the accounting department, and the ophthalmology department.

- The clinic department keeps data about each patient that visits the clinic. For each patient, the clinic department needs to keep this information:
  - The patient's identification number
  - The patient's name
  - The patient's address
  - The patient's illnesses
  - The date of each illness
  - The date that the patient came to the clinic for treatment
  - The treatment that was given for each illness
  - The doctor that prescribed the treatment
  - The charge for the treatment

- The accounting department also keeps information about each patient. The information that the accounting department might keep for each patient is:
  - The patient's identification number
  - The patient's name
  - The patient's address
  - The charge for the treatment
  - The amount of the patient's payments

  The information that the ophthalmology department might keep for each of its patients is:
  - The patient's identification number
  - The patient's name
  - The patient's address
  - The patient's illnesses that relate to ophthalmology
  - The date of each illness
  - The names of the members in the patient's household
  - The relationship between the patient and each household member

If each of these departments keeps separate files, each department uses only the data that it needs, but a lot of data is redundant. For example, every department in the clinic uses at least the patient's number, name, and address. Updating the data is also a problem because if several departments change the same piece of data, you have to update the data in several places. Because of this, it's difficult to keep the data in each department's files current. There's a danger of having current data in one department, and "old" data in another.

Storing Data in a Combined File

Another way to store data is to combine all of the files into one file for all of the departments at the clinic to use. In the medical example, the patient record that would be used by each department would contain these fields:

- The patient's identification number
- The patient's name
- The patient's address
- The patient's illnesses
- The date of each illness
- The date that the patient came to the clinic for treatment
- The treatment that was given for each illness
- The doctor that prescribed the treatment
- The charge for the treatment
- The amount of the patient's payments
- The names of the members in the patient's household

- The relationship between the patient and each household member

Using a combined file solves the updating problem because all of the data is in one place, but it creates a new problem: the programs that process this data have to access the entire data base record to get to the part that they need. For example, to process only the patient's number, charges, and payments, an accounting program has to access all of the other fields as well. In addition, changing the format of any of the fields within the patient's record affects all of the application programs, not just the programs that use that field. Using combined files can also involve security risks, since all of the programs have access to all of the fields in a record.

Storing Data in a Data Base

Storing data in a data base gives you the advantages of separate files and combined files: all of the data appears only once, and each program accesses only the data that it needs. This means that:

- When you update a field, you only have to update it in one place.
- Since you store each piece of information only in one place, you can't have an updated version of the information in one place and an out-of-date version of the information in another place.
- Each program accesses only the data it needs.
- You can keep programs from accessing private information.

In addition, storing data in a data base has two advantages that neither of the other ways has:

- If you change the format of part of a data base record, the change doesn't affect the programs that don't use the changed information.
- Programs aren't affected by how the data is stored.

Because the program is independent of the physical data, a data base can store all of the data only once and yet make it possible for each program to use only the data that it needs. In a data base, what the data looks like when it's stored, and what it looks like to an application program are two different things.

WHAT THE DATA LOOKS LIKE WHEN IT'S STORED

In IMS/VS, a record is stored and accessed in a hierarchy. A hierarchy shows how each piece of data in a record relates to other pieces of data in the record. Figure 1 shows the hierarchy you could use to store the patient information described earlier in this chapter.

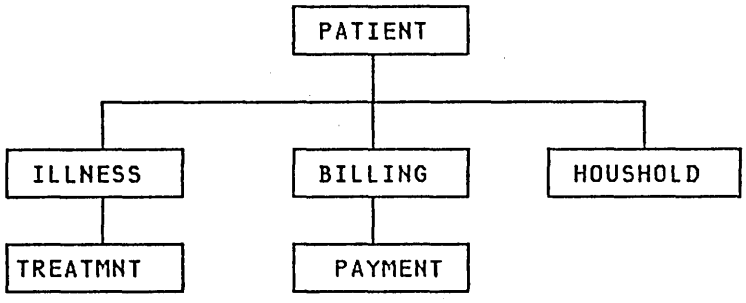

Figure 1. Medical Data Base Hierarchy

IMS/VS connects the pieces of information in a data base record by defining the relationships between the pieces of information that relate to the same subject. The result of this is a data base hierarchy. The hierarchy shows how each piece of information is related to other pieces of information in the record. The relationship between two pieces of information in the hierarchy means that one piece of information is either dependent on or equal to another piece of information.

In the medical data base, the data that you're keeping is information about a particular patient. Information that is not associated with a particular patient is meaningless. For example, keeping information about a treatment given for a particular illness is meaningless if the illness isn't associated with a patient. ILLNESS, TREATMNT, BILLING, PAYMENT, and HOUSHOLD must always be associated with one of the clinic's patients to be meaningful information.

There are five kinds of information you're keeping about each patient. The information about the patient's illnesses, billings, and household depends directly on the patient. The information about the patient's treatments and the patient's payments depends respectively on the patient's illnesses and the patients payments as well.

Each of the pieces of data represented in Figure 1 is called a segment in the hierarchy. A segment is the smallest unit of data that an application program can retrieve from the data base. Each segment contains one or more fields of information. The PATIENT segment, for example, contains all of the information that relates strictly to the patient: the patient's identification number, the patient's name, and the patient's address.

WHAT THE DATA LOOKS LIKE TO YOUR PROGRAM

IMS/VS uses two kinds of control blocks to make it possible for application programs to be independent of the way in which you store the data in the data base. One control block defines the physical structure of the data base; another defines an application program's view of the data base:

- A data base description, or DBD, is a control block that describes the physical structure of the data base. The DBD also defines the appearance and contents, or fields, that make up each of the segment types in the data base.

  For example, the DBD for the medical data base hierarchy shown in Figure 1 would describe to IMS/VS the physical structure of the hierarchy, and it would describe each of the six segment types in the hierarchy: PATIENT, ILLNESS, TREATMNT, BILLING, PAYMENT, and HOUSHOLD.

- A data base program communication block, or DB PCB, in turn, defines an application program's view of the data base. An application program often needs to process only some of the segments in a data base. A PCB defines which of the segments in the data base the program is allowed to access. The program is "sensitive" to the segments that it's allowed to access. The data structures that are available to the program contain only segments that the program it sensitive to.

For example, an accounting program that calculates and prints bills for the clinic's patients would need only the PATIENT, BILLING, and PAYMENT segments. You could define the data structure shown in Figure 2 in a DB PCB for this program.

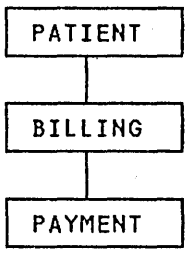

Figure 2. Accounting Program's View of the Data Base

A program that updates the data base with information on patients' illnesses and treatments, on the other hand, would need to process the PATIENT, ILLNESS, and TREATMNT segments. You could define the data structure shown in Figure 3 for this program.

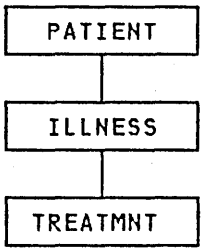

Figure 3. Patient Illness Program's View of the Data Base

Sometimes a program needs to process all of the segments in the data base. When this is true, the program's view of the data base as defined in the DB PCB is the same as the DL/I hierarchy that's defined in the DBD.

Each DB PCB defines a way in which the application program views and processes the data base. The DB PCB also tells IMS/VS how the program is allowed to process the segments in the data structure—whether the program can only read them, or whether it can update segments as well.

A program specification block, or PSB, contains the DB PCBs for a particular application program. A program may use only one DB PCB—which means it processes only one data structure—or it may use several DB PCBs, one for each data structure. There is one PSB for each application program.

Since an application program processes only the segments in a data base that it requires, if you change the format of a segment that a program doesn't process, you don't have to change the program. A program is affected only by the segments that it accesses. In addition to being sensitive to only certain segments in a data

base, a program can also be sensitive to only certain fields within a segment. This is called field level sensitivity. If you change a segment that the program isn't sensitive to, it doesn't affect the program. In the same way, if you change a field that the program isn't sensitive to, it doesn't affect the program.

HOW YOU PROCESS A DATA BASE RECORD

A data base record is a root segment occurrence and all of its dependents. In the medical example, a data base record is all of the information about one patient. The PATIENT segment in the medical data base is called the root segment. The segments below the root segment are called dependents, or children, of the root. For example, ILLNESS, BILLING, and HOUSHOLD are all children of PATIENT. ILLNESS, BILLING, and HOUSHOLD are called direct dependents of PATIENT; TREATMNT and PAYMENT are also dependents of PATIENT, but they are not direct because they are at a lower level in the hierarchy.

Each data base record has only one root segment occurrence, but it may have several occurrences at lower levels. For example, the data base record for a patient contains only one occurrence of the PATIENT segment type, but it may contain several ILLNESS and TREATMNT segment occurrences for that patient.

To process the information in the data base, your application program communicates with IMS/VS in three ways:

- Passing control: IMS/VS passes control to your application program through an entry statement in your program. Your program returns control to IMS/VS when it has finished its processing.
- Communicating processing requests: Your program communicates processing requests to IMS/VS by issuing calls to Data Language I, or DL/I. DL/I is an access method that handles the data in the data base.
- Exchanging information with DL/I: Your program exchanges information with DL/I through two areas in your program. First, DL/I reports the results of your calls in the DB PCB. Your program builds a mask of the DB PCB and uses this mask to check the results of the calls. Second, when you request a segment from the data base, DL/I returns the segment to your I/O area. When you want to update a segment in the data base, you place the new value of the segment in the I/O area.

An application program can read and update a data base. When you update a data base, you can replace segments, delete segments, or add segments. You indicate to DL/I the segment you want to process, and whether you want to read or update it, in a DL/I call.

A LOOK AT THE TASKS AHEAD OF YOU

There are five kinds of tasks in developing an IMS/VS application and the programs that are part of the application:

- Designing the application. Application design varies from installation to installation, and from one application to another. Because of this, this book does not try to cover the early tasks that are part of designing an application. Instead, it covers only the tasks that you are concerned with once the early specifications for the application have been developed. These subtasks are:
  - Analyzing Application Requirements. Two important parts of application design are defining the data that each of the business processes in the application will require, and designing a local view for each of the business

processes. "Chapter 2. Analyzing Application Requirements" explains these tasks.

- Understanding Online and Batch Processing. When you understand the business processes that are part of the application, you can analyze the requirements of each business process in terms of the processing available with different types of IMS/VS application programs. "Chapter 3. Understanding Online and Batch Processing" explains IMS/VS processing and the application requirements that each satisfies.

- Gathering Requirements for Data Base Options. You then need to look at the data base options that can efficiently answer the requirements and gather information about your application's data requirements that relates to each of the options. "Chapter 4. Gathering Requirements for Data Base Options" explains these options, and it helps you to gather information about your application that will be helpful to the data base administrator in choosing these options.

- Gathering Requirements for Data Communications Options. If your application will use data communications, you also need to look at its data communications requirements and gather information that relates to DC options. "Chapter 5. Gathering Requirements for Data Communications Options" explains the IMS/VS data communications options, and helps you to gather information about your application that will be helpful in choosing them.

- Developing specifications. This task depends completely on the application being developed and the installation.

- Implementing the design. Once the specifications for each of the programs in the application have been developed, you can structure and code the program according to those specifications:

  - Structuring the DL/I Portion of a Program. Once the program design is complete, you can structure the DL/I calls and data areas based on the programming specifications that have been developed. "Chapter 6. Structuring the DL/I Portion of a Program" tells you how to do this.

  - Coding the DL/I Portion of a Program. Once you know the structure of the program, you implement that structure by coding DL/I calls and the data areas used to communicate with IMS/VS. "Chapter 7. Coding the DL/I Portion of a Program" is a guide to coding a batch program.

  - Structuring a Message Processing Program. If you're writing a program that will communicate with terminals and other programs, you need to structure the message processing part of the program. "Chapter 8. Structuring a Message Processing Program" tells how you do this.

  - Coding a Message Processing Program. Again, once you have developed the structure of the program, you implement that structure by coding the DC calls that enable your program to send and receive messages. "Chapter 9. Coding a Message Processing Program" tells how you code the parts of an MPP.

  - Structuring and Coding a Batch Message Program. If your program performs batch processing online, there are some additional considerations in structuring and coding your program. "Chapter 10. Structuring and Coding a Batch Message Program" explains these considerations.

- Testing an Application Program. When you have completed coding your program, you test it by itself and then as part of a system. "Chapter 11. Testing an Application Program" gives some guidelines on program test.
- Documenting an Application Program. Documenting a program should not be something that you do at the end of a project. It is most effective when done incrementally; when the program is completely tested, however, there is some additional information you need to supply for people who use and maintain your program. "Chapter 12. Documenting an Application Program" gives some suggestions about the information you should record about your program.

CHAPTER 2. ANALYZING APPLICATION REQUIREMENTS

Designing an application that meets the requirements of end users involves a variety of tasks, and, usually, people from several departments at an installation. Application design begins when a department or business area communicates a need for some type of processing; it ends when each of the parts of the application system—for example, the programs, the data bases, the display screens, and the message formats—have been designed. This chapter gives an overview of application design and explains two of the tasks involved in the process:

- **An Overview of Application Design**

  This section gives a general description of application design so that you'll understand how the tasks explained later in this chapter fit into the application design process. It also gives some suggestions about how you document the application design process and describes some of the considerations in converting an existing application to IMS/VS.

- **Identifying Application Data**

  One part of application design is gathering and analyzing the data that an application requires. This section describes listing, naming, and documenting data.

- **Designing Local Views**

  This section explains how you develop local views for each business process in the application. A local view records a conceptual data structure and the and the relationships between the pieces of data in the structure. The information that you develop at this stage will be helpful to the data base administrator, or DBA, when the DBA designs the data base.

AN OVERVIEW OF APPLICATION DESIGN

Application design is a process that varies from installation to installation, and from application to application. The overview that's given in this section, and the suggestions about documenting application design and converting existing applications are not the only way that these tasks are performed. Each of these processes varies from installation to installation.

THE TASKS OF APPLICATION DESIGN

The purpose of this overview is to give you a frame of reference so that you'll understand where the techniques and guidelines explained later in this chapter fit into the process. The order in which you perform the tasks described here, and the importance given to each one, depend on your installation. Also, the people involved in each task, and their titles, may differ from installation to installation.

- Using Installation Standards

  Throughout the design process, be sure that you're aware of the standards that your installation has established. Some of the areas in which installations usually establish standards are: naming conventions (for example, for data bases, terminals, data elements); formats for screens and messages; programming and coding conventions (concerning things like the use of common subroutines and macros); and control of and access to the data base. Setting up standards in these areas

is usually an ongoing task that's the responsibility of data base and system administration.

- Following Your Installation's Security Standards

  Security means protection of the installation's resources from unauthorized access and use. Like defining standards, designing an adequate security system is often an ongoing task. As an application is modified or expanded, often the security has to be changed in some way as well. Security is an important consideration in the initial stages of application design. Establishing security standards and requirements is usually the responsibility of an area like system administration. These standards are based on the requirements of the applications at the installation. "Identifying Security Requirements" and "Identifying Online Security Requirements" on page 64 gives some suggestions about the kind of information that you can gather concerning your application's security requirements. This information can be helpful to data base administration and system administration in implementing data base and data communications security.

  Some of the areas in which security is a concern are: access to and use of the data bases; access to terminals; distribution of application output; control of program modification; and transaction and command entry.

- Identifying Application Data

  Identifying the data that an application requires is a major part of application design. One of the tasks of data definition is learning from end users what information will be required to perform the required processing. Once you have listed the data that will be required, other tasks are naming the data and documenting it. "Identyfying Application Data" describes these parts of data definition.

- Providing Input for Data Base Design

  In order to design a data base that meets the requirements of all of the applications that will process the data base, the DBA needs information about the data requirements of each application. One way to gather and supply this information is to design a local view for each of the business processes in your application. A local view is a description of the data that a particular business process requires. "Designing a Local View" explains how you can develop a conceptual data structure and analyze the relationships between the pieces of data in the structure for each business process in the application.

- Designing Application Programs

  Once the overall application flow and system externals have been defined, you define the programs that will perform the required processing. Some of the most important considerations involved in this task are installation standards, security and privacy requirements, and performance requirements. When you develop the specifications for the programs, the specification should include:

  - The security requirements of the application program
  - The input and output data formats and volumes
  - The data verification and validation requirements
  - The logic specifications for the program
  - The performance requirements of the program
  - The recovery requirements of the program

- The linkage requirements and conventions

"Chapter 3. Understanding Online and Batch Processing" describes the considerations in this task, and it describes the types of processing available with IMS/VS. "Chapter 4. Gathering Requirements for Data Base Options" describes some of the data base options that you should be aware of while developing program specifications. "Chapter 5. Gathering Requirements for Data Communications Options" describes some of the data communications options that you should be aware of at this stage.

In addition, you may be asked to provide some information about your application to the people responsible for network and display design.

DOCUMENTING THE APPLICATION DESIGN PROCESS

Recording information about the application design process is valuable to others who work with the application now and in the future. One kind of information that's helpful is information about why you designed the application the way you did. This information can be helpful to people who are responsible for the data base, your IMS/VS system, and the programs in the application—especially if any part of the application has to be changed in the future. Documenting application design is done most thoroughly when it's done during the design process, instead of at the end of it.

A good place to keep this information is in a data dictionary. For example, using the IBM DB/DC Data Dictionary (Program Product Number 5740-XXF), you can define a data processing environment—the application system, the programs, the programs' modules, the IMS/VS system, and so on. The DB/DC Data Dictionary Applications Guide explains how you can use the Data Dictionary for these purposes.

CONVERTING AN EXISTING APPLICATION

One of the main aspects in converting an existing application to IMS/VS is to know what already exists. Before starting to convert the existing system, find out everything you can about the way it works currently. For example, the information below can be of help to you when you begin the conversion:

- The record layouts of all of the records used by the application.
- The number of data element occurrences for each of the data elements.
- The structure of any existing related data bases.

IDENTYFYING APPLICATION DATA

One of the steps of identifying application data is to thoroughly understand the processing that the end user wants done. You need to understand the input data and the required output data to be able to define the data requirements of the application. You also need to understand the business processes that are involved in the end user's processing needs. Three of the tasks involved in identifying application data are:

- Listing the data required by the business process
- Naming the data
- Documenting the data

When you analyze the data that an application requires, you'll find that the data falls into one of two categories:

- An **entity** is a person, place, or thing, that is of interest to the end user, about which data may be recorded. It's what you're keeping information about.
- A **data element** is the smallest named unit of data pertaining to an entity. It's information that describes the entity.

For example, in an education application, "students" and "courses" are both entities; these are two subjects about which you collect and process data. Figure 4 shows some data elements that relate to the student and course entities.

Entities	Data Elements
Students	Student Name Student Number
Course	Course Name Course Number Course Length

Figure 4. Entities and Data Elements

When you store this data in a DL/I data base, groups of data elements are potential segments in the hierarchy; each data element is a potential field in that segment.

LISTING DATA ELEMENTS

This chapter uses as an example a company that provides technical education to its customers. The education company has one headquarters, called HQ, and several local education centers, called Ed Centers.

A class is a single offering of a course on a specific date at a particular Ed Center. There may be several offerings of one course at different Ed Centers; each of these is a separate class. HQ is responsible for developing all of the courses that will be offered, and each Ed Center is responsible for scheduling classes and enrolling students for its classes.

Suppose that one of the education company's requirements is for each Ed Center to print weekly current rosters for all of the classes at the Ed Center. The current roster is to give information about the class and the students enrolled in the class. HQ wants the current rosters to be in the format shown in the sample current roster in Figure 5.

CHICAGO 1/04/80

TRANSISTOR THEORY 41837
10 DAYS
INSTRUCTOR(S): BENSON, R.J. DATE: 1/14/80

STUDENT	CUST	LOCATION	STATUS	ABSENT	GRADE
1.ADAMS, J.W.	XYZ	SOUTH BEND, IND	CONF		
2.BAKER, R.T.	ACME	BENTON HARBOR, MICH	WAIT		
3.DRAKE, R.A.	XYZ	SOUTH BEND, IND	CANC		
.					
.					
.					
33.WILLIAMS, L.R.	BEST	CHICAGO, ILL	CONF		

CONFIRMED = 30
WAIT LISTED = 1
CANCELED = 2

Figure 5. Current Roster

To list the data elements for a particular business process, look at the required output. The current roster shown in Figure 5 is the roster for the class, "Transistor Theory" to be given in Chicago, the Ed Center, starting on January 14, 1980, for ten days. Each course has a course code associated with it—in this case, 41837. The code for a particular course will always be the same; if Transistor Theory is also given in New York, the course code will still be 41837. The roster also gives the name(s) of the instructor(s) who will be teaching the course. Although the example only shows one instructor, a course may require more than one instructor.

For each student, the roster keeps the following information: a sequence number for each student, the student's name, the student's company (CUST), the company's location, the student's status in the class, and the student's absences and grade. All of the above information on the course and the students is input information.

The current date (the date that the roster is printed) appears in the upper right corner (1/04/80). The current date is an example of data that is only output data. The current date is generated by the operating system and is not stored in the data base.

The bottom left-hand corner gives a summary of the class status. This data is not included in the input data; these values will be determined by the program during processing. It is processing data.

When you list the data elements, it's helpful to abbreviate them because you'll be referring to them frequently when you design the local view.

LIST OF CURRENT ROSTER DATA ELEMENTS

EDCNTR The name of the Ed Center giving the class

DATE The date the class starts

CRSNAME The name of the course

CRSCODE The course code

LENGTH The length of the course

INSTRS The name(s) of the instructor(s) teaching the class

STUSEQ# The student's sequence number

STUNAME The student's name

CUST The name of the student's company

LOCTN The location of the student's company

STATUS The student's status in the class—whether the student is confirmed, wait-listed, or canceled

ABSENCE The number of days on which the student was absent

GRADE The student's grade for the course

Once you've listed the data elements, choose the major entity that these elements describe. In this case, the major entity is class. There is a lot of information about each student as well, and some information about the course in general, but together all this information relates to a specific class. If the information about each student (status, absence, grade, for example) isn't related to a particular class, then the information is meaningless. This holds true for the data elements at the top of the list as well: the Ed Center, the date the class starts, and the instructor don't mean anything unless you know what class they're describing.

NAMING DATA ELEMENTS

Some of the data elements that your application uses may already exist and be named. After you have listed the data elements, find out if any of them exist by checking with the DBA, or the equivalent at your installation.

Before you start naming data elements, be sure you're aware of the naming standards at your installation. When you name data elements, use the most descriptive names possible. Remember that since other applications will probably be using at least some of the same data, the names should mean the same thing to everyone. Try not to limit the name to a meaning that makes sense only in your application; use global names rather than local names. A global name is a name whose meaning is clear outside of any particular application. A local name is a name that must be seen in the context of a particular application to be understood.

One of the problems with using local names is that you can develop synonyms, two names for the same data element. For example, in the current roster example, suppose the student's company was referred to simply as "company" instead of "customer." But suppose the accounting department for the education company used the same piece of data in a billing application—the name of the student's company—and referred to it as "customer." This would mean that two business processes were using two different names for the same piece of data. At worst, this could lead to redundant data if no one realized that "customer" and "company" contained the same data. To solve this, use a global name that is recognized by both departments using this data element. In this case, customer is more easily recognized and the better choice. This name uniquely identifies the data element and has a specific meaning within the education company.

Homonyms are the opposite of synonyms. A homonym is one word for two different things. For example, suppose HQ, for each course that is taught, assigns a number to the course as it is developed and calls this number the "sequence number." The Ed Centers, as they receive student enrollments for a particular class, assign a number to each student as a means of identification within the class. The Ed Centers call this number the "sequence number." Thus HQ and the Ed Centers are using the same name for two separate data elements. You can solve the problem by qualifying the names. The number that HQ assigns to each course can be called "course code" (CRSCODE), and the number that the Ed Centers assign to their students can be called "student sequence number" (STUSEQ#).

The name of a particular data element must identify that element and describe it as much as possible. Data element names should be:

- Unique: The name is clearly distinguishable from other names.
- Self-explanatory: The name is easily understood and recognized.
- Concise: The name is descriptive in a few words.
- Universal: The name means the same thing to everyone.

DOCUMENTING APPLICATION DATA

After you've determined what data elements a business process requires, it's a good idea to record as much information about each of these elements as possible. This information is useful to the DBA. Make sure that you are aware of any standards that your installation has established about data documentation. Many installations have standards concerning what information should be recorded about data, and how and where that information should be recorded. The amount and type of this information may vary from installation to installation; the list below is the type of information that is often recorded:

- The descriptive name of the data element. Data element names should be precise, yet they should be meaningful to people who aren't familiar with the application, as well as to the people who are familiar with the application.
- The length of the data element. This will be used to determine segment size and segment format.
- The character format—whether the data is alphameric, hexadecimal, packed decimal, or binary. The programmer will need this information.
- The range of possible values for the element. This is important for validity checking.
- The default value. The programmer will also need this information.
- The number of data element occurrences. This information will help the DBA to determine the space required for this data, and it affects performance considerations.
- How the business process affects the data element—will it just read it, or will it be updating it as well? This information will determine the processing option that is coded in the PSB for the application program.

Other information that you should record about the data concerns control considerations, such as maintenance and security:

- If the format of a particular data element changes, which business processes does that affect? For example, if an education data base has as one of its data elements a 5-digit code for each course, and the code is changed to 6 digits, which business processes will this affect?
- Where is the data now? Know the sources of the data elements required by the application.
- Which business processes make changes to a particular data element?
- Are there security requirements about the data in your application? For example, you would not want information such as employees' salaries available to everyone at the installation.

- Which department owns and controls the data?

One way to gather and record this information is to use a form similar to the one shown in Figure 6. Again, the amount and type of data that your record depends primarily on the standards at your installation. This form is provided as an example of the kind of data that is useful to record.

ID #	Data Element Name	Length	Format	Allowed Values	Null Value	Default Value	Number of Occurrences
5	Course Code	5 bytes	Char.	00100-90000	00000	N/A	There are 200 courses in the curriculum. An average of 10 are new/revised per year; an average of 5 are dropped per year.
25	Status	4 bytes	Char.	CONF WAIT CANC	blanks	WAIT	1 per student
36	Student Name	20 bytes	Char.	Alpha only	blanks	N/A	There are 3-100 students per class; this is an average of 40 per class.

Figure 6. Example of Data Elements Information Form

A data dictionary is a good place to record the facts about the application's data. When you are analyzing data, a dictionary can help you find out whether or not a particular data element already exists, and if it does, its characteristics. With the IBM DB/DC Data Dictionary, you can use the Data Dictionary online to determine what segments exist in a particular data base, to determine the fields that segments contain, and so on. You can also use it to create reports involving the same information.

The DB/DC Data Dictionary Applications Guide explains how you can use the Data Dictionary for these purposes.

DESIGNING A LOCAL VIEW

A local view is a description of the data that an individual business process requires. It includes a list of the data elements, a conceptual data structure that shows how you've grouped data elements by the entities that they describe, and the relationships between each of the groups of data elements. A group of data elements is called a data aggregate. Once you have grouped data elements by the entity they describe, you can determine the relationships between the aggregates. These relationships are called mappings. Based on the mappings, you can design a conceptual data structure for the business process. You should document this process as well.

ANALYZING DATA RELATIONSHIPS

When you analyze data relationships, you are developing conceptual data structures for the business processes in your application. Data structuring is a way to analyze the relationships between the data elements required by a business process, not a way to design a data base. The decisions about

segment formats and contents belong to the DBA. The information you develop is input for designing a data base.

Data structuring can be done in many different ways. The method explained in this section is an example.

Grouping Data Elements into Hierarchies

A group of data elements that describes a particular entity is called a data aggregate. For example, the data elements STUSEQ#, STUNAME, CUST, LOCTN, STATUS, ABSENCE, and GRADE all describe a student. This group of data elements is called the student data aggregate. STUSEQ#, STUNAME, CUST, LOCTN, STATUS, ABSENCE, and GRADE are the names of data elements.

Data elements have values as well; for the student data elements the values are a particular student's sequence number, the student's name, company, company location, the student's status in the class, the student's absences, and grade. The names of the data aggregate are not unique; all the students in the class are described in the same terms. The combined values, however, of a data aggregate occurrence are unique. No two students can have the same values in each of these fields.

As you group data elements into data aggregates and data structures, look at the data elements that make up each group and choose one or more data elements that will uniquely identify that group. A data aggregate's key is the data element or group of data elements in the aggregate that uniquely identifies the aggregate. Sometimes you have to use more than one data element to uniquely identify an aggregate.

By following the three steps explained in this section you can develop a conceptual data structure for a business process's data. This does not mean that you are developing the logical data structure that the program that performs the business process will end up with. The three steps are:

1. Isolate repeating data elements in a single occurrence of the data aggregate.
2. Isolate duplicate values in multiple occurrences of the data aggregate.
3. Group data elements with their controlling keys.

1. ISOLATING REPEATING DATA ELEMENTS: For the first step, look at a single occurrence of the data aggregate. Figure 7 shows what this looks like for the class aggregate.

DATA ELEMENT	CLASS AGGREGATE OCCURRENCE
EDCNTR	CHICAGO
DATE(START)	1/14/80
CRSNAME	TRANSISTOR THEORY
CRSCODE	41837
LENGTH	10 DAYS
INSTRS	multiple
STUSEQ#	multiple
STUNAME	multiple
CUST	multiple
LOCTN	multiple
STATUS	multiple
ABSENCE	multiple
GRADE	multiple

Figure 7. Single Occurrence of Class Aggregate

The fields that say multiple are the fields that repeat. Separate the data elements that repeat by moving them to a lower level; keep data elements with their controlling keys.

The data elements that repeat for a single class are STUSEQ#, STUNAME, CUST, LOCTN, STATUS, ABSENCE, and GRADE. INSTRS is also a repeating data element because some classes require two instructors, although this class requires only one.

When you isolate repeating data elements, you have the structure shown in Figure 8.

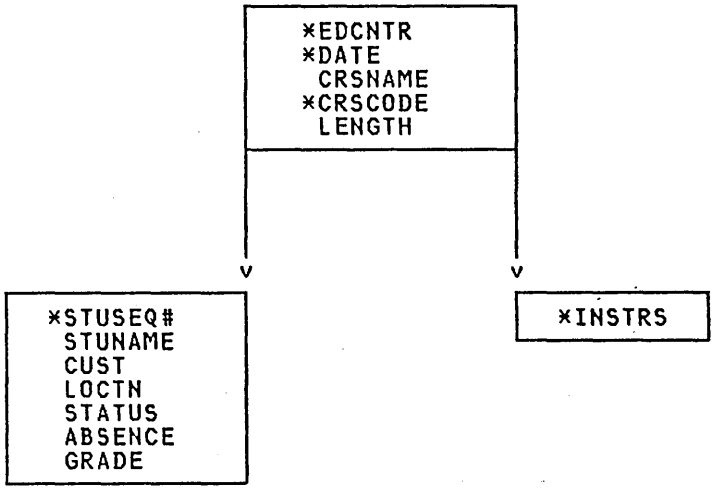

Figure 8. Current Roster Step 1

The asterisks in Figure 8 signify the data elements that make up the key. For the class aggregate, it takes more than one of the data elements to identify the course, so you need more than one data element to make up the key.

After you have shifted repeating data elements, make sure that each element is in the same group as its key. INSTRS is separated from the group of data elements describing student because the information about instructors is unrelated to the information about the students. The student sequence number does not control who the instructor is. A dependent aggregate's key is made up of the concatenated keys of all of the aggregates above the dependent aggregate. This is because a dependent's key doesn't mean anything if you don't know the keys of the higher aggregates. For example, if you knew that a student's sequence number was 4, you would be able to find out all of the information about the student associated with that number. This number would be meaningless, however, if it were not associated with a particular class. But, since the key for the student aggregate is made up of Ed Center, date, and course code, you would know what class the student was in.

Figure 8 shows these aggregates with the following keys:

- Course aggregate: EDCNTR, DATE, CRSCODE
- Student aggregate: EDCNTR, DATE, CRSCODE, STUSEQ#
- Instructor aggregate: EDCNTR, DATE, CRSCODE, INSTRS

2. ISOLATING DUPLICATE VALUES: For the second step, you need to look at multiple occurrences of the aggregate—in this case the values you might have for two classes. Figure 9 shows multiple

occurrences of the data elements. As you look at Figure 9, look for duplicating values. Remember that both occurrences describe one course.

DATA ELEMENT LIST	OCCURRENCE #1	OCCURRENCE #2
EDCNTR	CHICAGO	NEW YORK
DATE(START)	1/14/80	3/10/80
CRSNAME	TRANS THEORY	TRANS THEORY
CRSCODE	41837	41837
LENGTH	10 DAYS	10 DAYS
INSTRS	multiple	multiple
STUSEQ#	multiple	multiple
STUNAME	multiple	multiple
CUST	multiple	multiple
LOCTN	multiple	multiple
STATUS	multiple	multiple
ABSENCE	multiple	multiple
GRADE	multiple	multiple

Figure 9. Multiple Occurrences of Class Aggregate

The data elements that say multiple are the data elements that repeat. The values in these elements will not be the same. The aggregate will always be unique for a particular class.

In this step, compare the two occurrences and shift the fields with duplicate values to a higher level. If you need to, choose a key for aggregates that do not yet have keys.

CRSCODE, CRSNAME, and LENGTH are the fields that can have duplicate values. A lot of this process is common sense. Student status and grade, although they can have duplicate values, should not be separated because they are not meaningful values by themselves. These values would not be used to identify a particular student. This becomes clear when you remember to keep data elements with their controlling keys.

When you isolate duplicate values, you have the structure shown in Figure 10.

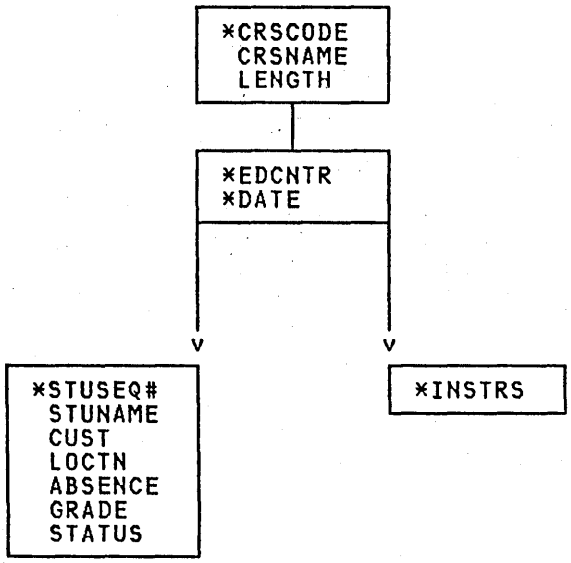

Figure 10. Current Roster Step 2

3. GROUPING DATA ELEMENTS WITH THEIR CONTROLLING KEYS: The third step is often a check on the first two steps. Sometimes the first two steps have already done what this step tells you to do.

At this stage, make sure that each data element is in the group that contains its controlling key. The data element should depend on the full key. If the data element only depends on part of the key, separate the data element along with the partial key that it depends on.

In this example, CUST and LOCTN do not depend on the STUSEQ#. They are related to the student, but they don't depend on the student. They identify the company and company address of the student.

CUST and LOCTN are not dependent on the course, the Ed Center, or the date. They are separate from all of these things. Since a student is only associated with one CUST and LOCTN, but a CUST and LOCTN can have many students attending classes, the CUST and LOCTN aggregate should be above the student aggregate. Figure 11 shows what the structure looks like when you separate CUST and LOCTN.

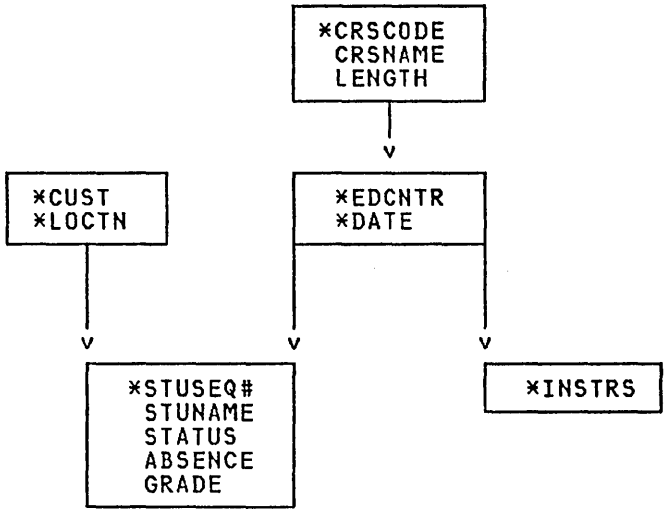

Figure 11. Current Roster Step 3

Where the customer and location information will be located is part of data base design. Data structuring should separate any inconsistent data elements from the rest of the data elements.

Determining Mappings

Once you have arranged the data aggregates in a conceptual data structure, you can examine the relationships between the data aggregates. A mapping between two data aggregates is the quantitative relationship between the two. The reason that you record mappings is that they reflect relationships between segments in the data structure that you've developed. If you store this information in a DL/I data base, the DBA can construct a data base hierarchy that will satisfy all of the local views, based on the mappings. In determining mappings, it's easier to refer to the data aggregates by their keys, rather than by their collected data elements.

There are two possible relationships between any two data aggregates:

- One-to-many

For each segment A, there are one or more occurrences of segment B. For example, for each class, there are multiple students. Mapping notation shows this like so:

Class <-------->> Student

- Many-to-many

  Segment B has many A segments associated with it and segment A has many B segments associated with it. In a hierarchic data structure, a parent can have one or more children, but each child can be associated with only one parent. The many-to-many association does not fit into a hierarchy, because each child can be associated with more than one parent. This is covered in Chapter 3, "Analyzing Data Requirements," in the IMS/VS Data Base Administration Guide. Many-to-many relationships occur between segments in two business processes. A many-to-many relationship indicates a conflict in the way that two business processes need to process those data aggregates. If you use DL/I, you can solve this kind of processing conflict by using secondary indexing or logical relationships.

The mappings for the current roster are:

- Course <--------->> Class

  For each course, there may be several classes scheduled, but a class is associated with only one course.

- Class <--------->> Student

  A class has many students enrolled in it, but a student may be in only one class offering of this course.

- Class <--------->> Instructor

  A class may have more than one instructor, but an instructor only teaches one class at a time.

- Customer/location <--------->> Student

  A customer may have several students attending a particular class, but each student is only associated with one customer and location.

LOCAL VIEW EXAMPLES

This section goes through three more examples of designing a local view. These examples are the Schedule of Classes, the Instructor Skills Report, and the Instructor Schedules. This section does not explain the steps of designing a local view; it simply takes you through the examples. Each example shows the two parts of designing a local view:

1. Gather the data. For each example, the data elements are listed and two occurrences of the data aggregate are shown. Two occurrences are shown because you need to look at both occurrences when you look for repeating fields and duplicate values.

2. Analyze the data relationships. First, group the data elements into a conceptual data structure using these three steps:

   a. Isolate repeating data elements in a single occurrence of the data aggregate by shifting them to a lower level. Keep data elements with their controlling keys.

   b. Isolate duplicating values in two occurrences of the data aggregate by shifting those data elements to a higher

level. Again, keep data elements with their controlling keys.

c. Group data elements with their controlling keys. Make sure that all of the data elements within one aggregate have the same controlling key. Isolate any that don't.

Then, determine the mappings between the data aggregates in the data structure you've developed.

Schedule of Courses

HQ keeps a schedule of all of the courses given each quarter and distributes it monthly. HQ wants the schedule to be sorted by course code and printed in the format shown in Figure 12.

```
                    COURSE SCHEDULE

COURSE:  TRANSISTOR THEORY      COURSE CODE:  41837
LENGTH:  10 DAYS                PRICE:  $280

     DATE                       LOCATION
     ------------------------------------
     APRIL 14                   BOSTON
     APRIL 21                   CHICAGO
     .
     .
     .
     NOVEMBER 18                LOS ANGELES
```

Figure 12. Schedule of Classes

1. Gather the data. Figure 13 lists the data elements and two occurrences of the data aggregate.

Data Elements	Occurrence 1	Occurrence 2
CRSNAME	TRANS THEORY	MICRO PROG
CRSCODE	41837	41840
LENGTH	10 DAYS	5 DAYS
PRICE	$280	$150
DATE	multiple	multiple
EDCNTR	multiple	multiple

Figure 13. Class Schedule Data Elements

2. Analyze the data relationships. First, group the data elements into a conceptual data structure.

   a. Isolate repeating data elements in one occurrence of the data aggregate, as shown in Figure 14.

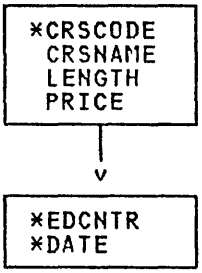

Figure 14. Class Schedule Step 1

b. Next, isolate duplicate values in two occurrences of the data aggregate.

   There are no duplicate values in this data aggregate.

c. Group data elements with their controlling keys.

   Data elements are grouped with their controlling keys in the present structure. No changes are necessary for this step.

   Once you've developed a conceptual data structure, determine the mappings for the data aggregates.

   The mapping for this local view is:

   Course <--------->> Class

Instructor Skills Report

Each of the Ed Centers needs to print a report giving the courses that each of its instructors is qualified to teach. The report is to be in the format shown in Figure 15.

INSTRUCTOR SKILLS REPORT

INSTRUCTOR	COURSE CODE	COURSE NAME
BENSON, R. J.	41837	TRANS THEORY
MORRIS, S. R.	41837	TRANS THEORY
	41850	CIRCUIT DESIGN
	41852	LOGIC THEORY
.		
.		
.		
REYNOLDS, P. W.	41840	MICRO PROG
	41850	CIRCUIT DESIGN

Figure 15. Instructor Skills Report

1. Gather the data. Figure 16 lists the data elements and two occurrences of the data aggregate.

DATA ELEMENTS	OCCURRENCE 1	OCCURRENCE 2
INSTR	BENSON, R. J.	MORRIS, S. R.
CRSCODE	multiple	multiple
CRSNAME	multiple	multiple

Figure 16. Instructor Skills Data Elements

2. Analyze the data relationships. First, group the data elements into a conceptual data structure.

   a. Isolate repeating data elements in one occurrence of the data aggregate as shown in Figure 17.

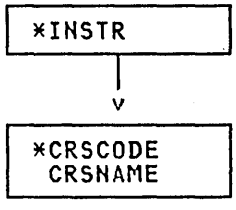

Figure 17. Instructor Skills Step 1

   b. Isolate duplicate values in two occurrences of the data aggregate.

      There are no duplicate values in this data aggregate.

   c. Group data elements with their controlling keys.

      All data elements are grouped with their keys in the current data structure. You don't have to make any changes to this data structure.

   Determine the mappings for the data aggregates.

   The mapping for this local view is:

   Instructor <--------->> Course

Instructor Schedules

HQ wants to produce a report giving the schedules for all the instructors. Figure 18 shows what the report is to look like.

INSTRUCTOR SCHEDULES

INSTRUCTOR	COURSE	CODE	ED CENTER	DATE
BENSON, R. J.	TRANS THEORY	41837	CHICAGO	1/14/80
	TRANS THEORY	41837	NEW YORK	3/03/80
	MICRO PROG	41840	NEW YORK	3/17/80
MORRIS, S. R.	CIRCUIT DES	41850	CHICAGO	1/21/80
	LOGIC THEORY	41862	BOSTON	2/25/80
REYNOLDS, B. H.	CIRCUIT DES	41850	LOS ANGELES	3/10/80

Figure 18. Instructor Schedules

1. Gather the data. Figure 19 lists the data elements and two occurrences of the data aggregate.

DATA ELEMENTS	OCCURRENCE 1	OCCURRENCE 2
INSTR	BENSON, R. J.	MORRIS, S. R.
CRSNAME	multiple	multiple
CRSCODE	multiple	multiple
EDCNTR	multiple	multiple
DATE(START)	multiple	multiple

Figure 19. Instructor Schedules Data Elements

2. Analyze the data relationships. First, group the data elements into a conceptual data structure.

   a. Isolate repeating data elements in one occurrence of the data aggregate as shown in Figure 20.

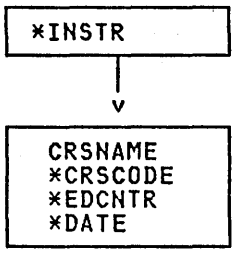

Figure 20. Instructor Schedules Step 1

   b. Isolate duplicate values in two occurrences of the data aggregate as shown in Figure 21.

      In this example, CRSNAME and CRSCODE can be duplicated for one instructor or for many instructors, for example, 41837 for Benson and 41850 for Morris and Reynolds.

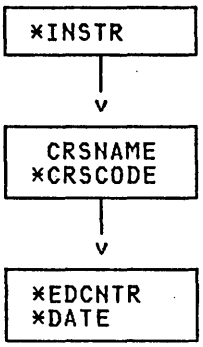

Figure 21. Instructor Schedules Step 2

   c. Group data elements with their controlling keys.

      All data elements are grouped with their controlling keys in the current data structure. You don't have to make any changes to the current data structure.

Determine the mappings for the data aggregates.

The mappings for this local view are:

Instructor <--------->> Course
Course <--------->> Class

Combining the requirements of the four examples presented in this chapter and designing a hierarchic structure for the data base based on these requirements are covered in Chapter 3, "Analyzing Data Requirements," in the IMS/VS Data Base Administration Guide.

CHAPTER 3. UNDERSTANDING ONLINE AND BATCH PROCESSING

This chapter describes IMS/VS online and batch processing, and the kinds of programs that perform both kinds of processing. How online and batch programs are scheduled, how they update the data base, and how they are recovered and restarted after a program or system failure are different. This chapter has three major sections:

- **Online Processing**

  "Online processing" is the kind of processing that's often done during prime shift. Online processing allows many people to use the system at the same time, through terminals. This section describes the characteristics of online processing and it explains the types of programs that run online.

- **Batch Processing**

  "Batch processing" is the kind of processing that's often done at night or on the weekends—separately from online processing. This section explains the characteristics of batch processing.

- **Summarizing IMS/VS Application Program Characteristics**

  The last section of this chapter contains a chart that summarizes and compares some of the characteristics of each type of IMS/VS application program.

ANALYZING PROCESSING REQUIREMENTS

One of the steps of application design is to decide how the business processes, or tasks, that the end user wants done can be best grouped into a set of programs that will efficiently perform the required processing. Some of the considerations in analyzing processing requirements are:

- When the task must be executed:
  - Will it be scheduled unpredictably, for example on terminal demand; or periodically, for example, weekly?
- How the program that performs the task is invoked:
  - Will it be invoked online, where response time is crucial; or by batch job submission, where a slower response time is acceptable?
- The consistency of the processing components:
  - Does this task involve more than one type of program logic? For example, does it involve mostly retrievals, and only one or two updates? If so, you should consider separating the updates into a separate program.
  - Does this task involve several large groups of data? If it does, it might be more efficient to separate the programs by the data they access.

- Any special requirements about the data or processing:

  Maintenance—how often must the data be updated?
  Security—should access to the program be restricted?
  Recovery—are there special recovery considerations in the program's processing?
  Integrity—do other departments use the same data?

There is not always a "right" answer when you are trying to determine how many programs can most efficiently do the required processing. It is often a matter of common sense, and not hard and fast rules. As you look at each task, examine the data and processing that each task involves. If a task requires different types of processing and has different time limitations (for example, weekly as opposed to monthly), that task may be more efficiently performed by several programs.

As you define each program, it's a good idea for maintenance and recovery reasons to keep programs as simple as possible. The simpler a program is—the less it does—the easier it is to maintain and restart after a program or system failure. At the same time, if the data that the application requires is in a large data base, it may be more efficient to have one program do more of the processing than usual. These are considerations that depend on the processing and the data of each application.

When doing this, it's a good idea to document each of the user tasks. Be sure you're aware of installation standards in this area. Some of the information that's often collected is when the task is to be invoked, a functional description, and maintenance, security, and recovery requirements. For example, for the Current Roster process described in "Chapter 2. Analyzing Application Requirements," you might record the information shown in Figure 22. The frequency is determined by the number of classes (20) for which the Ed Center will print current rosters each week.

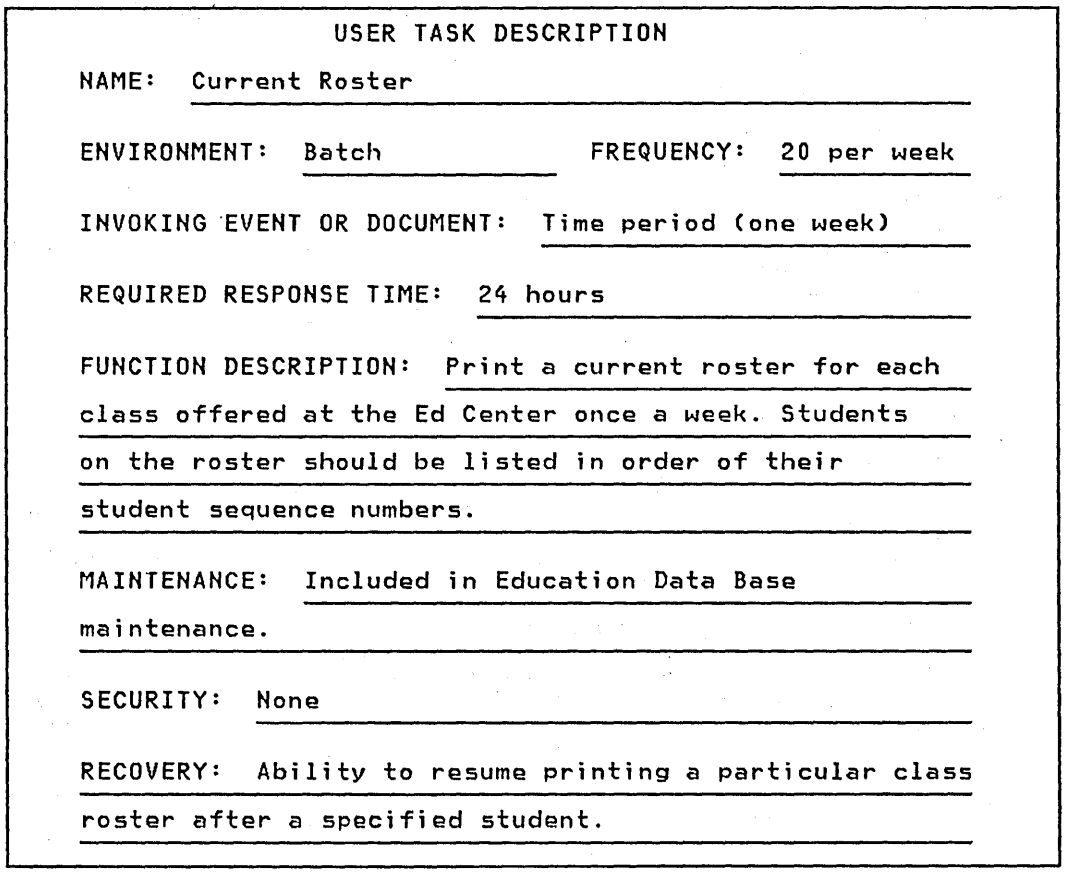

Figure 22. Current Roster Task Description

ONLINE PROCESSING

There are three major kinds of IMS/VS application programs that run online. These are:

- Message processing programs, or MPPs
- Fast Path programs
- Batch message programs, or BMPs

Online processing is typically done during the day, when the system is used most heavily. Online programs process requests and communicate with end users. Because online programs use IMS/VS resources concurrently, these programs run under the control of the IMS/VS control region. Each type of online program—an MPP, a Fast Path program, and a BMP—runs in a region that is dependent on the control region for access to the message queues and the data bases. Some of the application requirements that online programs can answer are:

- The information in the data base must be available to many users.
- The programs needs to communicate with terminals and other programs.
- The program must be available to users at remote terminals.

- Response time is crucial; you need the results of the application program's processing immediately.
- The information in the data base must be current.

There are two types of processing that an online program can do. Some online programs process messages. These are the programs that communicate with terminals and other programs through the message queues. When a person at a terminal enters a request for information, the request is called a message. IMS/VS collects messages on a message queue and schedules a program when there is a message for it to process. The program then retrieves its messages one at a time, processes them, and replies to the originators.

For example, a bank teller enters a request at a terminal for the current balance in a certain checking account. This is a message. Part of the message is a transaction code that identifies the processing request and associates the request with the application program that can answer the request. IMS/VS routes the message to the message queue, and the program retrieves the message, gets the requested information, and replies to the bank teller's terminal.

Online programs can also perform batch processing. These programs don't have to communicate with terminals or other programs for their input and output; they are simply batch-type programs that run online. Some of the reasons that you use an online program to do batch processing are:

- You want a batch program to access online data bases.
- You don't want to wait until the night or the weekend to update the data base because the data will not be as current.
- You don't want to degrade the response time of an MPP by having the MPP handle its data base updates.

There are three kinds of online programs. All of them are able to process messages and access IMS/VS data bases; some of them can perform batch processing as well.

- MPPs are IMS/VS application programs that process messages to and from the IMS/VS message queues. You use them only for message processing.
- Fast Path programs are online programs that emphasize fast processing of simple data structures and high volumes of transactions. There are two types of Fast Path application programs you can use.
  - A message-driven Fast Path program accesses the Fast Path message queues for its input; it is similar to an MPP.
  - A nonmessage-driven Fast Path program processes Fast Path data bases. It does not communicate with terminals or other programs, except in mixed mode. "Mixing Fast Path and IMS/VS Processing" explains mixed-mode processing. A nonmessage-driven Fast Path program is similar to a BMP.
- BMPs are IMS/VS application programs that can perform message processing or batch processing. BMPs have characteristics of both online and batch programs in that they run online, but they are started with JCL. Unlike MPPs, BMPs can access OS/VS files. There are two kinds of BMPs. One kind accesses the message queue for its input, the other doesn't. Both types can send output to the message queue.
  - A transaction-oriented BMP accesses message queues for its input. What makes it different from an MPP is that it doesn't have to access the message queues; it can process input from OS/VS files, and it can create OS/VS files.

- A batch-oriented BMP does not access the message queue for input; it is simply a batch program that runs online. It can send its output to the message queue. Batch-oriented BMPs are usually shorter than typical batch programs because you don't want to tie up online resources with a long-running program.

MESSAGE PROCESSING

The programs that can process messages from terminals and other programs are MPPs, message-driven Fast Path programs, and transaction-oriented BMPs. Before you look at each of these types of programs individually, there are some concepts of message processing that you need to understand.

When a person at a terminal enters a request for processing, that request is called a message. The message that the person enters can be one of three types:

- If the message starts with a transaction code, the message is a processing request for an application program. In this case, IMS/VS puts the message on the message queue and schedules the application program that can process that transaction.
- If the message starts with the name of a logical terminal, the control region sends the message to the queue and then to the logical terminal specified in the message. This is called a message switch.
- If the message starts with a slash ("/"), it is a command. IMS/VS executes the command.

The important thing to notice about the messages that a terminal can enter is that only messages that contain transaction codes go to an application program. The control region handles the other two types of messages.

When a program that processes messages is scheduled, its main purpose is to process the messages that are waiting for it. It may or may not have to access the data base to do this. Generally, this type of program goes through these four steps in its processing:

1. Retrieve a message from the message queue.
2. Process the message; this includes getting any information you need from the data base.
3. Reply to the terminal or program that sent the input message.
4. Check to see if there are any more messages. If there are, repeat the process. If there aren't, terminate the program.

How IMS/VS Identifies Terminals

When a person at a terminal sends a message to a program, IMS/VS places the name of the terminal that sent the message in the I/O PCB mask. This is not the name of the physical terminal; IMS/VS uses what are called "logical terminals." A logical terminal is a name that is related to a physical terminal. One physical terminal can have one or more logical terminals associated with it. The advantage of using a logical terminal name instead of the physical terminal itself is that you don't have to worry about the physical addresses of the terminals you're working with; IMS/VS makes sure that your messages go to the physical terminal associated with the name you're using. If a physical terminal becomes inoperative, the master terminal operator can dynamically reassign the logical terminal associated with the inoperative physical terminal to another physical terminal. This means that the output from the MPP

would go to a new physical terminal without affecting the MPP's processing.

How IMS/VS Protects Online Data: Sync Points

IMS/VS data communications makes it possible for people at terminals and application programs to access the data at one time. Because of this, IMS/VS has to have control over which users and programs access the data, and when. If IMS/VS doesn't control the data, several application programs could access and update the same segment at the same time and no one would know the correct value of the segment. To avoid this, IMS/VS serializes and enqueues segments for online application programs.

IMS/VS does two things to protect data integrity. First, when an online program accesses a data base record, IMS/VS puts a hold on that record until the program is finished with the record. If the program is only reading from the data base record, IMS/VS considers the program to be finished with it when the program moves to another data base record in the same data structure that contains the first data base record. IMS/VS then takes the hold off the previous data base record and makes it available to the next program that wants to use it. If, on the other hand, the program is updating some of the data in the data base record, IMS/VS has to hold the record for a longer period of time to make sure that the update is valid.

The second thing that IMS/VS does to protect data involves synchronization points. A synchronization point, or sync point, is a place in a program at which the program indicates to IMS/VS that all of the processing thus far is accurate. When a program reaches a sync point, that means that even if the program terminates abnormally after the next call, the processing up to the sync point is accurate. When an online program updates a data base record, IMS/VS doesn't release the record until the program reaches a sync point. Where sync points occur in a program depends on the type of processing the program does.

In addition to indicating to IMS/VS that a program's processing is accurate, sync points establish places in the program from which the program can be restarted.

One way to understand the differences between the online message programs—MPPs, message-driven Fast Path programs, and transaction-oriented BMPs—is to look at specific ways in which they differ and compare them. The main ways in which these three types of programs differ are:

- Where sync points occur in each type of program
- How IMS/VS handles recovery for each type of program
- When IMS/VS applies the updates produced by each type of program

MPPs

An MPP is an online program that processes messages and accesses online data bases. It is controlled by the IMS/VS control region. You use an MPP when you want the program to answer requests entered at a terminal; these requests deal with IMS/VS online data bases.

An MPP cannot process OS/VS files or GSAM data bases. GSAM, or Generalized Sequential Access Method, is used to convert OS/VS files into nonhierarchic data bases that IMS/VS can checkpoint.

MPP OPTIONS: An MPP has several options concerning its interactions with terminals. The MPP can send its output to the same terminal that sent the input message, or it can send output to other terminals and application programs as well.

An MPP can also process a conversation. Conversational processing allows the person at the terminal to interact more than once with an application program or a set of application programs. In other words, the person at the terminal might have several parts to a transaction, instead of only one.

The messages that a terminal sends to an MPP are made up of segments. A message from a terminal is made up of as many segments as necessary. The MPP retrieves one message segment at a time until IMS/VS indicates to the program that it has retrieved all the segments of that message.

SYNC POINTS IN AN MPP: Where sync points occur in an MPP depends on whether the program is single or multiple mode. When an MPP reaches a sync point, the program indicates to IMS/VS that its processing thus far is accurate. When this happens, IMS/VS releases the resources that the program has enqueued since the last sync point, and it sends the output messages that the program has created to their destinations.

When you specify single mode for an MPP, a sync point occurs each time the MPP issues a call for a new message. This means that IMS/VS sends the output messages that the program creates to their destinations at each sync point. Single-mode is good for recovery reasons because, in the event of a program or system failure, a single-mode MPP can be restarted from the most recent call for a new message.

If the program is multiple mode, the only sync points in the program occur when you issue a checkpoint call to cause a sync point, and when the program terminates. These are the only times during the program that IMS/VS sends the program's output messages to their destinations. Because there are fewer sync points to process in multiple-mode MPPs than there are in single-mode MPPs, multiple-mode MPPs may be able to give slightly higher performance than single-mode MPPs.

When an MPP reaches a sync point, IMS/VS does two things:

1. IMS/VS releases any data base records that the MPP has updated since the last sync point so that these records are available to other application programs.

2. IMS/VS sends output messages from the MPP to their final destinations. Until the program reaches a sync point, IMS/VS holds the program's output messages at a temporary destination. This ensures that, if the program terminates abnormally, terminals and programs will not receive inaccurate data from the MPP.

RECOVERY IN AN MPP: When an MPP terminates abnormally, IMS/VS:

- Backs out all of the changes that the MPP has made to the data base since the last sync point

- Throws away the output messages that the MPP has produced since the last sync point

- Throws away the input message the MPP had retrieved and was processing when the MPP terminated abnormally

Once the master terminal operator has restarted the MPP, IMS/VS gives the MPP the next message.

In a single-mode MPP, the program reaches a sync point each time it issues a call to retrieve a new message from the message queue. This means that when IMS/VS restarts the program after the program has terminated abnormally, the program will start by processing the next message.

On the other hand, when a multiple-mode MPP terminates abnormally, IMS/VS can only restart it from a checkpoint call. This can mean that instead of having only the most recent message

to reprocess, the MPP might have several messages to reprocess. This depends on when the MPP issued the last checkpoint call. Using single mode, rather than multiple mode, simplifies recovery for the MPP.

WHEN IMS/VS SCHEDULES AN MPP: When a person at a terminal wants to process a particular transaction, the person enters a transaction code. A transaction code is associated during system definition with the program that can process it. An MPP can be defined to process only one transaction code, or it can be defined to process several different transaction codes.

When IMS/VS receives a message with a transaction code, IMS/VS checks the priority for the program associated with the transaction code. If there are messages waiting for other programs with higher priorities, IMS/VS schedules those programs first. Program priorities are decided by the DB/DC system administrator.

When IMS/VS schedules an MPP, the MPP processes the message or messages that are waiting for it—within limits—and then terminates. The limits are defined by the IMS/VS DB/DC system administrator when the application program is defined. For each MPP, the DB/DC system administrator specifies:

- The number of messages for a particular transaction code that the MPP can process in a single scheduling
- The amount of time (in seconds) in which the MPP is allowed to process a single transaction

The MPP can process several messages during one scheduling, or it can process only one message during a scheduling.

Message-Driven Fast Path Programs

A message-driven Fast Path program is similar to an MPP. The purpose of Fast Path is to provide better response than you can normally get from an MPP or a transaction-oriented BMP. As a result, message-driven Fast Path programs do not have all the options available to them that MPPs do. For example, message-driven Fast Path programs cannot use conversational processing; they must respond to the terminal that sent the message before the terminal can enter any more requests. This is called transaction response mode. In addition, messages to Fast Path programs can only be single segment messages. This helps Fast Path achieve its high performance.

All the transactions that Fast Path programs process must be wait-for-input transactions. This means that when the program finishes processing the transaction, the program remains in main storage, even when there are no more messages for it to process. This is another part of Fast Path that gives it high performance and response rate.

FAST PATH DATA BASES: Fast Path programs access two kinds of data bases. (They can also access IMS/VS data bases; "Mixing Fast Path and IMS/VS Processing" explains this.) These are special data bases that, again, are designed for high performance.

- **Main storage data bases, or MSDBs,** are data bases that contain only root segments. These data bases are designed to handle the most frequently used data. The data is held in main storage. MSDBs can be owned by a particular terminal; only that terminal can update that data. If an MSDB isn't owned by a particular terminal, then the data in the MSDB is generally available.
- **Data entry data bases, or DEDBs,** are data bases designed to handle large volumes of detailed data. Each DEDB contains a root segment and up to seven dependent segment types and resides on direct access storage.

SYNC POINTS IN MESSAGE-DRIVEN FAST PATH PROGRAMS: Message-driven Fast Path programs must be defined as single mode. This means that a sync point occurs each time a message-driven Fast Path program retrieves a message and when the program issues a checkpoint call. IMS/VS holds the data base updates from a Fast Path program in buffers between sync points; it isn't until the Fast Path program reaches a sync point that IMS/VS actually applies the updates to the data base. At a Fast Path sync point, IMS/VS first checks to make sure that there's room for the updates in the data base, then it applies the updates and releases the IMS/VS and Fast Path resources that the program has been holding.

RECOVERY IN A MESSAGE-DRIVEN FAST PATH PROGRAM: When a message-driven Fast Path program terminates abnormally, IMS/VS doesn't have to back out the data base updates, because data base updates from Fast Path programs aren't applied until the program reaches a sync point. Nor does IMS/VS send output messages until the program reaches a sync point.

Since each call to the message queue is a sync point, message-driven Fast Path programs, like single-mode MPPs, don't have to issue checkpoint calls.

WHEN IMS/VS SCHEDULES A MESSAGE-DRIVEN FAST PATH PROGRAM: Fast Path programs are not scheduled in the same way that MPPs are. Fast Path can use multiple copies of an application program so that a transaction from a terminal doesn't have to wait very long before it's processed. This improves response time to Fast Path transactions significantly. Messages are handled on a first-in, first-out basis. Fast Path transactions bypass IMS/VS scheduling.

Transaction-Oriented BMPs

Although there are some differences between a transaction-oriented BMP and an MPP, they are similar. A transaction-oriented BMP has the same recovery and sync point considerations as an MPP does, and a transaction-oriented BMP can do much of the same kinds of processing that an MPP can. For example, like an MPP, a transaction-oriented BMP can:

- Process input from an MPP or a terminal by issuing calls for the messages to the message queue.
- Send output to a terminal; for example, the result of one update or processing totals can be sent to a terminal.
- Send a transaction to the message queue for another BMP or an MPP to process

Some of the differences between transaction-oriented BMPs and MPPs are:

- A transaction-oriented BMP is started with JCL; an MPP is normally scheduled by IMS/VS.
- A transaction-oriented BMP can process GSAM data bases; an MPP cannot.
- A transaction-oriented BMP can process OS/VS files; an MPP cannot.

A typical use of a transaction-oriented BMP is reading and processing of transactions that were sent to the message queue by an MPP. For an example of a transaction-oriented BMP, see the sample program in "Appendix B. Sample Batch Message Program."

Having a BMP process transactions from an MPP can be more efficient than having the MPP process them if the MPP's response time is very important; but this method does split the transaction into two parts when it doesn't necessarily have to be split. If you do this, you should run the BMP during periods of the day when the system load is low. Using a BMP to process transactions from

an MPP is a good idea when the response time for the MPP is critical.

If you have a BMP perform data base updates that were generated by an MPP, you should design the BMP such that the message can be reentered as input to the BMP. This is a recovery consideration. For example, suppose an MPP gathers data base updates for three BMPs to process, and one of the BMPs terminates abnormally. In this kind of situation, you need to be able to find out what message the terminated BMP was processing and have one of the BMPs reprocess it.

BATCH PROCESSING ONLINE

There are two types of programs that are designed to perform batch processing online:

- A **batch-oriented BMP** is an IMS/VS batch application program that runs online. It does not communicate with other terminals or programs; it processes online data bases.
- A **nonmessage-driven Fast Path program** processes Fast Path data bases and is similar to a batch-oriented BMP. It doesn't normally communicate with other programs or terminals, although it can when in mixed-mode.

Batch-Oriented BMPs

In a batch-oriented BMP, the only sync points are the checkpoint calls that the program issues. Establishing sync points by issuing checkpoint calls in a BMP is important because the checkpoint calls:

- Establish places in the program from which the program could be restarted in the event of a system failure or abnormal termination
- Release resources that IMS/VS has enqueued for the program

If a batch-oriented BMP doesn't issue checkpoint calls frequently enough, it can be abnormally terminated (or it can cause another application program to be abnormally terminated) by IMS/VS for one of the following reasons:

- The space needed to enqueue information about the segments that the program has read and updated exceeds what has been defined for the IMS/VS system at system definition.
- As a result of the number of segments that the program has enqueued, other programs are having to wait excessively for those segments, because the program has not issued a checkpoint call to free its enqueued segments.
- The dynamic log is out of space because too much time has elapsed since the program's last sync point. IMS/VS records the "before" image of each segment that a BMP updates on the dynamic log. When the BMP reaches a sync point, the space that has been used by the BMP's before images is freed for other programs' use. If a BMP goes for too long without issuing a checkpoint call, the dynamic log can reach a wraparound point. When this happens, the next time that IMS/VS tries to write on the dynamic log, IMS/VS sends a warning message to the master terminal operator. The master terminal operator might then have to abnormally terminate the program whose "before" images were being written on the dynamic log. The program that's abnormally terminated could be the BMP that caused the wraparound, or it could be another application program.

Recovery in a batch-oriented BMP is similar to recovery in an MPP: if the program terminates abnormally, or if the system fails,

IMS/VS backs out the data base updates that the program has made since the last sync point. You then have to restart the program with JCL. If the BMP processes OS/VS files, it is your responsibility to back out the updates made to OS/VS files since the program's last sync point. There is a call that the program can issue to make this easier—this is the basic checkpoint call with the OS/VS option. This call establishes sync points from which you can subsequently restart the program. The call has the option of letting you request an OS/VS checkpoint as well. If you use this option, OS/VS repositions OS/VS files, and you can use OS/VS restart to restart your program. A disadvantage of using basic checkpoint with the OS/VS option is that you cannot change your program between the time that it terminates abnormally and the time you want to restart it. "Choosing a Checkpoint Call" describes this call in more detail.

Batch message processing answers an application requirement that can't be answered by either message processing or offline batch processing: it updates the data bases online and keeps the data more current than is practical with offline batch processing. To use batch message processing most effectively, however, avoid a large amount of batch-type processing online. If the BMP performs long-running processing such as report-writing and data base scans, schedule it during nonpeak hours of processing so that it doesn't degrade response time for MPPs. On the other hand, you don't have to keep your DB/DC system up only to run a batch-oriented BMP; you can run a batch-oriented BMP as a batch program offline. If you do run batch-oriented BMPs as batch programs, they cannot run concurrently. Batch-oriented BMPs can run concurrently only online.

Note: Batch message processing can answer requirements at your installation that aren't answered by other types of processing. At the same time, BMPs can have a negative effect on the response time of MPPs. Because of this, the response time requirements at your installation should be the main consideration in deciding on the extent to which you'll use batch message processing.

Nonmessage-Driven Fast Path Programs

A nonmessage-driven Fast Path program is similar to a batch-oriented BMP. It cannot access the message queue. The only sync points in a nonmessage-driven Fast Path program occur when the program issues checkpoint calls or sync calls. Sync calls are used only in nonmessage-driven Fast Path programs.

Recovery in a nonmessage-driven Fast Path program is like recovery in message-driven Fast Path programs. Nonmessage-driven Fast Path programs cannot use GSAM. You can design nonmessage-driven Fast Path programs to process OS/VS files; if you do, however, it is your responsibility to back out the updates made to OS/VS files in the event of a program or system failure.

MIXING FAST PATH AND IMS/VS PROCESSING

The online processing that has been described so far in this chapter is either Fast Path or IMS/VS processing. You can mix Fast Path and IMS/VS processing; this is called mixed mode. Here's what you can do in mixed mode:

- Fast Path programs can access IMS/VS data bases.
- IMS/VS programs can access Fast Path data bases.
- Fast Path programs can send messages to IMS/VS programs and terminals.

The following restrictions apply to mixed mode processing:

- IMS/VS programs can't send messages to Fast Path programs. unless you have Intersystem Communication with MSC.

- MPPs cannot pass conversations to a Fast Path program.
- Offline batch programs cannot process Fast Path data bases.
- Fast Path programs cannot enter commands.

BATCH PROCESSING

There is only one type of application program that does not run online; this is a batch program. Because a batch program runs by itself—it doesn't compete with any other programs for IMS/VS resources—it can run independently of the control region. It's possible to run a DB/DC system and a batch system at the same time; they just can't access the same data bases. Batch programs usually run at night or on weekends; they are typically longer-running programs than online programs.

For example, you use batch programs when you have a large number of data base updates to do, a report to print, or a lot of data to gather. For example, the current roster task described earlier in this chapter could be efficiently run as a batch program. You could run this program once a week to print the current roster for a particular class.

The reason that you do this type of processing when the rest of the system is not being used is so that you don't tie up system resources when there are a lot of people at terminals or a lot of other programs trying to use the computer's resources. Some of the requirements that batch programs answer are:

- The application program produces a large amount of output.
- There is no need for the program to be invoked by another program or user.
- The program is a long-running program that can reasonably be run at night or on the weekend (or anytime that the online system isn't running) so that it doesn't tie up system resources.
- Turnaround time for the program's output is not crucial; you don't need the processing results right away.

SYNC POINTS IN A BATCH PROGRAM

The only sync points in a batch program are the checkpoint calls that the program issues. There are two reasons for a batch program to issue checkpoint calls:

- Checkpoints establish points from which the program can be restarted. This is especially important in long-running batch programs, because the time and cost involved in reprocessing a long batch program can be significant.
- A batch program that issues checkpoint calls can be run online as a BMP. For example, the sample program in "Appendix A. Sample Batch Program" could be run as a batch-oriented BMP or as a batch program.

Where and how often the program issues checkpoint calls depend on the program's length and the type of processing it does.

RECOVERY IN A BATCH PROGRAM

When an online program terminates abnormally, IMS/VS takes care of most of the recovery by restoring the data to its values before the terminating program updated it. In a batch program, however, IMS/VS doesn't do this. IMS/VS applies the updates from a batch program as the program makes the updates. If the program terminates abnormally, you have to run the IMS/VS Data Base

Backout utility to restore the data base, or you can restore an earlier copy of the data base by using the IMS/VS Data Base Image Copy utility and the IMS/VS Data Base Recovery utility. Then, unless the program has requested OS/VS checkpoints, you can fix the program error, restart the program from the most recent checkpoint, and finish executing the program. If the program has requested OS/VS checkpoints, however, you cannot fix the program between the time it terminates abnormally and the time you restart it.

SUMMARIZING IMS/VS APPLICATION PROGRAM CHARACTERISTICS

Figure 23 is a summary of the characteristics of the six types of IMS/VS application programs. A "Y" in a column indicates a yes answer to the question in that column; an "N" in a column indicates a no answer to the question in that column.

Can this type of program:	MPP	Fast Path		BMP		Batch
		Message-Driven	Nonmessage-Driven	Batch-Oriented	Transaction-Oriented	
Access online IMS/VS data bases?	Y	Y	Y	Y	Y	N
Access offline DL/I data bases?	N	N	N	N	N	Y
Access Fast Path data bases?	Y	Y	Y	Y	Y	N
Access GSAM data bases?	N	N	N	Y	Y	Y
Access OS/VS files?	N	N	Y	Y	Y	Y
Process input messages?	Y	Y	N	N	Y	N
Send output to the message queue?	Y	Y	N	Y	Y	N

Figure 23. Summary of IMS/VS Application Program Characteristics

CHAPTER 4. GATHERING REQUIREMENTS FOR DATA BASE OPTIONS

This chapter guides you in gathering information that the DBA will use in designing a data base and implementing that design. After designing hierarchies for the data bases that your application will access, the DBA evaluates data base options in terms of which options will best meet application requirements. Whether or not these options are used depends on the collected requirements of the applications. In order to design an efficient data base, the DBA needs information about the individual applications. This chapter describes the type of information that can be helpful to the DBA, and how the information you're gathering relates to different data base options.

This chapter describes the different aspects of your application that you need to examine for each DL/I option. These are the tasks involved in this process:

- **Analyzing Data Access**

  Look at the way that the programs in your application will access the data. Will the data access be mostly direct? Mostly sequential? Or a combination? The answers to these questions and more detailed ones influence the choice of access methods for DL/I data bases.

- **Understanding How Data Structure Conflicts Are Resolved**

  Look at the data base hierarchy that the DBA has designed and compare the ways in which your application programs use the data. Are there conflicts between the way one program needs the data to be organized and the way another needs it organized? There are three DL/I options that can change the data base structure for a particular application program. These are field level sensitivity, secondary indexing, and logical relationships.

- **Identifying Security Requirements**

  Identify any security requirements involved in the data that your application uses and be aware of the DL/I security mechanisms that can be used in each case. These are processing options in the DB PCB, segment sensitivity, and field level sensitivity.

- **Identifying Recovery Requirements**

  Look at the recovery requirements and options available for each program. Specifying checkpoint type and frequency are high level design decisions that must be made individually for each application program.

ANALYZING DATA ACCESS

The DBA chooses an access method for a DL/I data base based on how the majority of programs that use the data base will access the data. There are four basic DL/I access methods, and three access methods that are used in special cases. Some of the information that you can gather to help the DBA in this decision is answers to questions like the following:

- To access a data base record, a program must first access the root of the record. How will each program access root segments?
  - Directly
  - Sequentially

- Both

- The segments within the data base record are the dependents of the root segment. How will each program access the segments within each data base record?

  - Directly
  - Sequentially
  - Both

  It's important to note the distinction between accessing a data base record and accessing segments within the record. A program could access data base records sequentially, but once within a record, the program might access the segments directly. These are two different requirements, and can influence the choice of access method.

- To what extent will the program update the data base?

  - Will the program be adding new data base records?
  - Will the program be adding new segments to existing data base records?
  - Will the program be deleting segments or data base records?

  Again, don't ignore the distinction between updating a data base record and updating a segment within the data base record. They are different requirements.

DIRECT ACCESS

The advantage of direct access processing is that you can get good results for both direct and sequential processing. Direct access means that by using a randomizing routine or an index, DL/I can find any data base record that you want; access is not based on the sequence of data base records in the data base.

DL/I has two direct access methods.

- The Hierarchical Direct Access Method, HDAM, processes data directly by using a randomizing routine to store and locate root segments.
- The Hierarchical Indexed Direct Access Method, HIDAM, has an index to help it provide direct processing of root segments.

The direct access methods use pointers to maintain the hierarchic relationships between segments of a data base record. By following pointers, DL/I can access a path of segments without passing through all of the segments in the preceding paths.

Some of the requirements that direct accessing satisfies are:

- Fast direct processing of roots using an index or a randomizing routine
- Good sequential processing of data base records with HIDAM using the index
- Fast access to a path of segments via pointers

In addition, when you delete data from a direct access data base, the new space is available almost immediately. This gives you efficient space utilization, and it means that you don't have to reorganize the data base often because of unused space. Direct access methods maintain their own pointers and addresses internally.

A disadvantage with direct access is that you have a larger IMS/VS overhead because of the pointers. If direct access answers your data access requirements, however, it is more efficient than using a sequential access method.

Primarily Direct Processing: HDAM

HDAM is efficient for a data base that will have primarily direct access, but with some sequential accessing. HDAM uses a randomizing routine to locate its root segments, then chains dependent segments together in their hierarchic paths. The OS access methods that HDAM can use are VSAM and OSAM.

The requirements that HDAM satisfies are:

- Direct access of roots by root keys—HDAM uses a randomizing routine to locate root segments
- Direct access of paths of dependents
- Adding new data base records and new segments—the new data goes into the nearest available space
- Deleting data base records and segments—the space created by a deletion can be used by any new segment

HDAM CHARACTERISTICS: For root segments in an HDAM data base, DL/I:

- Can store them anywhere; in other words they don't have to be in sequence because the randomizing routine locates them.
- Uses a randomizer to locate the relative block number and root anchor point, or RAP, within the block that points to the root segment.
- Returns root segments in physical sequence, not key sequence, if you retrieve root segments sequentially.

For dependent segments, an HDAM data base:

- Can store them anywhere
- Chains all segments of one data base record together with pointers

AN OVERVIEW OF HOW HDAM WORKS: When a data base record is stored in an HDAM data base, HDAM keeps one or more RAPs at the beginning of each physical block. The RAP points to a root segment. HDAM also keeps a pointer at the beginning of each physical block that points to any free space in the block. When you insert a segment, HDAM uses this pointer to locate free space in the physical block. To locate a root segment in an HDAM data base, you give HDAM the root key. The randomizing routine gives it the relative physical block number and the RAP that points to the root segment. The RAP number specified gives HDAM the location of the root within a physical block.

Although HDAM can place root and dependents anywhere in the data base, it's better to choose HDAM options that keep roots and dependents close together.

HDAM performance depends largely on the randomizing routine you use. Performance can be very good. Performance also depends on other implementation factors such as:

- The block size you use.
- The number of RAPs per block.
- The pattern for chaining together different segments. You can chain segments of a data base record in two ways:

- In hierarchic sequence starting with the root.
- Parents can contain pointers to each of their paths of dependents.

HDAM is not good for sequential access of data base records by root key unless you use a randomizer that stores roots in physical key sequence or a secondary index.

Direct and Sequential Processing: HIDAM

HIDAM is the access method that is most efficient for an approximately equal amount of direct and sequential processing. The OS access methods it can use are VSAM and ISAM/OSAM. The specific requirements that it satisfies are:

- Direct and sequential access of records by their root keys
- Direct access of paths of dependents
- Adding new data base records and new segments—the new data goes into the nearest available space
- Deleting data base records and segments—the space created by a deletion can be used by any new segment

HIDAM can answer most processing requirements that involve an even mixture of direct and sequential processing; a situation in which it's not very efficient is sequential access of dependents.

HIDAM CHARACTERISTICS: For root segments, a HIDAM data base:

- Initially loads them in key sequence
- Can store new root segments wherever there's space
- Uses an index to locate a root that you request and identify by supplying the root's key value

For dependent segments, a HIDAM data base:

- Can store segments anywhere, preferably fairly close together
- Chains all segments of a data base record together with pointers

AN OVERVIEW OF HOW HIDAM WORKS: HIDAM uses two data bases: one, the primary data base, holds the data, and the other is an index data base. The index data base contains entries for all of the root segments in order of their key fields. For each key entry, the index data base contains the address of that root segment in the primary data base.

When you access a root, you supply the key to the root. HIDAM looks the key up in the index to find the address of the root, then goes to the primary data base to find the root.

HIDAM chains dependent segments together so when you access a dependent segment, HIDAM uses the pointer in one segment to locate the next segment in the hierarchy.

When you process data base records directly, HIDAM locates the root through the index, then locates the segments from the root. HIDAM locates dependents through pointers.

If you are going to process data base records sequentially, you can specify some special pointers in the DBD for the data base so that DL/I doesn't have to go to the index to locate the next root segment. These pointers chain the roots together. If you don't chain roots together, HIDAM always goes to the index to locate a root segment. When you process data base records sequentially, HIDAM accesses roots in key sequence in the index. This only

applies to sequential processing; if you want to access a root segment directly, HIDAM uses the index, and not pointers, to find the root segment you've requested.

SEQUENTIAL ACCESS

When you use a sequential access method, the segments in the data base are stored in hierarchic sequence, one after another. There are no pointers in a sequential data base.

DL/I has two sequential access methods. Like the direct access methods, one has an index and the other doesn't:

- The Hierarchical Sequential Access Method, HSAM, only processes root segments and dependent segments sequentially.
- The Hierarchical Indexed Sequential Access Method, HISAM, processes data sequentially but has an index so that you can access records directly. HISAM is primarily for sequentially processing dependents, and directly processing data base records.

Some of the general requirements that sequential accessing satisfies are:

- Fast sequential processing
- Direct processing of data base records with HISAM
- Small IMS/VS overhead on storage because sequential access methods relate segments by adjacency rather than with pointers

There are three disadvantages to sequential access methods: first, sequential access methods give slower access to the rightmost segments in the hierarchy, because HSAM and HISAM have to read through all of the other segments to get to them. Second, HISAM requires frequent reorganization to reclaim space from deleted segments and to keep the logical records of a data base record physically adjoined. And third, you can't update HSAM data bases; you have to create a new data base to change any of the data.

Sequential Processing Only: HSAM

HSAM is a hierarchical access method that can handle only sequential processing. You can retrieve data from HSAM data bases, but you can't update any of the data. The OS access methods that HSAM can use are QSAM and BSAM.

HSAM is good for situations in which:

- You are storing historical data that you will not need to update.
- You are using the data base to just collect data or statistics, but will not need to update it.
- You will be processing the data only sequentially.

HSAM CHARACTERISTICS: HSAM stores data base records in the sequence in which you submit them. You can only process records and dependent segments sequentially; "sequentially" means in the order in which you've loaded them. HSAM stores dependent segments in hierarchic sequence.

AN OVERVIEW OF HOW HSAM WORKS: HSAM data bases are very simple data bases. The data is stored in hierarchic sequence, one segment after the other. There are no pointers, no indexes, just the HSAM data base.

Primarily Sequential Processing: HISAM

HISAM is an access method that stores segments in hierarchic sequence with an index to locate root segments. It also has an overflow data set; you store segments in a logical record until you reach the end of the logical record. When you run out of space on the logical record, but you still have more segments belonging to the data base record, you store the remaining segments in an overflow data set. The OS access methods that HISAM can use are VSAM and ISAM/OSAM.

HISAM is well-suited for:

- Direct access of record by root keys
- Sequential access of records
- Sequential access of dependent segments

There are situations in which your processing has some of the characteristics above but where HISAM is not necessarily a good choice. These situations are when:

- You have to access dependents directly.
- You have a high number of inserts and deletes.
- A lot of the data base records exceed average size and have to use the overflow data set. This is because the segments that overflow into the overflow data set require additional I/O.

HISAM CHARACTERISTICS: For data base records, HISAM data bases:

- Store records in key sequence
- Can locate a particular record with a key value by using the index

For dependent segments, HISAM data bases:

- Start each HISAM data base record in a new logical record in the primary data set
- Store the remaining segments in one or more logical records in the overflow data set if the data base record won't fit in the primary data set

AN OVERVIEW OF HOW HISAM WORKS: The reason that HISAM is not well-suited for a lot of inserting and deleting segments is that, unlike HIDAM and HDAM, HISAM doesn't reuse space right away. HISAM data bases have to repack to use extra space to make room for new segments. HISAM space is reclaimed when you reorganize a HISAM data base. HISAM data bases have to shift data around when you insert a new segment to make room for the new segment, and they leave unused space after deletions.

ACCESSING OS/VS FILES THROUGH IMS/VS: GSAM

There is one additional access method that DL/I programs can use: this is the Generalized Sequential Access Method, or GSAM. GSAM makes it possible for a sequential OS/VS data set to be handled by IMS/VS as a simple data base. The OS access methods that GSAM can use are BSAM and VSAM.

A GSAM data base:

- Is an OS/VS data set record defined as a data base record. The record is handled as one unit; it contains no segments or fields and the structure is not hierarchic.
- Can be accessed by OS/VS or by IMS/VS.

One common use of GSAM is as input to batch-oriented BMPs or batch programs. Batch-oriented BMPs and batch programs can also send output to GSAM data bases. To process a GSAM data base, an application program issues calls similar to the ones it issues to process a DL/I data base. The program can read data sequentially from a GSAM data base, and it can send output to a GSAM data base. GSAM is a sequential access method; you can only add records to an output data base sequentially. For an example of a program that uses GSAM, see the sample batch program in "Appendix A. Sample Batch Program." This program reads its input from a GSAM data base and sends its output to another GSAM data base.

ACCESSING IMS/VS DATA THROUGH OS/VS: SHSAM AND SHISAM

There are two data base access methods that give you simple hierarchical data bases that OS/VS can use as data sets. These access methods can be particularly helpful when you're converting data from OS/VS files to an IMS/VS data base. Again, one is indexed and one is not:

- The Simple Hierarchical Sequential Access Method, SHSAM
- The Simple Hierarchical Indexed Sequential Access Method, SHISAM

When you use these access methods, you define an entire data base record as one segment. The segment does not contain any DL/I control information or pointers; the data format is the same as it is in OS/VS data sets. The OS access methods that SHSAM can use are BSAM and QSAM. SHISAM uses VSAM.

What makes SHSAM and SHISAM data bases useful during transitions is that they can be accessed by OS/VS access methods without IMS/VS.

UNDERSTANDING HOW DATA STRUCTURE CONFLICTS ARE RESOLVED

The order in which application programs need to process fields and segments within hierarchies is frequently not the same for each. When the DBA finds a conflict in the way that two or more programs need to access the data base, DL/I has three options that can solve these problems. Each option solves a different kind of conflict. These options and the kinds of conflicts they solve are:

- When an application program doesn't need access to all of the fields in a segment, or if the program needs to access them in a different order, the DBA can use field level sensitivity for that program. Field level sensitivity makes it possible for an application program to access only a subset of the fields that a segment contains; or for an application program to process a segment's fields in an order that is different from their order in the segment.
- When an application program needs to access a particular segment by a field other than the segment's key field, the DBA can use a secondary index for that data base.
- When the application program needs to relate segments from different hierarchies, the DBA can use logical relationships. Using logical relationships can give the application program a logical hierarchy that includes segments from several hierarchies.

USING DIFFERENT FIELDS: FIELD LEVEL SENSITIVITY

Field level sensitivity means the same kind of security for fields within a segment that segment sensitivity does for segments within a hierarchy: an application program can access only those fields within a segment, and those segments within a hierarchy to which it is sensitive.

Field level sensitivity also makes it possible for an application program to use a subset of the fields that make up a segment, or to use all of the fields in the segment but in a different order. If there are fields within a segment that the application program doesn't need to process, using field level sensitivity means that the program doesn't have to process those fields that it doesn't use.

AN EXAMPLE OF FIELD LEVEL SENSITIVITY: For example, suppose that a segment containing data about an employee contains the fields shown in Figure 24. These fields are:

- Employee number: EMPNO
- Employee name: EMPNAME
- Birthdate: BIRTHDAY
- Salary: SALARY
- Address: ADDRESS

Figure 24. Physical Employee Segment

A program that printed mailing labels for employees' checks each week would not need all the data in the segment. If the DBA decided to use field level senstivity for that application, the program would receive only the fields it needed in its I/O area. Figure 25 shows what the program's I/O area would contain.

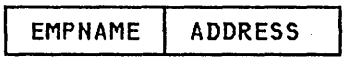

Figure 25. Employee Segment with Field Level Sensitivity

Field level sensitivity makes it possible for a program to receive a subset of the fields that make up a segment, the same fields but in a different order, or both. Another situation in which field level senstivity is very useful is when new uses of the data base involve adding new fields of data to an existing segment. In this situation, you want to avoid recoding programs that use the current segment.

By using field level sensitivity, the old programs can see only the fields that were in the original segment. The new program can see the old and the new fields.

WHERE FIELD LEVEL SENSITIVITY IS SPECIFIED: You specify field level sensitivity in the PSB for the application program by using a sensitive field (SENFLD) statement for each field to which you want the application program to be sensitive.

RESOLVING PROCESSING CONFLICTS IN A HIERARCHY: SECONDARY INDEXING

Sometimes a data base hierarchy doesn't meet all the processing requirements of the application programs that will process it. Secondary indexing can be used to solve two kinds of processing conflicts:

- When an application program needs to retrieve a segment in a sequence other than the one that has been defined by the segment's key field

- When an application program needs to retrieve a segment based on a condition that is found in a dependent of that segment

A way to understand these conflicts and how secondary indexing can resolve them is to look at a couple of examples. The examples explained here are two application programs that process the patient hierarchy shown in Figure 26. There are three segment types in this hierarchy:

- PATIENT contains three fields: the patient's identification number, the patient's name, and the patient's address. The patient number field is the key field.
- ILLNESS contains two fields: the date of the illness and the name of the illness. The date of the illness is the key field.
- TREATMNT contains four fields: the date the medication was given; the name of the medication; the quantity of the medication that was given; and the name of the doctor who prescribed the medication. The date that the medication was given is the key field.

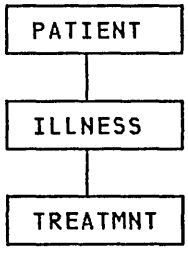

Figure 26. Patient Hierarchy

Using a Different Key

When an application program retrieves a segment from the data base, the program identifies the segment by the segment's key field. But sometimes an application program needs to retrieve a segment in a sequence other than the one that has been defined by the segment's key field. Secondary indexing makes this possible.

For example, suppose you have an online application program that processes requests as to whether or not an individual has ever been to the clinic. If you're not sure whether or not the person has ever been to the clinic, then you won't be able to supply the identification number for the person. But the key field of the PATIENT segment is the patient's identification number.

Segment occurrences of a segment type (for example, the segments for each of the patients) are stored in a data base in order of their keys (in this case, by their patient identification numbers). If you issue a request for a PATIENT segment and identify the segment you want by the patient's name instead of the patient's identification number, DL/I might have to search through all of the PATIENT segments to find the PATIENT segment you've requested. DL/I doesn't know where a particular PATIENT segment is just by having the patient's name.

To make it possible for this application program to retrieve PATIENT segments in the sequence of patients' names (rather than in the sequence of patients' identification numbers), you can index the PATIENT segment on the patient name field and store the index entries in a separate data base. The separate data base is called a secondary index.

Then, if you indicate to DL/I that DL/I is to process the PATIENT segments in the patient hierarchy in the sequence of the index

entries in the secondary index data base, DL/I can locate a PATIENT segment if you supply the patient's name. DL/I goes directly to the secondary index and locates the PATIENT index entry with the name you've supplied; the PATIENT index entries are in order of the patient names. The index entry is a pointer to the PATIENT segment in the patient hierarchy. DL/I can determine whether or not a PATIENT segment for the name you've supplied exists, and return the segment to the application program if the segment exists. If the requested segment doesn't exist, DL/I indicates this to the application program by returning a "not-found" status code.

There are three terms involved in secondary indexing that you should know.

- The **pointer segment** is the index entry in the secondary data base that DL/I uses to find the segment you've requested. In the example above, the pointer segment is the index entry in the secondary index data base that points to the PATIENT segment in the patient hierarchy.
- The **source segment** is the segment that contains the field that you're indexing. In the example above, the source segment is the PATIENT segment in the patient hierarchy, since you're indexing on the name field in the PATIENT segment.
- The **target segment** is the segment in the data base that you're processing that the secondary index points to; it's the segment that you want to retrieve.

In the example above, the target segment and the source segment are the same segment—the PATIENT segment in the patient hierarchy. When the source segment and the target segment are different segments, secondary indexing solves the processing conflict described below.

The PATIENT segment that DL/I returns to the application program's I/O area doesn't look any different than it would if secondary indexing had not been used.

The DB PCB key feedback area is different. When DL/I retrieves a segment without using a secondary index, DL/I places the concatenated key of the retrieved segment in the DB PCB key feedback area. The concatenated key contains all the keys of the segment's parents, in order of their positions in the hierarchy. The key of the root segment is first, followed by the key of the segment on the second level in the hierarchy, then the third, and so on—with the key of the retrieved segment last. But when you retrieve a segment from an indexed data base, the contents of the key feedback area after the call are a little different. Instead of placing the key of the root segment in the left-most bytes of the key feedback area, DL/I places the key of the pointer segment. For example, suppose index segment A shown in Figure 27 on a field in segment C. Segment A is the target segment, and segment C is the source segment.

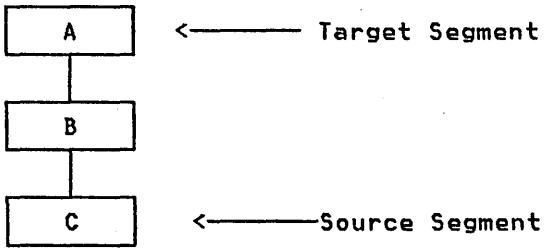

Figure 27. Indexing a Root Segment

When you use the secondary index to retrieve one of the segments in this hierarchy, the key feedback area contains one of the following:

- If you retrieve segment A, the key feedback area contains the key of the pointer segment from the secondary index.
- If you retrieve segment B, the key feedback area contains the key of the pointer segment concatenated with the key of segment B
- If you retrieve segment C, the key of the pointer segment, the key of segment B, and the key of segment C are concatenated in the key feedback area.

Although this example creates a secondary index for the root segment, you can index dependent segments as well. If you do this, you create an inverted structure: the segment you index becomes the root segment, and its parent becomes a dependent. For example, suppose you index segment B on a field in segment C. In this case, segment B is the target segment, and segment C is the source field. Figure 28 shows the physical data base structure and the structure created by the secondary index.

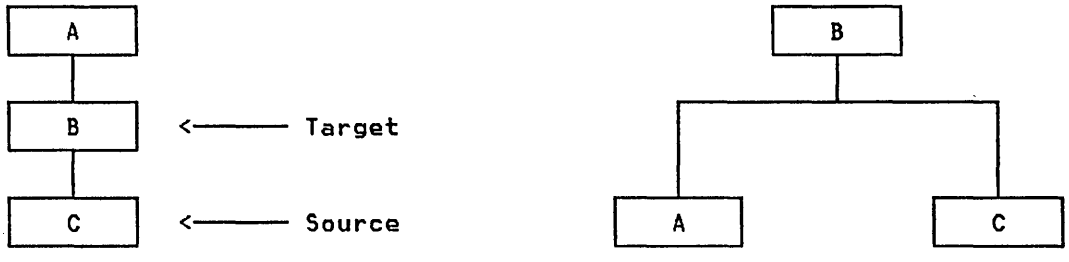

Figure 28. Indexing a Dependent Segment

When you retrieve the segments in this secondary index data structure on the right, DL/I returns the following to the DB PCB key feedback area:

- If you retrieve segment B, the key feedback area contains the key of the pointer segment in the secondary index data base.
- If you retrieve segment A, the key feedback area contains the key of the pointer segment concatenated with the key of segment A.
- If you retrieve segment C, the key feedback area contains the key of the pointer segment concatenated with the key of segment C.

Retrieving Segments Based on a Dependent's Qualification

Sometimes an application program needs to retrieve a segment, but only if one of the segment's dependents meets a certain qualification. For example, suppose that the medical clinic used in the example above wants to print a monthly report of the patients who have visited the clinic during that month. If the application program that processes this request doesn't use a secondary index, the program has to retrieve each PATIENT segment, then retrieve the ILLNESS segment for each PATIENT segment. The program tests the date in the ILLNESS segment to determine whether or not the patient has visited the clinic during the current month, and prints the patient's name if the answer is yes. The program continues retrieving PATIENT segments and ILLNESS segments until it has retrieved all the PATIENT segments.

But with a secondary index, you can make the processing that the program has to do much simpler. To do this, you index the PATIENT segment on the date field in the ILLNESS segment. When you define the PATIENT segment in the DBD, you give DL/I the name of the field that you're indexing the PATIENT segment on, and the name of the segment that contains the index field. The application program can then issue a call to DL/I for a PATIENT segment and qualify the call with the date in the ILLNESS segment. The PATIENT segment that DL/I returns to the application program looks just as it would if you were not using a secondary index.

In this example, the PATIENT segment is the target segment; it's the segment that you want to retrieve. The ILLNESS segment is the source segment; it contains the information that you want to use to qualify your request for PATIENT segments. The index segment in the secondary data base is the pointer segment. It points to the PATIENT segments.

CREATING A NEW HIERARCHY: LOGICAL RELATIONSHIPS

When an application program needs to associate segments from different hierarchies, logical relationships can make it possible for that program to do so. A couple of the conflicts that logical relationships can solve are:

- When two application programs need to process the same segment—but they need to access the segment through different hierarchies
- When a segment's parent in one application program's hierarchy acts as that segment's child in another application program

Accessing a Segment through Different Paths

Sometimes an application program needs to process the data in a different order than the way it's arranged in the hierarchy. For example, an application program that processes data in a purchasing data base also requires access to a segment in a patient data base:

- Program A processes information in the patient data base about the patients at a medical clinic: the patients' illnesses and their treatments.
- Program B is an inventory program that processes information in the purchasing data base about the medications that the clinic uses: the item, the vendor, information about each shipment, and information about when and under what circumstances each medication was given.

Figure 29 shows the hierarchies that Program A and Program B require for their processing. There is a conflict in their processing requirements: they both need to have access to the information that's contained in the TREATMNT segment in the patient data base. This information is:

- The date that a particular medication was given
- The name of the medication
- The quantity of the medication given
- The doctor that prescribed the medication

To Program B this isn't information about a patient's treatment; it's information about the disbursement of a medication. To the purchasing data base, this is the disbursement segment (DISBURSE).

Figure 29 shows the hierarchies for Program A and Program B. Program A needs the PATIENT segment, the ILLNESS segment, and the TREATMNT segment. Program B needs the ITEM segment, the VENDOR segment, the SHIPMENT segment, and the DISBURSE segment. The TREATMNT segment and the DISBURSE segment contain the same information.

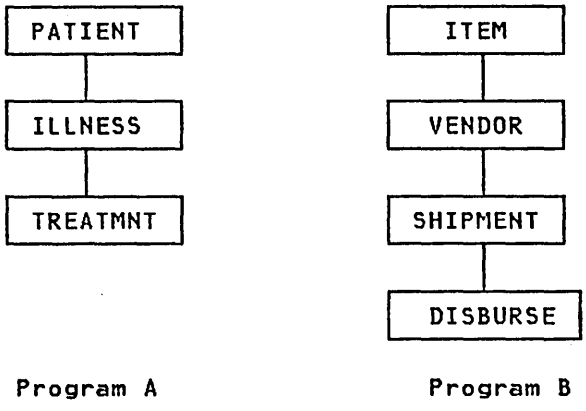

Figure 29. Patient and Inventory Hierarchies

Instead of storing this information in both hierarchies, you can use a logical relationship. A logical relationship solves the problem by storing a pointer from where the segment is needed in one hierarchy to where the segment exists in the other hierarchy. In this case, you can have a pointer in the DISBURSE segment to the TREATMNT segment in the medical data base. When DL/I receives a request for information in a DISBURSE segment in the purchasing data base, DL/I goes to the TREATMNT segment in the medical data base pointed to by the DISBURSE segment. Figure 30 shows the physical hierarchy that Program A would process and the logical hierarchy that Program B would process. DISBURSE is a pointer segment to the TREATMNT segment in Program A's hierarchy.

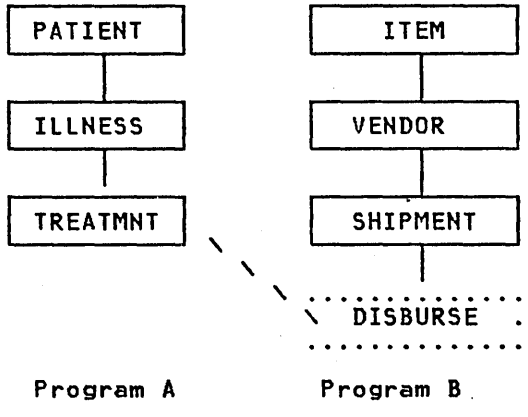

Figure 30. Logical Relationships Example

To define a logical relationship between segments in different hierarchies, you use a logical DBD. A logical DBD defines a hierarchy that does not exist in storage, but can be processed as though it does. Program B would use the logical structure shown in Figure 30 as though it were a physical structure.

Inverting a Parent/Child Relationship

Another type of conflict that logical relationships can resolve occurs when a segment's parent in one application program acts as that segment's child in another application program:

- The inventory program (Program B above) needs to process information about medications using the medication as the root segment.
- A purchasing application program, Program C, processes information about which vendors have sold which medications. Program C needs to process this information using the vendor as the root segment.

Figure 31 shows the hierarchies for each of these application programs.

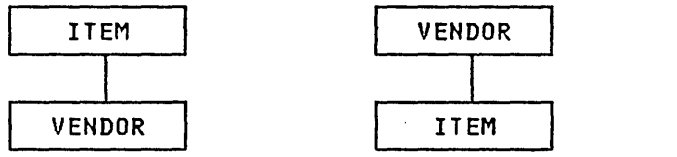

Figure 31. Supplies and Purchasing Hierarchies

Logical relationships can solve this problem by using pointers. Using pointers in this example would mean that the ITEM segment in the purchasing data base would contain a pointer to the actual data stored in the ITEM segment in the supplies data base. The VENDOR segment, on the other hand, would actually be stored in the purchasing data base. The VENDOR segment in the supplies data base would point to the VENDOR segment stored in the purchasing data base.

Figure 32 shows the hierarchies these two programs.

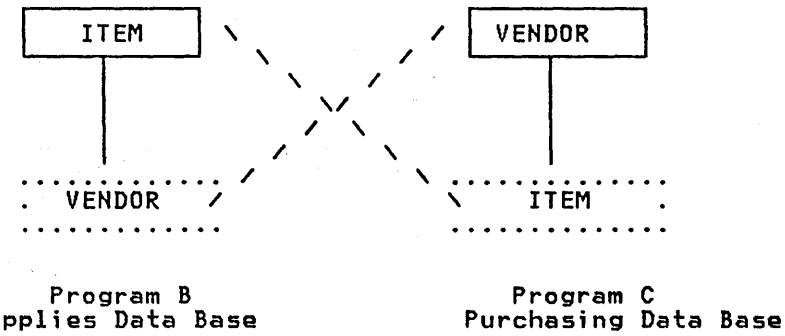

Figure 32. Program B and Program C Hierarchies

If you didn't use logical relationships in this situation, you would:

- Keep the same data in both paths, which means you would be keeping redundant data

- Have the same disadvantages as separate files of data:
  - You would have to update multiple segments each time one piece of data changed.
  - You would need more storage.

IDENTIFYING SECURITY REQUIREMENTS

If you find that some of the data in your application has a security requirement, DL/I can provide security for that data in two ways:

- **Data sensitivity** is a way to control what data a particular program can access.
- **Processing options** are a way to control how a particular program can process that data.

KEEPING A PROGRAM FROM ACCESSING THE DATA: DATA SENSITIVITY

A DL/I program can only access data to which it is sensitive. You can control the data to which your program is sensitive on three levels:

- **Segment sensitivity** can prevent an application program from accessing all of the segments in a particular hierarchy. It's how you tell DL/I which segments in a hierarchy the program is allowed to access.
- **Field level sensitivity** can keep a program from accessing all of the fields that make up a particular segment. Field level sensitivity tells DL/I which fields within a particular segment a program is allowed to access.
- **Key sensitivity** means that the program can access segments below a particular segment, but it cannot access the particular segment. DL/I will return only the key of this type of segment to the program.

You define each of these levels of sensitivity in the PSB for the application program. Key sensitivity is defined in the processing option for the segment. Processing options are how you indicate to DL/I exactly what a particular program may or may not do to the data. You specify a processing option for each hierarchy that the application program processes; you do this in the DB PCB that represents each hierarchy. You can specify one processing option for all the segments in the hierarchy, or you can specify different processing options for different segments within the hierarchy.

Segment sensitivity and field level sensitivity are defined using special statements in the PSB.

Segment Sensitivity

You define what segments an application program is sensitive to in the DB PCB for the hierarchy that contains those segments. For example, suppose that the patient hierarchy shown in Figure 26 belongs to the medical data base shown in Figure 33. The patient hierarchy is like a subset of the medical data base.

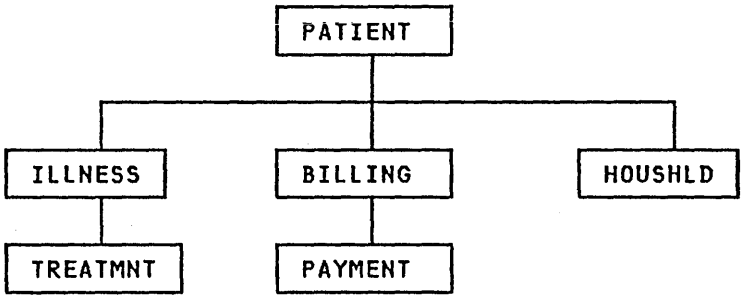

Figure 33. Medical Data Base Hierarchy

To make it possible for an application program to view only the segments PATIENT, ILLNESS, and TREATMNT from the medical data base, you specify in the DB PCB that the hierarchy you're defining has these three segment types, and that they are from the medical data base. You define the data base hierarchy in the DBD; you define the application program's view of the data base hierarchy in the DB PCB.

Field Level Sensitivity

In addition to providing data independence for an application program, field level sensitivity can also act as a security mechanism for the data that the program uses.

If a program needs to access some of the fields in a segment, but one or two of the fields that the program doesn't need to access are confidential, you can use field level sensitivity. If you define that segment for the application program as containing only the fields that are not confidential, you prevent the program from accessing the confidential fields. Field level sensitivity acts as a mask for the fields you wish to restrict access to.

Key Sensitivity

To access a segment, an application program must be sensitive to all segments at a higher level in the segment's path. In other words, in Figure 34, a program has to be sensitive to segment B in order to access segment C.

Suppose that an application program needs segment C to do its processing. But if segment B contains confidential information (such as an employee's salary), then the program should not be able to access that segment. Using key sensitivity lets you withhold segment B from the application program, and at the same time give the program access to segment B's dependents.

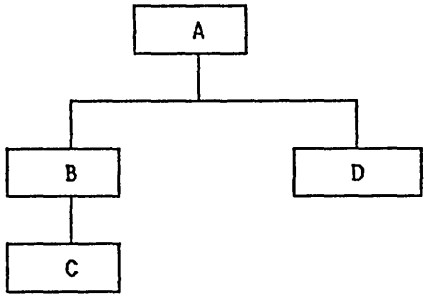

Figure 34. Sample Hierarchy

When a sensitive segment statement has a processing option of K specified for it, the program cannot access that segment, but the program can pass beyond that segment to access the segment's dependents. When the program does access the segment's dependents, DL/I doesn't return that segment; DL/I returns only the segment's key with the keys of the other segments accessed to the program's I/O area.

PREVENTING A PROGRAM FROM UPDATING DATA: PROCESSING OPTIONS

There are five processing options that you use to indicate to DL/I whether the program can just read segments in the hierarchy, or whether the program can update segments as well. These are:

G You specify G if your program can read segments.

I You specify I if your program can insert segments.

R You specify R if your program can replace segments. The R processing option includes the G processing option. In other words, DL/I assumes that if your program will be replacing segments, your program will be reading segments as well.

D You specify D if your program can delete segments. Like the R processing option, if you specify D it's as though you've specified G as well.

A Specifying A means that your program needs all the processing options above. It's the equivalent of specifying G, I, R, and D.

Processing options act as security mechanisms in that they can limit what a program can do to a particular segment or to the hierarchy. If a program doesn't need to delete segments from a data base, there's no reason to specify the D option. Specifying only those processing options that the program requires ensures that the program can't update any of the data that it's not supposed to.

You must specify a processing option for a DB PCB. If you specify a processing option for a particular segment, the processing option you specify for the segment takes precedence over the one you specify for the DB PCB. For example, if you specified G, I, and R for the DB PCB—meaning that the program could retrieve, insert, and replace segments in that hierarchy—and then specified A for a particular segment, the program would be able to delete that segment as well.

There are two additional processing options that you use only for online programs. These are E and GO.

E means that that program has exclusive use of the hierarchy or the segment you use it with. The E processing option is a security mechanism in that it overrides program isolation; other programs cannot access that data while the program with the E processing option is running.

When an application program that has any of the processing options described so far retrieves a segment, IMS/VS enqueues the segment for that application program. If the program is only retrieving the segment, the segment's available to other programs as soon as the application program moves to the next data base record. If the program is updating the segment, the segment is unavailable to other programs until the program reaches a sync point.

But when an application program with a processing option of GO retrieves a segment, IMS/VS does not enqueue the segment. The segment continues to be available to other application programs. This is because GO means that the program can only read the data; the program is not allowed to update the data base. IMS/VS doesn't enqueue segments for programs with processing options of GO. The segment remains available to other application programs while the GO program is reading it.

An application program with a processing option of GO can also retrieve a segment even if another program is updating the segment. This means that the program doesn't have to wait for segments that other programs are accessing, but it also means that it's possible for the GO program to retrieve a segment containing an invalid pointer. If an online application program terminates abnormally, IMS/VS backs out the changes that it has made to the data base since the program's last sync point. But if the updating program terminates abnormally before reaching the next sync point, the segments that the program has updated may contain invalid pointers. Because of this, it's possible for the GO application program to retrieve a segment containing a pointer that's invalid. IMS/VS protects online programs—except for those with a processing option of GO—from invalid pointers by preventing them from retrieving updated segments until the updating program reaches a sync point. But a program with the GO processing option can retrieve a segment even if IMS/VS has enqueued that segment for another application program. If the GO application program retrieves a segment that contains an invalid pointer, IMS/VS terminates the program abnormally.

To prevent a GO application program from being terminated abnormally in this situation, there are two additional processing options you can use with GO: N and T. When you use N with GO (GON), IMS/VS returns a GG status code to your program if the segment you're retrieving contains an invalid pointer. You can then decide what you want to do; for example, you might continue processing by reading a different segment. When you use T with GO (GOT), IMS/VS retries the call the program just issued; if the program that was updating the requested segment has reached a sync point since you tried to retrieve the segment, the updated segment is valid and you would be free to retrieve it. If, when IMS/VS retries the call for you, the pointer is still invalid, IMS/VS then returns a GG status code.

IDENTIFYING RECOVERY REQUIREMENTS

The degree to which you need to plan for recovery for a program depends more than anything else on the type of processing that the program does. There are three DL/I calls that are related to recovery: symbolic checkpoint, restart, and basic checkpoint.

Two of these calls take checkpoints of your program so that, if your program were to terminate abnormally, your program could be restarted from a place other than the beginning of the program. Checkpoint calls establish synchronization points throughout your program. A program can be restarted only from a synchronization point.

The third call gives you a way to restart your program. It restores your program's data areas to the way they were when the program terminated abnormally, and it indicates to DL/I the synchronization point from which the program is to be restarted.

These calls are most important to batch programs and BMPs, since, for the most part, IMS/VS handles recovery and restart for message programs. But because batch programs and BMPs are not started by IMS/VS (like message programs) and they do not have the built-in synchronization points that message programs do, batch programs and BMPs need checkpoints to establish synchronization points.

CHOOSING A CHECKPOINT CALL

There are two kinds of checkpoint calls you can use: symbolic and basic. Although the reasons you use each call and the results of each call differ, there are some similarities. Issuing either kind of call in a batch program or a batch-oriented BMP causes IMS/VS to record all modified data base buffers and to send messages to the system console and the system log describing the checkpoint.

The biggest difference between the symbolic and basic calls is the type of restart each uses. With the symbolic call, you can checkpoint areas in your program, and the program must use the restart call. The restart call restores those user areas that were checkpointed and restarts the program.

With the basic checkpoint call, on the other hand, you have the option of requesting an OS/VS checkpoint, and then using OS/VS restart to restart your program. You cannot use the OS/VS option in Fast Path programs. This is the only supplied method of restart with the basic call; if you don't use OS/VS checkpoint and restart, it's up to you to provide program restart. If an application program that updates OS/VS files terminates abnormally, it is your responsibility to back out the OS/VS updates made by the terminating application program.

Another advantage of the symbolic call concerns programs that access OS/VS files. If an application program accesses OS/VS files, you can convert those files to GSAM and use symbolic checkpoint. If you have to restart the program, the restart call automatically repositions GSAM input and output files for restart processing. If you can't convert the files to GSAM, you can use the OS/VS option on the basic call.

Because of it's easier restart, programs should use symbolic checkpoint whenever possible. To see what the calls look like and how they fit in to the rest of an application program, you can refer to the sample batch program in "Appendix A. Sample Batch Program" and the sample BMP in "Appendix B. Sample Batch Message Program." Both of these programs issue symbolic checkpoint and restart calls. Figure 35 is a summary of the differences between the symbolic and basic checkpoint calls.

	Symbolic Checkpoint	Basic Checkpoint
To restart the program	Use the restart call	Use OS/VS option and OS/VS restart; or supply your own method
To use with OS files	Convert OS files to GSAM	Specify the OS/VS option on the call
Advantages	•You restart the program simply by using the restart call. •You can change the program before you restart it.	•Provides checkpoint and restart for programs accessing OS files.

Figure 35. Summary of Symbolic and Basic Checkpoint Calls

HOW OFTEN TO USE CHECKPOINTS

Checkpoint frequency depends on the type of processing you're using.

Checkpoints in Batch Programs

Any batch program that updates data bases should issue checkpoints. The main consideration in deciding how often to checkpoint a batch program is the time required to back out and reprocess the program after a failure. A general recommendation is one checkpoint call every ten or fifteen minutes. Whatever frequency you decide on, it's a good idea to make the frequency easy to modify, in case you find that it's too high or too low. For example, a counter in the program can keep track of the number of data base updates the program has performed. When the count reaches a certain limit, the program issues a checkpoint call.

If there's a chance that you might have to back out the entire batch program, the program should issue the checkpoint call at the very beginning of the program. IMS/VS will back out the program to the checkpoint that you specify, or to the most recent checkpoint, if you don't specify a checkpoint.

Another important reason for checkpointing batch programs is that although they may currently run in a DL/I region, they might later need to access online data bases. This would require converting them to BMPs. Checkpointing a BMP is important for reasons other than recovery.

Checkpoints in Batch-Oriented BMPs

Checkpoints in batch-oriented BMPs are important for three reasons beyond recovery. These are enqueue lockout, enqueue space, and dynamic log space.

- Enqueue Lockout

  When a BMP retrieves or updates a segment occurrence, IMS/VS prevents other programs from accessing that segment occurrence until the updating BMP moves to the next data base record (if the BMP has only retrieved the segment occurrence), or reaches a sync point (if the BMP has updated the segment occurrence). If a BMP retrieves and updates many

data base records between checkpoint calls, it can tie up large portions of the data bases, causing excessive wait times for other programs trying to access the same segment occurrence. (The exception to this is a BMP with a processing option of GO. IMS/VS does not enqueue segments for programs with this processing option.) Issuing checkpoint calls releases all segment occurrences that the BMP has enqueued and makes them available to other programs.

- Enqueue Space

  Failure to issue checkpoints frequently enough can also cause a storage space problem. Because program isolation enqueues are held until a sync point is reached, the amount of storage required for enqueue records can exceed the amount that was specified for the system, and result in an abnormal termination of the application program. The amount of storage available is specified when the IMS/VS system is defined.

- Dynamic Log Space

  A checkpoint call frees dynamic log records. If checkpoints are not issued frequently enough, the dynamic log can reach a "wraparound" point. If this happens, the program has to issue checkpoints more frequently, or the storage for the dynamic log has to be increased.

Checkpoints in MPPs and Transaction-Oriented BMPs

Issuing a checkpoint call in an MPP or a transaction-oriented BMP generates an internal retrieval call to the message queue. If there is a message available, IMS/VS returns it to the program. How often you issue checkpoint calls in these programs depends on whether the programs are single or multiple mode.

Single mode means that each retrieval call to the message queue is a sync point for the program. Single mode is recommended for recovery because sync points occur automatically, each time the program issues a call to retrieve a new message. Single mode programs don't need checkpoint calls, unless a single-mode transaction-oriented BMP is too long to restart from the beginning. This type of program should issue all checkpoint calls, instead of retrieval calls to the message queue. This is for recovery reasons. A checkpoint call, in addition to establishing a sync point in the program's processing, returns a new message, if one is available, to the application program's I/O area. In other words, it acts like a call to the message queue for a new message. If a program issues a checkpoint call, does some processing, and then issues a call to the message queue to retrieve a new message, the program can have a recovery problem if it terminates abnormally. The checkpoint call and the message retrieval call both establish sync points; but a program can be restarted only from a checkpoint. If the program terminates abnormally after issuing the message retrieval call, IMS/VS backs out the data base updates to the most recent sync point—the message retrieval call. But IMS/VS will restart the program from the most recent checkpoint call. The program's restart processsing will be inaccurate because the processing prior to the abnormal termination was not entirely backed out.

In multiple mode programs, the only sync points are the checkpoint calls that the program issues. If the program terminates abnormally, IMS/VS backs out the program's data base updates and cancels the messages it's created since its last checkpoint call. Since the only sync points are the checkpoint calls that the program issues, you can issue calls to the message queue and checkpoint calls. The considerations for issuing checkpoint calls in multiple mode programs are recovery considerations: the program should issue checkpoint calls frequently enough to make it easy to back out and recover the program if the program terminates abnormally.

Checkpoints with Data Sharing

When several application programs in one IMS/VS system run online, the IMS/VS control region makes it possible for them to read and update IMS/VS data bases concurrently. It does this by preventing a program from accessing data that another program is updating until the updating program indicates to IMS/VS that its updates are valid. Data sharing extends what the control region does by making it possible for application programs in separate IMS/VS systems—running in the same or separate processors—to access data bases concurrently.

In an online IMS/VS system, an application program cannot access a segment if another program is reading or updating it. If the program is reading the segment, the segment is available to other programs as soon as the first program moves its position to a new data base record. If the program is updating the segment, the segment does not become available to other application programs until the updating program reaches a sync point. If the updating program does not issue checkpoints frequently enough, other application programs needing the same segment might have to wait for long periods of time. Issuing frequent checkpoint calls reduces the time that other programs have to wait for updated segments. When application programs in separate IMS/VS systems share data bases, issuing frequent checkpoint calls in all types of programs is important because the data base is being shared among application programs in several IMS/VS systems.

CHAPTER 5. GATHERING REQUIREMENTS FOR DATA COMMUNICATIONS OPTIONS

If your IMS/VS system uses IMS/VS data communications, you will need to gather information about the data communications requirements of your application as well as the DL/I requirements.

This chapter tells you what information you need to provide to the data communications administrator and the system programmer, and it tells you why that information is important. After the explanation of each DC option, you will find a description of the information that you need to provide to those responsible for choosing or not choosing these options.

The tasks that this stage of application design involves are:

- **Identifying Online Security Requirements**

  IMS/VS protects online data through the use of sign-on, terminal, and password security. Sign-on security makes it possible for you to allow only authorized individuals access to IMS/VS. Terminal security means that individuals can enter certain transaction codes or commands only from specific terminals. Password security enables you to associate a transaction code or command with a password, so that only individuals who know the password can enter that transaction code or command. This section tells you what characteristics of your application you should examine when considering each type of security.

- **Analyzing Screen and Message Formats**

  Choosing and defining the kind of editing that an application program requires depends on the program's input and output. The program's input is the information that the program receives from someone at a terminal or from another application program. The program's output is the information that the program sends back to the person at the terminal or to another application program. This section gives an overview of how IMS/VS edits messages between an application program and a terminal, and it tells you what information you need to supply.

- **Gathering Requirements for Conversational Processing**

  If an MPP bases its processing on successive input from the terminal, or if the input from the terminal is too long or complex to be entered as one message, conversational processing can make it possible for the transaction to be processed in several cycles. If an MPP uses conversational processing, there is some additional information that you need to provide about the MPP's processing and data.

- **Identifying Output Message Destinations**

  In addition to replying to the terminal that sent an input message, an MPP can send output messages to other terminals. To do this, the MPP must use one or more alternate PCBs. This section describes the information that you can provide about your application that relates to this requirement.

IDENTIFYING ONLINE SECURITY REQUIREMENTS

You use security to protect the data from unauthorized use. Protecting the data in an online system doesn't stop at protecting the data base itself; you need to protect the application programs that access that data as well. For example, you don't want a

program that processes paychecks to be available to everyone who can access the system.

There are three ways that you can protect an application program from being used by unauthorized users. These are sign-on, terminal, and password security. Chapter 6, "Establishing IMS/VS Security," in the IMS/VS System Administration Guide explains how you define these types of security.

LIMITING ACCESS TO SPECIFIC INDIVIDUALS: SIGN-ON SECURITY

With sign-on security, individuals who want to access IMS/VS have to be defined to IMS/VS before they are allowed to use IMS/VS. If your installation uses MVS, sign-on security is available through the Resource Access Control Facility (RACF), or a user-written security exit. If your installation doesn't use MVS, sign-on security is available through a user-written security exit.

When a person signs on to IMS/VS, IMS/VS verifies that the person is authorized to use IMS/VS before the person is allowed access to the system. IMS/VS checks authorization when you enter the /SIGN ON command. You can also limit the transaction codes and commands that certain individuals are allowed to enter. You do this by associating an individual's user identification (USERID) with the transaction codes and commands that you want that individual to be able to enter.

LIMITING ACCESS TO SPECIFIC TERMINALS: TERMINAL SECURITY

Terminal security enables you to limit the entry of a transaction code to a particular terminal or group of terminals in the system. How you do this depends on how many programs you want to protect. To protect a particular program, you can either authorize a transaction code to be entered from a list of logical terminals; or you can associate each logical terminal with a list of the transaction codes that a user can enter from that logical terminal. For example, you could protect the paycheck application program by defining the transaction code associated with it as valid only when entered from the terminals in the payroll department. If, on the other hand, you wanted to restrict it more than this, you could associate the paycheck transaction code with only one logical terminal. This means that to enter that transaction code, you have to be at a certain terminal.

LIMITING ACCESS TO THE PROGRAM: PASSWORD SECURITY

The third way you can protect the application program is to require a password when a person enters the transaction code associated with the application program you want to protect. If you use only password security, that means that before IMS/VS will process a particular transaction code, the person entering the transaction code must also enter its password.

If you use password security with terminal security, that allows you to restrict access to the program even more. In the paycheck example, using password security and terminal security means that you can restrict unauthorized individuals within the payroll department from executing the program.

SUPPLYING SECURITY INFORMATION

When you evaluate your application in terms of its security requirements, you need to look at each program individually. Once you've done this, you should be able to supply the following information to the person in charge of security at your installation.

- For programs that require sign-on security:

- List the individuals who should be able to access IMS/VS

- For programs that require terminal security:
  - List the transaction codes that must be secured.
  - List the terminals that should be allowed to enter each of these transaction codes. If the terminals you're listing are already installed and being used, identify the terminals by their logical terminal names. If not, identify them by the department that will use them—for example, the accounting department.

- For programs that require password security:
  - List the transaction codes that should require passwords.

- For commands that require security:
  - List the commands that require sign-on or password security.

ANALYZING SCREEN AND MESSAGE FORMATS

When an application program communicates with a terminal, an editing routine translates messages from the way they are entered at the terminal to the way the program expects to receive and process them. The decisions about how IMS/VS will edit your program's messages are based on how the data that's being processed should be presented to the person at the terminal and to the application program. You need to describe how you want data from the program to appear on the terminal screen, and how you want data from the terminal to appear in the application program's I/O area.

To supply information that will be helpful in this decision, you should be familiar with how IMS/VS edits your messages. IMS/VS has two editing routines that an application program can use:

- **Message Format Service, or MFS,** is most commonly used with display screens. MFS uses control blocks that define what a message should look like to the person at the terminal and to the application program.

- **Basic edit** is available to all IMS/VS application programs. Basic edit removes control characters from input messages and inserts the control characters you specify in output messages to the terminal.

Chapter 3, "Design Considerations for IMS/VS Networks," in the IMS/VS System Administration Guide contains information on defining IMS/VS editing routines.

AN OVERVIEW OF MFS

MFS uses four kinds of control blocks to format messages between an application program and a terminal. The information you gather about how you want the data formatted when it is passed between the application program and the terminal is contained in these control blocks. MFS creates the control blocks from the information supplied in the message statement (MSG) and the format statement (FMT).

There are two control blocks that describe input messages to IMS/VS.

- The device input format, or DIF, describes to IMS/VS what the input message will look like when it's entered at the terminal.

- The message input descriptor, or MID, tells IMS/VS how the application program expects to receive the input message in its I/O area.

By using the DIF and the MID, IMS/VS can translate the input message from the way that it's entered at the terminal to the way it should appear in the program's I/O area.

There are two control blocks that describe output messages to IMS/VS.

- The message output descriptor, or MOD, tells IMS/VS what the output message will look like in the program's I/O area.
- The device output format, or DOF, tells IMS/VS how the message should appear on the terminal screen.

Many installations have a specialist who defines application requirements to MFS for all the applications at the installation. To define the MFS control blocks for an application program, the specialist needs to know how you want the data to appear at the terminal and in the application program's I/O area for both input and output. The IMS/VS Message Format Service User's Guide explains how you define this information to MFS.

AN OVERVIEW OF BASIC EDIT

Basic edit removes the control characters from an input message before the application program receives it, and inserts the control characters you specify when the application program sends a message back to the terminal. To format output messages at a terminal using basic edit, you need to supply the necessary control characters for the terminal you're using.

Again, you need to describe how you want the data to be presented at the terminal, and what it will look like in the program's I/O area.

EDITING CONSIDERATIONS IN YOUR APPLICATION

Before you describe the editing requirements of your application, be sure that you are aware of standards that your installation has concerning screen design. Make sure that the requirements that you describe comply with those standards.

The information you should provide about your program's editing requirements is:

- How you want the screen to be presented to the person at the terminal for the person to enter the input data. For example, if an airlines agent wants to reserve seats on a particular flight, the screen that asks for this information could look like this:

```
FLIGHT#:
NAME:
NO. IN PARTY:
```

- What the data should look like when the person at the terminal enters the input message
- What the input message should look like in the program's I/O area
- What the data should look like when the program builds the output message in its I/O area
- How the output message should be formatted at the terminal
- The length and type of data that your program and the terminal will be exchanging

The type of data you're processing is only one consideration when you analyze how you want the data presented at the terminal. In addition, you should consider the needs of the person at the terminal—the human factors in your application—with the effect of the screen design on the efficiency of the application program—the performance factors in the application program. Unfortunately, sometimes there is a tradeoff between human factors and performance factors. A screen design that's easily understood and used by the person at the terminal may not be the design that gives the application program its best performance. Again, your first concern should be that you're following whatever screen standards your installation has established.

A terminal screen that has been designed with human factors in mind is one that puts the person at the terminal first; it's one that makes it as easy as possible for that person to interact with the application program. Some of the things you can do to make it easy for the person at the terminal to understand and respond to your application program are:

- Display a small amount of data at one time.
- Use a format that is clear and uncluttered.
- Provide clear and simple instructions.
- Display one idea at a time.
- Require short responses from the person at the terminal.
- Provide some means for help and ease of correction for the person at the terminal.

At the same time, you don't want the way in which a screen is designed to have a negative effect on the application program's response time, or on the system's performance. When you design a screen with performance first in mind, you want to reduce the processing that IMS/VS must do with each message. To do this, the person at the terminal should be able to send a lot of data to the application program in one screen so that IMS/VS doesn't have to process additional messages. And the program should not require two screens to give the person at the terminal information that it could give on one screen.

When describing how the program should receive the data from the terminal, you need to consider the program logic and the type of data you're working with.

GATHERING REQUIREMENTS FOR CONVERSATIONAL PROCESSING

Conversational processing means that the person at the terminal can have several interactions with the application program. When you use conversational processing, the person at the terminal enters some information, and an application program processes the information and responds to the terminal. The person at the terminal then enters more information for an application program to process. Each of these interactions between the person at the terminal and the program is called a step in the conversation.

WHAT HAPPENS IN A CONVERSATION

A conversation is a series of terminal/program interactions.

Although it's not apparent to the person at the terminal, a conversation can be processed by several application programs instead of only one. If you use only one application program to process the conversation, the conversation may or may not take place during one scheduling of the program. For example, IMS/VS might schedule the program to start the transaction, then, if there are no other messages for the program to process, the

program will terminate. When the person at the terminal responds, IMS/VS schedules the program again to continue the conversation.

In order for the same program or another program to continue a conversation, the program must have the necessary information to continue processing. IMS/VS stores data from one stage of the conversation to the next in a scratchpad area, or SPA. When a program continues the conversation (the same program or a different one), IMS/VS gives the program the SPA for the conversation associated with that terminal.

In the airlines example above, the first program might save the flight number and the names of the people traveling, then pass control to another application program to reserve seats for those people on that flight. The program saves this information in the SPA. If the second application program didn't have the flight number and names of the people traveling, it wouldn't be able to do its processing.

DESIGNING A CONVERSATION

A conversation can be processed by one application program, or it can be processed by several application programs. The first part of designing a conversation is to design the flow of the conversation. If the requests from the person at the terminal will be processed by only one application program, then designing the flow of the conversation is simply a matter of designing that program. If, on the other hand, you decide that the conversation should be processed by several application programs, you need to decide which stages of the conversation each program will process, and what each program will do when it has finished processing its stage of the conversation.

When a person at a terminal enters a transaction code that has been defined as conversational, IMS/VS schedules the conversational program (for example, Program A), that is associated with that transaction code. When Program A issues its first call to the message queue, IMS/VS returns the SPA that has been defined for that transaction code to Program A's I/O area. The person at the terminal has to enter the transaction code (and password, if there is one) only on the first input screen. IMS/VS treats data in subsequent screens as a continuation of the conversation started in the first screen.

After it has retrieved the SPA with a message retrieval call, Program A can issue another message retrieval call to retrieve the input message from the terminal. After it has processed the message, Program A can then continue the conversation, or it can end the conversation. There are several ways to do both of these things.

To continue the conversation, Program A can:

- Reply to the terminal that sent the message.
- Reply to the terminal and pass the conversation to another conversational program, for example, Program B. This is called a deferred program switch, and it means that Program A responds to the terminal and then passes control to another conversational program—in this case, Program B. Program A does this by placing the transaction code associated with Program B in the SPA, then issuing a call to return the SPA to IMS/VS. For this type of switch, Program A references the I/O PCB in the call. After doing this, Program A is no longer part of the conversation. The next input message that the person at the terminal enters will go to Program B, although the person at the terminal will be unaware that this message is being sent to a second program.
- Pass control of the conversation to another conversational program without first responding to the originating terminal. This is called an immediate program switch; an immediate

program switch lets you pass control directly to another conversational program without having to respond to the originating terminal. Program A does this by placing the new transaction code in the SPA, then issuing a call to return the SPA to IMS/VS. In an immediate switch, however, the call that the program issues must reference an alternate PCB. When you do this, the program that you pass the conversation to has the responsibility of responding to the person at the terminal. To continue the conversation, Program B then has the same choices as Program A did: It can respond to the originating terminal and keep control, or it can pass control in a deferred or immediate program switch.

If Program A wants to end the conversation, it can:

- Move blanks to the first 8 bytes of the SPA, then return the SPA to IMS/VS.
- End the conversation and pass control to a nonconversational program. This is also called a deferred switch, but Program A ends the conversation before passing control to another application program. The second application program can be an MPP or a transaction-oriented BMP that processes transactions from the conversational program.

THINGS YOU NEED TO KNOW ABOUT THE SPA

When Program A passes control of a conversation to Program B, Program B needs to have the data that Program A saved in the SPA in order to continue the conversation. IMS/VS gives the SPA for the transaction to Program B when Program B issues its first message call.

There are some restrictions about passing conversational control to another program that have to do with the type and size SPA associated with the transaction codes for the programs. Each SPA has three characteristics about it that are defined for it at system definition. They are:

- The size of the SPA
- Where the SPA will be stored—in main storage or on a direct access storage device
- Whether the SPA is fixed length or variable length.

Also at system definition, the system administrator defines the maximum size allowed in the system for each type (main storage or direct access) of SPA. There are some restrictions about passing a conversation to another program. These restrictions depend on the SPA's length (fixed or variable) and storage (main storage or direct access). There are two restrictions that concern the SPA's length.

- If the first program in the conversation uses a fixed-length SPA, all the programs in the conversation must use fixed-length SPAs as well.
- If the first program in the conversation uses a variable-length SPA, the other programs in the conversation can use either fixed- or variable-length SPAs. If the first program to be scheduled for the conversation uses a variable-length SPA, and the next program uses a smaller or larger SPA, IMS/VS truncates or extends the SPA for the new program. If IMS/VS truncates the SPA, it saves the truncated data.

What these two restrictions mean is that if the first program in the conversation uses a fixed SPA, the program can pass control only to programs that use fixed-length SPAs. If, on the other hand, the first program in the conversation uses a

variable-length SPA, the program can pass control to a program that uses either a fixed- or variable-length SPA.

There is another restriction that depends on the maximum SPA length defined for the system. At system definition, the system administrator defines a maximum length for main storage SPAs and a maximum length for direct access storage SPAs. A program cannot pass the conversation to another program whose SPA size is larger than this maximum size. If the maximum size defined for main storage SPAs is different from the maximum size defined for direct access SPAs, then IMS/VS uses the maximum length of the type of SPA used by the first program in the conversation. For example, if the SPA used by the first program is a main storage SPA, then the maximum length defined for main storage SPAs is the maximum length for any SPAs (main storage or direct access) used during the conversation.

Suppose that Programs A, B, C, and D process transactions A, B, C, and D respectively. Figure 36 shows the type and length of SPA defined for each transaction. For this example, assume that the maximum length for a main storage SPA is 100 bytes, and the maximum length for a direct access SPA is 500 bytes.

Transaction Code	Type of Storage for SPA	Length of SPA
TRANA	Main Storage	50 bytes
TRANB	Direct Access	100 bytes
TRANC	Direct Access	300 bytes
TRAND	Main Storage	100 bytes

Figure 36. Example of SPA Storage

The length of SPA that IMS/VS allows during a conversation depends on the type and length of the SPA used by the first program scheduled during the conversation. For example:

- If Program A or Program D is the first program scheduled during the conversation, the maximum length for any SPA used during the conversation is 100 bytes. That's because these programs process transactions whose SPAs are defined as main storage SPAs, and the maximum length for a main storage SPA is 100 bytes.

  If Program A is the first program to be scheduled, it can pass the conversation to Program B or Program D, but Program A can't pass the conversation to Program C.

- If Program B or Program C is the first program scheduled during the conversation, the maximum SPA length for the conversation is 500 bytes. If Program B is the first program scheduled during the conversation, it can pass control to any of the other programs. The same is true of Program C.

"Conversations" describes how you structure a conversational program.

Recovery Considerations in Conversations

Because a conversation involves several stages and can involve several application programs, there are some special things you should understand about recovery in a conversation. This list is a summary of these special considerations:

- One thing you can do to make recovery easier is to design the conversation so that all the data base updates are done in the

last step of the conversation. This way, if the conversation terminates abnormally, IMS/VS can back out all the updates because they were all made during the same stage of the conversation. Updating the data base during the last stage of the conversation is also a good idea, because the input from each stage of the conversation is available.

- Although a conversation can terminate abnormally during any step of the conversation, IMS/VS backs out only the data base updates and output messages resulting during the last step of the conversation. IMS/VS doesn't back out data base updates or cancel output messages for previous steps, even though some of that processing might be inaccurate as a result of the abnormal termination.

- There is a system service call that can be helpful if the program determines that some of its processing was invalid. The rollback call (ROLB) backs out all of the changes that the program has made to the data base, and cancels the output messages that the program has created (except those sent with an express PCB, as explained below), since the program's last sync point.

- The program can use an express PCB to send a message to the person at the terminal, and to the master terminal operator. When the program sends a message using an express PCB, IMS/VS sends the message no matter what. Messages sent with an express PCB are sent to their final destinations even if the program terminates abnormally or issues a rollback call. A rollback call does not cancel messages sent with an express PCB.

- To help verify the accuracy of the previous processing, and to correct the processing that's determined to be inaccurate, you can use the conversational abnormal termination routine—DFSCONE0. IMS/VS schedules DFSCONE0 and uses it for all conversations that terminate abnormally. DFSCONE0 can identify the transaction that was being processed when the abnormal termination occurred because IMS/VS gives the SPA to DFSCONE0. The SPA contains the transaction code. The SPA also indicates which step in the conversation was being processed when the conversation terminated abnormally. DFSCONE0 needs this information so that it can determine what action to take.

- You can write an MPP to examine the SPA, send a message notifying the person at the terminal of the abnormal termination, make any necessary data base calls, and use a user-written or system-provided exit routine to schedule it.

IDENTIFYING OUTPUT MESSAGE DESTINATIONS

When an application program sends an output message, it does so by issuing a call and referencing the I/O PCB or an alternate PCB. In the same way that a DB PCB represents a hierarchy that the application program processes, the I/O PCB and alternate PCBs represent logical terminals and other application programs with which the application program communicates. An application program can send messages to other application programs and to terminals.

THE ORIGINATING TERMINAL

To send a message to the logical terminal that sent the input message, the program uses an I/O PCB. Sending a message with the I/O PCB sends the output message to the logical terminal that sent the message. The program doesn't have to do anything to the I/O PCB before sending the message. IMS/VS puts the name of the logical terminal that sent the message in the I/O PCB when the program receives the message.

TO OTHER PROGRAMS AND TERMINALS

When you want to send an output message to a terminal other than, or in addition to, the terminal that sent the input message, you use an alternate PCB. You can set the alternate PCB for a specific logical terminal when the program's PSB is generated, or you can define the alternate PCB as being modifiable. A program can change the destination of a modifiable alternate PCB while the program is executing, so if you use a modifiable alternate PCB, you can send output messages to several alternate destinations.

There are situations in which the application program must respond to the originating terminal before the person at the originating terminal can send any more messages. These situations occur when a terminal is in exclusive mode; a terminal is in response mode; or during conversational mode.

- Exclusive mode applies to a terminal. When a terminal is in exclusive mode, the only messages it can receive are those that are in response to an input message that was entered at the terminal after the /EXCLUSIVE command was entered. The terminal remains in exclusive mode until the person at the terminal enters the /END command. If there were other messages sent to the terminal while exclusive mode was in effect, IMS/VS sends them to the terminal after the /END command has been entered.

- Response mode can apply to a communication line, a terminal, or a transaction. When response mode is in effect, IMS/VS won't accept any input from the communication line or terminal until the program has sent a response to the previous input message. The originating terminal is unusable (for example, the keyboard locks) until the program has processed the transaction and sent the reply back to the terminal.

  You can define communication lines and terminals as operating in response mode; not operating in response mode; or operating in response mode only if processing a transaction that's been defined as response mode. You specify response mode for communication lines and terminals on the TYPE and TERMINAL macros, respectively, at IMS/VS system definition. You can define any transaction as a response mode transaction; you do this on the TRANSACT macro at IMS/VS system definition. Response mode is in effect if:

  - The communication line has been defined as being in response mode.
  - The terminal has been defined as being in response mode.
  - The transaction code has been defined as response mode.

- Conversational mode applies to a transaction. When a program is processing a conversational transaction, the program must respond to the originating terminal after each input message it receives from the terminal.

In these processing modes, the program doesn't send a message to an alternate destination; it must respond only to the originating terminal. But sometimes the originating terminal is a physical terminal that's made up of two components—for example, a printer and a punch. If the physical terminal is made up of two components, each component has a different logical terminal name. If the program needs to send an output message to the printer part of the terminal, it has to use a different logical terminal name than the one associated with the input message. In other words, it has to send the output message to an alternate destination. There is a special kind of alternate PCB that a program can use in these situations; it's called a response alternate PCB.

A response alternate PCB lets you send messages when exclusive mode, response mode, or conversational mode is in effect. The destination of a response alternate PCB must be a logical

terminal—you can't use a response alternate PCB to represent another application program. When you use a response alternate PCB during response mode or conversational mode, the logical terminal represented by the response alternate PCB must represent the same physical terminal as the originating logical terminal. This is not a requirement when using a response alternate PCB during exclusive mode.

You can also specify an alternate PCB as being an express PCB. When you send messages using any of the PCBs described so far, IMS/VS doesn't send the message if your program terminates abnormally. But when you send messages using an express PCB, IMS/VS sends the messages to the destination represented by the express PCB no matter what happens to your application program after you've sent the message. If the application program terminates abnormally, or if the program issues a rollback call, IMS/VS cancels messages that the program has sent using an I/O PCB, an alternate PCB, a modifiable alternate PCB, or a response alternate PCB. But IMS/VS sends messages that use an express alternate PCB to their final destinations as soon as the program issues the call to send the message. These messages are not affected if the program terminates abnormally, or if the program issues a rollback call.

You should provide the answers to the questions below to the data communications administrator to help in establishing your application's requirements:

- Will the program be required to respond to the terminal before the terminal can enter another message?
- Will the program be responding only to the terminal that sends input messages?
- If the program needs to send messages to other terminals or programs as well, is there only one alternate destination?
- What are the other terminals to which the program will have to send output messages?
- Should the program be able to send an output message before it terminates abnormally?

PART 2. APPLICATION PROGRAMMING GUIDE

This part is a guide to coding IMS/VS application programs. The tasks this part explains are:

- Structuring the DL/I Portion of a Program
- Coding the DL/I Portion of a Program
- Structuring a Message Processing Program
- Coding a Message Processing Program
- Structuring and Coding a Batch Message Program
- Testing an Application Program
- Documenting an Application Program

CHAPTER 6. STRUCTURING THE DL/I PORTION OF A PROGRAM

This chapter explains how you use DL/I to read and update a data base. How you use DL/I calls to do this is the same, whether you're writing a message processing program (MPP), a Fast Path program, a batch message program (BMP), or a batch program.

This chapter gives you the basics you need to specify DL/I call sequences and call formats for any of these types of programs. If you are writing a batch program that might someday be converted to a batch message program (BMP), you can refer to "Planning Ahead for Batch-to-BMP Conversion" for additional design considerations that can make this conversion easier.

Some of the information in this chapter applies only to batch programs. You will find additional considerations for MPPs in "Chapter 8. Structuring a Message Processing Program," and for BMPs in "Chapter 10. Structuring and Coding a Batch Message Program." "Processing Fast Path Data Bases" explains Fast Path exceptions to the information in this chapter.

A batch program has only one source of input—the input you supply—but an online program can also receive input from terminals and other programs. A batch program might retrieve and update data base records directly, or it might process data sequentially, then print a report by listing a particular part of the data base.

This chapter gives you an introduction to how an application program processes a DL/I data base, then explains the decisions that you will be making about your program:

- **How You Read and Update a DL/I Data Base: An Overview**

  This section explains the parts of a DL/I program and also explains the examples that are used throughout this chapter.

- **Retrieving Information**

  This section describes the calls that a program uses to retrieve information from the data base: GU, GHU, GN, GHN, GNP, and GHNP.

- **Updating Information**

  This section explains the calls that a program uses to update information in the data base: DLET and REPL. It also explains how you use the get hold calls with the update calls.

- **Inserting Information**

  This sections tells you how you use the ISRT call to add new segment occurrences to an existing data base, and to initially load a data base.

- **Determining Your Position in the Data Base**

  This section explains when your position in the data base is important, how each of the data base calls affects your position in the data base, and where your position is after a call that DL/I is unable to satisfy.

- **Techniques to Make Programming Easier**

  This section explains some things you can do to make your programming easier. It also gives some general programming guidelines.

- **Checking Status Codes**

  This section explains different types of status codes and explains what your error routine should do.

- **Taking Checkpoints**

  This section explains why checkpoints are important in a batch program and describes symbolic CHKP and XRST and basic CHKP.

- **Using Secondary Indexes and Logical Relationships**

  This section tells you how secondary indexing and logical relationships affect your programming.

- **Planning Ahead for Batch-to-BMP Conversion**

  This section describes some of the reasons that people convert batch programs to BMPs and gives some suggestions about the conversion.

- **Designing a Program that Uses GSAM**

  This section explains how you process a GSAM data base.

- **Processing Fast Path Data Bases**

  This section describes how you use the DL/I calls to process Fast Path data bases. It also explains three additional calls that you can use with Fast Path data bases: the field call, or FLD; the position call, or POS; and the synchronization call, or SYNC.

HOW YOU READ AND UPDATE A DL/I DATA BASE: AN OVERVIEW

To access data in a DL/I data base, you issue calls to DL/I from your application program. The calls tell DL/I what you want done; whether you want to retrieve information, delete information, replace information, or add information. When you retrieve or update information from the data base, you do so in segments.

In addition to specifying what you want DL/I to do in a call, you can give DL/I some additional search criteria about the segment you want to process. You give this information in segment search arguments, or SSAs. Giving this information helps DL/I to find the segment you want.

DL/I communicates the results of your call to you in two places. First, you use an I/O area in your program to pass segments back and forth between your program and DL/I. What the I/O area contains depends on the kind of call you're issuing:

- When you retrieve a segment, DL/I places the segment you requested in the I/O area.
- When you add a new segment, you place the new segment in the I/O area.
- When you delete or replace a segment, you have to first retrieve the segment using one of the get hold calls. When you issue a get hold call for a segment, DL/I places the requested segment in an I/O area, just as it does after the other retrieve calls.
  - To delete that segment, you issue a DLET call and reference the same I/O area in the DLET call. You don't have to do anything to the segment or the I/O area; you just issue the DLET call.

- To replace a field or fields in the segment, you modify those fields in the segment in your I/O area. Then you issue a REPL call and reference that I/O area.

The second place that DL/I uses to describe the results of your call is the DB PCB. After a retrieval call, DL/I places this information about the call's results in the DB PCB:

- The level number of the lowest segment that DL/I accessed while processing the call.
- A status code that tells you whether or not the call was successful.
- The name of the lowest-level segment that DL/I accessed while processing the call.
- The concatenated key of the segment being accessed. A segment's concatenated key is made up of the keys of all the segment's parents and the segment's own key.
- The length of the concatenated key of the segment being accessed.

To find out about the results of a DL/I call, your program needs to look at this information in the DB PCB. But PCBs are outside the application program; the PSB that contains a given program's PCBs resides in an IMS/VS library. To use the DB PCB, the program defines a mask of the DB PCB. It can then reference the mask to check the DB PCB after each call and find out about the success or failure of the call. An application program doesn't change any fields in a DB PCB; the program only checks it to determine what happened when the call was issued.

Figure 37 shows the structure of a DL/I program. The numbers on the right refer to the notes that follow.

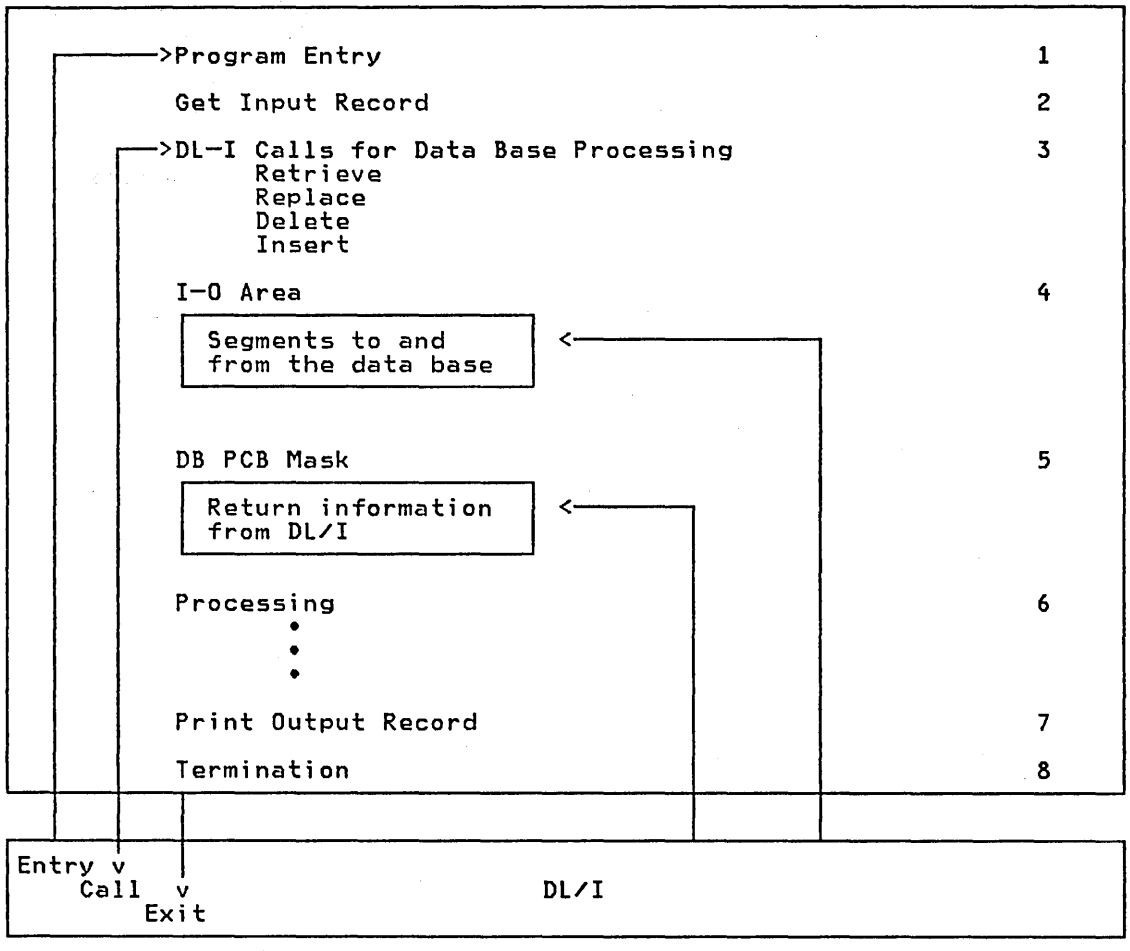

Figure 37. DL/I Program Structure

Notes:

1. **Program entry.** IMS/VS passes control to the application program. The entry statement lists the DB PCBs that the program uses in the order in which they appear in the PSB.
2. **Get an Input Record.** The program retrieves an input record.
3. **DL/I calls.** The program issues DL/I calls to read and update information from the data base.
4. **I/O area.** DL/I returns the requested information to the program's I/O area.
5. **DB PCB mask.** DL/I describes the results of each DL/I call in the DB PCB mask. Your program checks the information in this area after each DL/I call to find out whether or not the call was successful.
6. **Processing.** The program does the required processing, including issuing additional DL/I calls, if necessary.
7. **Print Output Record.** The program might print an output record based on its processing.
8. **Termination.** The program returns control to IMS/VS when it has finished its processing. In a batch program you can, if you wish, set the return code and pass it to the next step in

the job. If you don't use the return code in this way, it's a good idea to set it to zero as a programming convention. You can also use the return code for this purpose in BMPs. But in MPPs, setting the return code is meaningless, since MPPs can't pass return codes.

DL/I CALLS

A DL/I program reads and updates data by issuing DL/I calls. There are three calls you can use to retrieve data, two calls you can use to update existing data, and one call you can use to add new data to the data base.

The retrieval calls are get unique (GU), get next (GN), and get next within parent (GNP). There are also three special retrieval calls, called get hold calls, that you use just before you want to update or delete a segment. Each retrieval call has a corresponding get hold call: get hold unique, or GHU; get hold next, or GHN; and get hold next within parent, or GHNP.

There are two calls you use to update data that already exists in the data base. These are delete (DLET) and replace (REPL). But before you can update a segment, you have to retrieve the segment. You do this by using one of the get hold calls; then you issue the DLET or REPL call immediately after the get hold call.

There is one call you use to add new segments to a data base; this is the insert call (ISRT). You use ISRT to add new segments to an existing data base, and you use it to initially load a data base.

A DL/I call is made up of a call statement and a list of parameters. The parameters for each call are function, DB PCB, I/O area, and SSA. (There is an additional parameter that is required in PL/I; this is the number of the parameters that follow for this call.) The parameters give the addresses in your program of the information described above.

Figure 38 shows the parameters and the information they provide.

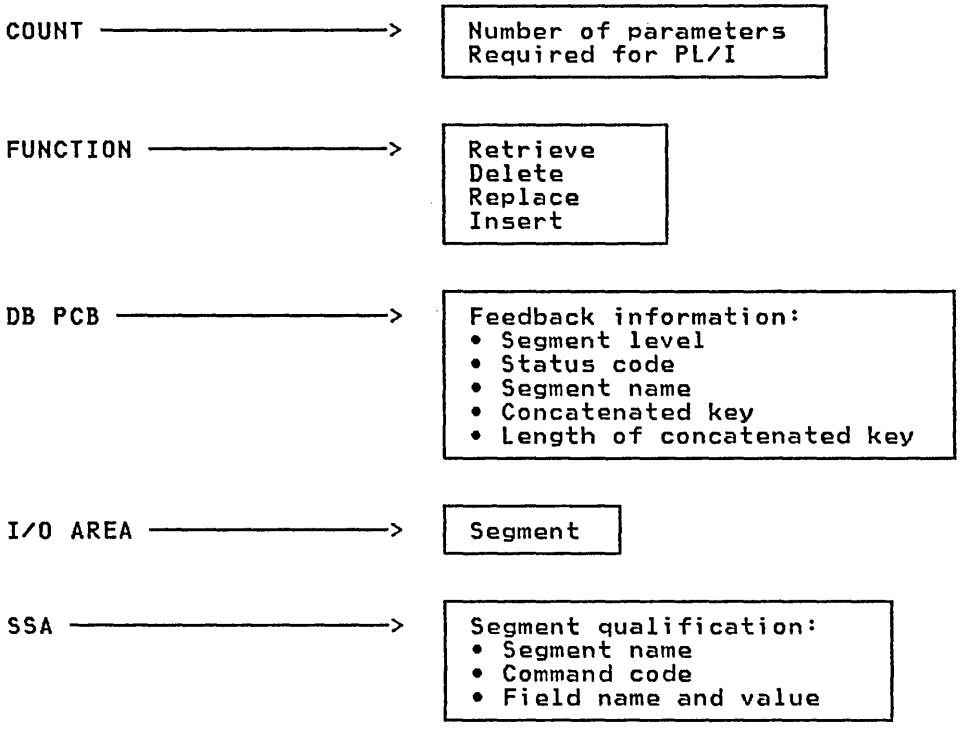

Figure 38. DL/I Call Parameters

SSAs

Using segment search arguments (SSAs) in a DL/I call lets you provide IMS/VS with information that is used to satisfy the DL/I call. You can provide just the segment name, or you can further describe the segment you want by including a condition that the segment must meet. There are two kinds of DL/I call statements:

- A call with no SSAs is an unqualified call.
- A call that contains one or more SSAs is a qualified call.

Just as DL/I calls may be qualified or unqualified, SSAs may also be qualified or unqualified. When you say that a DL/I call is qualified, you're saying that it contains at least one qualified or unqualified SSA:

- An SSA that contains only a segment name is an unqualified SSA. An unqualified SSA describes the segment **type** that you want to access.
- An SSA that contains, in addition to the segment name, one or more qualification statements, is a qualified SSA. A qualified SSA describes the segment **occurrence** that you want to access.

A DL/I call can include more than one SSA; if it does, it can contain a mixture of qualified and unqualified SSAs.

An unqualified SSA gives DL/I the name of the segment type that you want to access. Figure 39 shows the structure of an unqualified SSA.

Seg Name	b
8	1

Figure 39. Unqualified SSA Structure

The segment name field (Seg Name) is 8 bytes long, and it must be followed by a 1-byte blank.

In addition to the name of the segment type you want to access, a qualified SSA contains a description of the particular segment occurrence you want. This description is called a qualification statement and has three parts. Figure 40 shows the structure of this kind of SSA.

Seg Name	(	Fld Name	R.O.	Fld Value	)
8	1	8	2	Variable	1

Figure 40. Qualified SSA Structure

Using a qualification statement enables you to give DL/I information about the particular segment occurrence you're looking for. You do this by giving DL/I the name of a field within the segment, and the value of the field that you're looking for.

The qualification statement is enclosed in parentheses. The first field contains the name of the field (Fld Name) that you want DL/I to use in searching for the segment. The second field contains a relational operator (R.O.) that tells DL/I how to compare the values in the fields in the data base with the value you supply. The relational operator can be any one of the following:

- Equal, represented as

  =b
  b=
  EQ

- Greater than, represented as

  >b
  b>
  GT

- Less than, represented as

  <b
  b<
  LT

- Greater than or equal to, represented as

  >=
  =>
  GE

- Less than or equal to, represented as

  <=
  =<
  LE

- Not equal to, represented as

  ¬=
  =¬
  NE

Note:

"b" indicates a blank.

The third field, Fld Value, contains the value that you want DL/I to use as the comparative value.

Using qualified SSAs makes it possible for you to describe to DL/I virtually any segment occurrence in the hierarchy that you want to retrieve.

You can use more than one qualification statement in an SSA; "Using Multiple Qualification Statements" explains how you do this.

Command Codes

There is one more thing that you can include in an SSA. This is a command code. Command codes make DL/I calls do more than the calls do by themselves.

Both qualified and unqualified SSAs can include one or more command codes. The first command code follows the segment name field in the SSA and is separated from the segment name field with an asterisk (*).

Figure 41 shows the format of an unqualified SSA with a command code.

Seg Name	*	Cmd Code
8	1	1

Figure 41. Unqualified SSA with Command Code

Figure 42 shows the structure of a qualified SSA with a command code.

Seg Name	*	Cmd Code	(	Fld Name	R.O.	Fld Value	)
8	1	1	1	8	2	Variable	1

Figure 42. Qualified SSA with Command Code

For example, command codes can change the way in which DL/I handles the segment you have requested. Suppose you code a D command code on a GU call that retrieves segment F in the hierarchy shown in Figure 43. This call would look like this:

```
GU   Abbbbbbb*D
     Cbbbbbbb*D
     Ebbbbbbb*D
     Fbbbbbbbb
```

DL/I always returns the lowest segment in the path to your I/O area. But when you code a D command code, DL/I returns all the preceding segments in the path, in this case A, C, and E, as well. This method is called a path call.

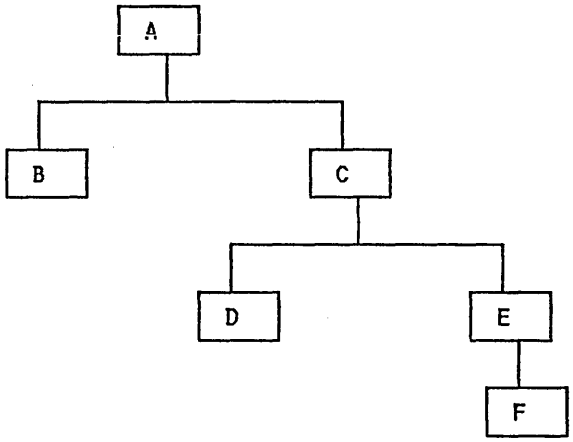

Figure 43. D Command Code Example

You can use command codes with all the DL/I calls except DLET. "Using Command Codes" explains the command codes and gives examples of each; the following is a summary of the command codes and their meanings.

C The C command code lets you identify a segment by its concatenated key. You code the concatenated key in the area of the SSA that would otherwise contain the qualification statement. You can use the C command code with get calls and ISRT calls.

D The D command code lets you retrieve or insert a sequence of segments in a hierarchic path using only one call—instead of having to retrieve each segment with a separate call. A call that includes the D command code is called a path call. You can use the D command code with get calls and ISRT calls.

F You use the F command code to indicate to DL/I that you want DL/I to start searching for the segment you've specified with the first occurrence of that segment type under a specific parent. The F command code makes it possible for a GN or GNP call to "back up" in the data base. You can use the F command code with get calls and ISRT calls.

L The L command code indicates to DL/I that you want to go directly to the last occurrence of a segment under its parent; you use it when you know that the segment you're looking for is the last occurrence of the segment type. You can use the L command code with get calls and ISRT calls.

N If you retrieve a sequence of segments using the D command code, and you don't want to replace all of those segments, you can use the N command code to indicate to DL/I which segment or segments you don't want to replace. You code the N command code in the SSA for each of the segments you don't want replaced. The N command code applies only to REPL calls; DL/I ignores it if you include it in any other type of call.

P Normally, DL/I establishes parentage at the lowest level satisfied in a particular call. You can set parentage at a higher level by including the P command code in the SSA for the level at which you want

parentage set. You use the P command code only with get calls. DL/I ignores it if you use it with other calls.

Q When your program is processing online data bases, you can use the Q command code to hold a segment so that other programs can't update it while your program is using it. You only use the Q command code in online programs; DL/I ignores it if you use it in other circumstances. "Reserving and Releasing Segments" explains how you use the Q command code and what its effects are. You can use the Q command code with get calls and ISRT calls.

U When DL/I processes a get or ISRT call, it establishes a position on the segment occurrence that satisfies the call at each level in the path of the segment you're retrieving or inserting. A U command code on an SSA in a get call or an ISRT call tells DL/I not to move from the established position at the level of the SSA when trying to satisfy the call. You can use the U command code with get calls and ISRT calls.

V Coding a V command code on an SSA is like coding the U command code on each level of the call. When you use the V command code, DL/I can't move from its current position on any of the levels of the path. You can use the V command code with get calls and ISRT calls.

- (null) The null command code (coded "-") makes it possible for you to supply the command code that you want to use during program execution, instead of coding it as a constant in the SSA.

DB PCB Masks

An application program communicates processing requests to DL/I by issuing DL/I calls. DL/I, in turn, describes the results of these calls in the DB PCB that was used for the call. To find out about the success or failure of the DL/I call, the application program builds a mask of the DB PCB in the program's data area, and then references the fields of the DB PCB through the DB PCB mask.

A DB PCB mask must contain the same fields as a DB PCB, in the same order, and of the same length. Figure 44 shows the order and lengths of these fields, and the notes below the figure describe the contents of each field. A DB PCB mask for a GSAM data base is slightly different from a DB PCB mask for a DL/I data base. "PCB Masks for GSAM Data Bases" explains these differences.

1. Data Base Name 8 bytes
2. Segment Level Number 2 bytes
3. Status Code 2 bytes
4. Processing Options 4 bytes
5. Reserved for DL/I 4 bytes
6. Segment Name 8 bytes
7. Length of Key Feedback Area 4 bytes
8. Number of Sensitive Segments 4 bytes
9. Key Feedback Area variable length

Figure 44. DB PCB Mask

When you code a DB PCB mask, you also give it a name, but this is not a field in it. The name you give to the DB PCB is the name of the area that contains all the fields in the DB PCB. In COBOL and assembler language programs, you list the names you've given your DB PCBs in the entry statement; in PL/I programs, you list the pointers to your DB PCBs. DB PCBs don't have names assigned to them in the PSB. In your entry statement, you associate the name in your program with a particular DB PCB based on the order of all of the PCBs in the PSB. In other words, the first PCB name in the entry statement corresponds to the first PCB; the second name in the entry statement corresponds to the second PCB; and so on.

Each DB PCB represents a hierarchic structure that the application program processes. The DB PCBs for a particular application program are contained in the PSB for the application program.

A DB PCB mask contains the following fields:

1. **Data Base Name**

   This is the name of the DBD. This field is 8 bytes long and contains character data.

2. **Segment Level Number**

   When DL/I retrieves the segment you've requested, DL/I places the level number of that segment in this field. If you're retrieving several segments in a hierarchic path with one call, DL/I places the number of the lowest level segment retrieved. If DL/I is unable to find the segment you've requested, it gives you the level number of the last segment it encountered that satisfied your call. This is the lowest segment on the last path that DL/I encountered while searching for the segment you requested. This field contains numeric data. It is 2 bytes long and right-justified.

3. **DL/I Status Code**

   DL/I places a 2-character status code in this field after each DL/I call. This code describes the results of the DL/I call. DL/I updates it after each call and does not clear it between calls. The application program should test this field after each call to find out whether or not the call was successful. If the call was completely successful, this field contains blanks.

   There are three categories of status codes. Some indicate exceptional but valid conditions. They are often for information only. For example, GB means that DL/I has reached the end of the data base before satisfying the call. This situation is expected in sequential processing and is not usually the result of an error.

   Other status codes, such as AK, which means that you have included an invalid field name in an SSA, indicate errors in the program. The program should have error routines available for these status codes. You can terminate the program abnormally, correct it, and restart it.

   The third category of status codes indicates an I/O or system error. For example, an NO status code means that there has been an I/O error concerning ISAM, OSAM, BSAM, or VSAM. If your program encounters a status code in this category, it should terminate immediately. This type of error can't normally be fixed without a system programmer, data base administrator, or system administrator.

4. **Processing Options**

   This is a 4-byte field containing a code that tells DL/I what type of calls this program can issue. It is a security mechanism, in that it can prevent a particular program from updating the data base, even though the program can read the data base. This value is coded in the PROCOPT parameter of the PCB statement when the PSB for the application program is generated. The value does not change.

5. **Reserved for DL/I**

   This 4-byte field is used by DL/I for internal linkage. It is not used by the application program.

6. **Segment Name**

   After each call, DL/I places in this field the name of the segment from the lowest level reached in trying to satisfy the call. When a retrieval is successful, this field contains the name of the retrieved segment. If the retrieval is unsuccessful, this field contains the last segment, along the path to the requested segment, that satisfies the call. This field is 8 bytes long.

7. **Length of Key Feedback Area**

   This is a 4-byte binary field that gives the current length of the key feedback area described below. Since the key feedback area isn't usually cleared between calls, the program needs this length to determine the length of the current concatenated key in the key feedback area.

8. **Number of Sensitive Segments**

   This is a 4-byte binary field that contains the number of segment types in the data base to which the application program is sensitive.

9. **Key Feedback Area**

After a successful retrieval call, DL/I places the concatenated key of the retrieved segment in this field. The length of this field is the length given in the 4-byte length of key feedback area field above. If DL/I is unable to find the segment you've requested, DL/I places the concatenated key of the lowest-level segment encountered during the search for the requested segment. A segment's concatenated key is made up of the keys of each of the segment's parents and its own key. Keys are positioned left to right, starting with the key of the root segment and following the hierarchic path. The only situation in which DL/I clears this area is when DL/I is unable to find a root segment; when this happens, DL/I clears the area to binary zeros. The rest of the time, DL/I places the concatenated key for the current retrieval call on top of the concatenated key that's in this field from the last retrieval call. That's why you need to know the current length of the concatenated key; DL/I updates the length each time it gives you a new concatenated key so that you'll be able to accurately determine the current length of this field.

Of these nine fields, there are five that will be important to you as you structure the program. These are the fields that give information about the results of the call. They are the segment level number, the status code, the segment name, the length of the key feedback area, and the key feedback area. The status code is the field your program will use the most often to find out whether or not the call was successful. The key feedback area will contain the data from the segments you have specified; the level number and segment name will help you determine your position in the data base after an error or an unsuccessful call.

FOR EXAMPLE

The call examples used throughout this chapter use the medical hierarchy shown in Figure 45. To understand the examples, you will need to be familiar with the hierarchy and each of the segment formats.

The Medical Hierarchy

Suppose that a medical clinic uses the data base hierarchy in Figure 45 to process information about its patients.

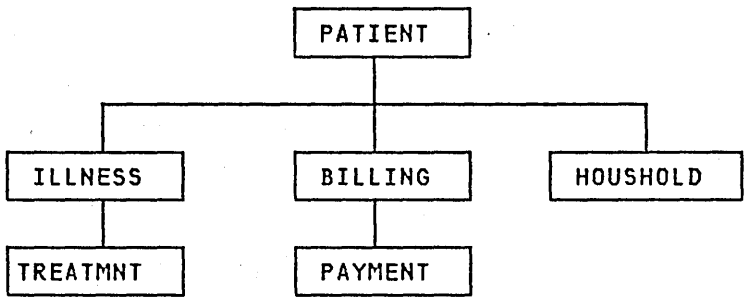

Figure 45. Medical Hierarchy

Medical Data Base Segment Formats

To understand the examples in this chapter, you will need to know the format of each segment in the hierarchy. The contents of each of the segment types in the hierarchy are described below. The number below each field is the length (in bytes) that has been defined for that field.

- **PATIENT**

  Figure 46 shows the PATIENT segment. It has three fields: patient number (PATNO), patient name (NAME), and the patient's address (ADDR). PATIENT has a unique key field: PATNO. PATIENT segments are stored in ascending order of their patient numbers. The lowest patient number in the data base is 00001; the highest is 10500.

PATNO	NAME	ADDR
5	10	30

Figure 46. PATIENT Segment

- **ILLNESS**

  Figure 47 shows the ILLNESS segment. It has two fields: the name of the illness (ILLNAME), and the date that the patient came to the clinic with the illness (ILLDATE). The key field is ILLDATE. Since it's possible for a patient to come to the clinic with more than one illness on the same date, this key field it nonunique. That means that there may be more than one ILLNESS segment with the same key field value. For segments with nonunique keys (or segments that have no keys specified for them), you specify the sequence in which you want them stored in the data base in the DBD. ILLNESS segments are stored on a first-in-last-out basis; in other words, if you retrieved all of the ILLNESS segments for a patient, DL/I would return the ILLNESS segment with the most recent date first, followed by the ILLNESS segment with the date before that. The last ILLNESS segment that DL/I returned would be the ILLNESS segment with the earliest date. You specify this for a segment in the DBD by specifying RULES=FIRST. This tells DL/I where to put new segment occurrences when you insert them. ILLDATE is specified in the format MMDDYYYY:

ILLDATE	ILLNAME
8	10

Figure 47. ILLNESS Segment

- **TREATMNT** Figure 48 shows the TREATMNT segment. It contains four fields: the date of the treatment (DATE); the medicine that was given to the patient (MEDICINE); the quantity of the medicine that the patient was given (QUANTITY); and the name of the doctor who prescribed the treatment (DOCTOR). The key field of the TREATMNT segment is DATE; since a patient may receive more than one treatment on the same date, DATE is a nonunique key field. TREATMNT, like ILLNESS has been specified as having RULES=FIRST. TREATMNT segments also stored on a first-in-last-out basis. DATE is specified in the same format that ILLDATE is—MMDDYYYY:

DATE	MEDICINE	QUANTITY	DOCTOR
8	10	4	10

Figure 48. TREATMNT Segment

- **BILLING**

  Figure 49 shows the BILLING segment. It has only one field—the amount of the current bill. BILLING has no key field.

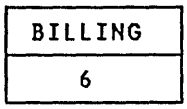

Figure 49. BILLING Segment

- **PAYMENT**

  Figure 50 shows the PAYMENT segment. It also has only one field—the amount of the payments for the month. The PAYMENT segment has no key field.

Figure 50. PAYMENT Segment

- **HOUSHOLD**

  Figure 51 shows the HOUSHOLD segment. It contains the names of the members of the patient's household, and how each is related to the patient. RELNAME is the key field.

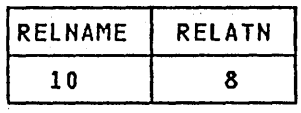

Figure 51. HOUSHOLD Segment

What Happens When You Issue a Call

When you issue a DL/I call, you communicate with DL/I through the call itself, the SSA, the I/O area, and the DB PCB mask. Here's an example of how all of these parts of a DL/I program work together.

Assume you want to find out when and why a particular patient was most recently at the clinic. For example, "Why was Ann Stevens last here, and when was it? Her patient number is 05123." Because PATNO is the key field in the PATIENT segment, you need the PATNO when you code the SSA for the segment. You would issue a GU call with these SSAs to retrieve this information:

```
GU   PATIENTb(PATNObbb=b05123)
     ILLNESSbb
```

This call says that you want the ILLNESS segment that was most recently added to the data base for the patient whose patient number is 05123. PATIENT and PATNO are padded with blanks on the right because each of these fields in the SSA must be 8 bytes long. 05123 is not padded with blanks, because the PATNO field is defined as being 5 bytes long in the DBD, and 05123 is 5 bytes. The length of the field you use in the SSA must be equal to the defined length of the field.

If you issue the call correctly, DL/I returns an ILLNESS segment for the patient whose number is 05123 to your I/O area. If this patient's most recent visit to the clinic was on February 1, 1980, because of stomach flu, the I/O area looks like this:

```
02011980STOMACHFLUxxxx...x
```

The x's in the I/O area represent whatever was in your I/O area before you issued the call.

The DB PCB that you used in the call contains this information for the call:

- Segment level number: **02**
- Status code: **bb**
- Segment name: **ILLNESSb**
- Length of concatenated key: **0013**
- Concatenated key: **0512302011980xxxxx...x**

What all of this tells you is that the lowest segment that DL/I accessed during your request was at the second level (02); that the call was completely successful (blank status code, represented by "bb"); that the name of the lowest segment accessed during your request was ILLNESS; that the length of the concatenated key of the segment returned is 13 bytes; that the key of the parent of the segment accessed is 05123, and that this segment's key is 02011980. You have to know the length of each key field to know where the key for one segment stops and the key for the next segment begins. The length of the concatenated key is given in binary in the DB PCB. This example shows it in decimal (0013) for clarity.

RETRIEVING INFORMATION

There are three calls that retrieve information from the data base: get unique, or GU; get next, or GN; and get next within parent, or GNP. You use GU to a particular segment directly, or to establish a position in the data base. You use GN for sequential retrieval; and you use GNP to retrieve dependents of a particular segment.

The information in this section about the get calls also applies to the corresponding get hold calls. What's true for GU is also true for GHU; what's true for GN is also true for GHN; and what's true for GNP is also true for GHNP. In each case, the only difference between the get call with the hold and the get call without the hold is that a DLET or REPL call can follow a hold call. "Updating Information" explains how you use the get hold calls with REPL and DLET.

When you issue retrieval calls, you can describe each segment in a hierarchic path in an SSA by segment type or by a field value. The last segment in the path is the segment you're trying to retrieve.

RETRIEVING SEGMENTS DIRECTLY: GU

A GU call can directly retrieve any sensitive segment in the data base. The GU call is the only call that can establish position backward in the data base. (The GN and GNP calls, when used with the F command code, can back up in the data base, but with limitations. "Retrieving and Inserting the First Occurrence: F" explains this.) Unlike GN and GNP, a GU call does not move forward in the data base automatically. If you issue the same GU call repeatedly, DL/I retrieves the same segment to you each time you issue the call.

How You Use GU

A GU call is a request for a segment as described or qualified by the SSAs you supply. You use it when there's a specific segment you want; you can also use it to establish your position in the data base. When you issue a GU, DL/I starts searching for the segment you've specified with the segment that's at current position in the data base. If you haven't established position in the data base, or if you reached the end of the data base as a result of the last call, DL/I starts searching for the segment you've specified at the beginning of the data base.

GU Examples

These are some examples of when you would use a GU call. The examples use the medical hierarchy explained earlier in this chapter.

- "What illness was Robert Martin here for most recently? Was he given any medication on that day for that illness? His patient number is 05163."

  To answer this request for information, you would issue a GU call with the following SSAs:

```
GU   PATIENTb(PATNObbb=b05163)
     ILLNESSbb
```

  Once you had retrieved the ILLNESS segment with the date of the patient's most recent visit to the clinic, you could issue another call to find out whether or not he was treated during that visit. If the date of his most recent visit was January 5, 1980, you could issue the call below to find out whether or not he was treated on that day for that illness:

```
GU   PATIENTb(PATNObbb=b05163)
     ILLNESSb(ILLDATEb=b01051980)
     TREATMNT(DATEbbbb=b01051980)
```

- "What is Joan Carter currently being treated for? Her patient number is 10320."

  In this example, you want the ILLNESS segment for the patient whose patient number is 10320:

```
GU   PATIENTb(PATNObbb=b10320)
     ILLNESSb
```

Using SSAs with GU

Because you use GU when you want to retrieve a specific segment, or when you want to set your position in the data base to a specific place, you almost always use qualified GU calls. A GU call may have as many SSAs as there are levels in the hierarchy defined by the DB PCB that you're referencing in the GU call. If the segment you want is on the fourth level of the hierarchy, you can use four SSAs to retrieve the segment. (There would never be any reason to use more SSAs than levels in the hierarchy. If your hierarchy only has three levels to it, you would never need to locate a segment lower than the third level.) Here's what happens when you use GU with different kinds of SSAs:

- A GU with one unqualified SSA retrieves the first occurrence of the segment you've specified.

  For example, if you issue a GU call qualified only with PATIENT (GU PATIENTb) as the first call in your program, DL/I returns the first PATIENT segment in the data base to your program. This is the PATIENT segment with the lowest PATNO—00001.

If you issue this call when you do have position established in the data base, DL/I still returns the PATIENT segment for the patient with the patient number 00001.

- A GU with a qualified SSA can retrieve the segment described in the SSA regardless of that segment's location relative to current position. For example, suppose your position in the data base was on the TREATMNT segment for the PATIENT segment with the key 01034, and you wanted to retrieve the TREATMNT segment with the date March 4, 1980, for the patient whose number is 01000. If you issued the call:

```
GU   PATIENTb(PATNObbb=b01000)
     TREATMNT(DATEbbbb=b03041980)
```

  DL/I can retrieve that segment even though the PATIENT segment with the patient number equal to 01000 is behind current position.

- A GU with multiple qualified SSAs returns the first occurrence that satisfies the higher level requirements if the segments have nonunique keys or no keys at all. For example, when you issue a call for an ILLNESS segment qualified with a particular date, DL/I returns the first ILLNESS segment it encounters that has that date. This may not be the ILLNESS segment you want to retrieve. Since ILLNESS segments have nonunique keys, there could be several ILLNESS segments for the same PATIENT with the same DATE. This is because it would be possible for a patient to have two illnesses on the same date. TREATMNT segments also have nonunique keys, since it would also be possible for a patient to receive several treatments on one day.

- When a segment has nonunique keys, or no keys at all, a GU with qualified SSAs will return only the first occurrence of that segment. In other words, if a patient had two TREATMNT segments with the date April 4, 1980, a GU for the PATIENT segment and the SSA TREATMNTb(DATEbbbb=b04041980) could only retrieve the first of these segment occurrences. To retrieve the second TREATMNT segment with the date April 4, 1980, you could retrieve the first TREATMNT segment with a GU, then issue a GN with the same SSA.

- When you issue a GU that mixes qualified and unqualified SSAs, at each level DL/I retrieves the first occurrence of the segment type that satisfies the call.

- If you leave out an SSA for one of the levels in a GU with multiple SSAs, DL/I assumes an unqualified SSA for that segment.

Using Command Codes with GU

All the command codes except N can be used with GU. The D command code is used frequently with GU, because it allows you to retrieve a specific dependent segment and all of its parents in the hierarchic path by issuing only one call.

GU and Parentage

Parentage is set to the segment retrieved for any subsequent GNP call. If the GU call was unsuccessful, the previous parentage, if any, is destroyed.

GU Status Codes

Some of the status codes that apply specifically to GU are:

bb Blanks. DL/I has retrieved the segment you requested.

GE DL/I was unable to find one or more of the segments described by the SSAs in the call. "Current Position after Unsuccessful Calls" explains where your position in the data base is after a GE status code.

GG DL/I returns a GG status code to a program with a processing option of GOT or GON when the segment that the program is trying to retrieve contains an invalid pointer. Position in the data base after a GG status code is just before the first root segment occurrence in the hierarchy. The PCB key feedback area will contain the length of the key of the last root segment accessed.

RETRIEVING SEGMENTS SEQUENTIALLY: GN

A GN call retrieves the next segment in the hierarchy that satisfies the SSAs, if any, that you supply. Since the segment retrieved by a GN call depends on the current position in the hierarchy, GN is often issued after a GU call. If no position has been established in the hierarchy, GN retrieves the first segment in the data base. A GN call retrieves a segment or path of segments by moving forward from the current position in the data base. As processing moves forward DL/I looks for segments at each level to satisfy the call.

Sequential retrieval in a hierarchy is always top to bottom and left to right. For example, if you repeatedly issue unqualified GNs against the hierarchy in Figure 52, DL/I returns the segment occurrences in the data base record in this order:

After the root segment (A1), B1 and its dependents:

C1, D1, F1, D2, D3, E1, E2, and G1

followed by H1 and its dependents:

I1, I2, J1, and K1.

If you issue an unqualfied GN again, after DL/I has returned K1, DL/I returns the root segment occurrence whose key follows segment A1 in the data base.

A GN call qualified with the segment type, on the other hand, can retrieve all the occurrences of a particular segment type in the data base. For example, if you issue a GN call with qualified SSAs for segments A1 and B1, and an unqualified SSA for segment type D, DL/I returns segment D1 the first time you issue the call, segment D2 the second time you issue the call, and segment D3 the third time you issue the call. If you issue the call a fourth time, DL/I returns a status code of GE. GE means that DL/I could not find the segment you requested.

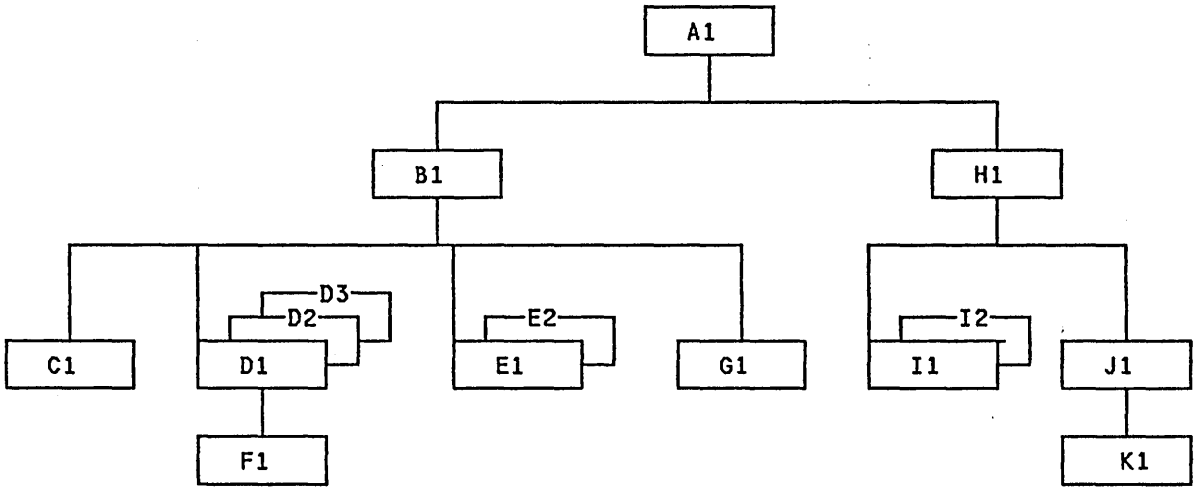

Figure 52. Hierarchic Sequence

How You Use GN

A GN call is a request for a segment, as described by the SSAs you supply, that's linked to the preceding call. The starting place for DL/I's search is current position. Special status codes are returned whenever a different segment type at the same level or higher level is returned. No special status code is returned when a segment at a lower level is returned. You can check for a lower level in the PCB. Use qualified GN calls whenever possible.

When you use GN:

- Processing moves forward, except when the F command code is used.
- DL/I uses the current position set by the previous call as the starting point of the search.
- The segment retrieved is determined by a combination of the next sequential position in the hierarchy and the SSAs included in the call.

An unqualified GN retrieves the next segment in the hierarchy starting at the current position. Each unqualified GN call retrieves the next sequential segment forward from the current position. For example, to answer the processing request:

Print out the entire medical data base.

You would simply issue an unqualified GN call repeatedly until DL/I returned a GB status code.

Information-only status codes for this type of GN are:

GA The call was successful. In satisfying the call, a hierarchic boundary was crossed to a higher level. This status code applies only to GN calls that are unqualified.

GB The end of the data base has been reached. If the GN call is issued after a GB status code has been returned, the first segment in the hierarchy is returned.

GK The call was successful. The segment returned is a different segment type at the same level. This status code applies only to GN calls that are unqualified.

Using SSAs with GN

Like a GU, a GN can have as many SSAs as there are levels in the hierarchy. Using fully qualified SSAs with GN calls clearly identifies the hierarchic path and the segment you want. Using fully qualified SSAs also provides for better documentation, control, and future change implications. Be sure that you are familiar with your installation's standards concerning data base calls.

A GN with an unqualified SSA retrieves the next occurrence of that segment type by going forward from the current position. For example, to satisfy the processing request:

We need a list of all patients that have been to this clinic.

You would continue issuing a GN qualified with PATIENT until DL/I returned a GB status code. GB means that DL/I has reached the end of the data base before being able to satisfy your call.

```
GN   PATIENTbb
```

A GN with qualified SSAs retrieves the next occurrence of the specified segment type that satisfies the SSAs. For example, suppose that, at the end of each month, the clinic wanted to know the names of the patients who had been to the clinic during the month:

What are the names of the patients we have
seen this month?

You would continue issuing the GN call with the SSAs shown below until DL/I returned a GB status code: (The example shows the call you would use at the end of April in 1980.) Since you would need the PATIENT segments in order to list the names of the patients, you would want DL/I to return the PATIENT segments, in addition to the ILLNESS segment, to your program. If you coded the D command code on the SSA for the PATIENT segment, DL/I would return the PATIENT segments to your program.

```
GN   PATIENTb*D
     ILLNESSb(ILLDATEb>=04011980)
```

When you specify a GN that has multiple SSAs, the presence or absence of unqualified SSAs in the call has no effect on the operation unless you use command codes in the call. DL/I uses only qualified SSAs plus the last SSA to determine the path and retrieve the segment. Unspecified or unqualified SSAs for higher level segments in the hierarchy mean that any high level segment that is the parent of the correct lower level specified or qualified segment will satisfy the call.

A GN with one SSA qualified on a unique key field defines by key value a unique segment that you want to retrieve. Higher level qualified SSAs define the unique segments that are part of the path.

Using Command Codes with GN

The command codes that you can use with GN are C, D, F, L, P, Q, U, and V.

GN and Parentage

Parentage is set at the segment retrieved by any subsequent GNP call unless you use the P command code in the call. If a GN was unsuccessful, the previous parentage, if any, is destroyed.

GN Status Codes

bb Blanks. Call was successful; DL/I has returned the segment(s) you requested to your I/O area.

GA DL/I has returned the segment you requested to your I/O area. This segment is from a higher level than the last segment. This is a warning that the position in the data base has changed with respect to the previous path.

GB In trying to satisfy the GN call, DL/I reached the end of the data base.

GE DL/I was unable to find one or more of the segments described by the SSAs in the call for one of these reasons:

- The segment you asked for doesn't exist.
- The segment you asked for can't be found by searching forward in the hierarchy. In other words, current position is past the segment you requested.

"Current Position after Unsuccessful Calls" explains where your position in the data base is after a GE status code.

GG DL/I returns a GG status code to a program with a processing option of GOT or GON when the segment that the program is trying to retrieve contains an invalid pointer. Position in the data base after a GG status code is just before the first root segment occurrence in the hierarchy. The PCB key feedback area will contain the length of the key of the last root segment accessed.

GK DL/I has returned the segment you requested to your I/O area. This segment is a different segment type at the same level. This status code is a warning for calls without SSAs. It tells you that the program is working with a different segment type.

RETRIEVING DEPENDENTS SEQUENTIALLY: GNP

A get next within parent call, abbreviated GNP, like a GN, retrieves segments sequentially. The difference between a GN and a GNP is that for a GNP call the segments that can satisfy the call are limited to the lower level dependent segments of the established parent. Your program must issue a successful GU or GN call to establish parentage before it can issue a GNP call. The GNP call will not affect the current parentage, unless you use the P command code in the GNP call.

How You Use GNP

Because GNP lets DL/I limit the scope of the call, using GNP can be more efficient than GN. This is not always true—it depends on your particular application.

The most important part of using a GNP is to understand how parentage is set. Parentage is set in one of two ways:

- The lowest level segment returned by the most recent call against a particular PCB. If you subsequently issue another GU or GN, but against a different PCB, this will not affect the parentage that you set using the first PCB in the previous call.

- The use of the P command code for the purpose of setting parentage at a particular level.

A GNP request is linked to the previous DL/I calls issued by your program in two ways:

- Positioning: The search for the requested segment starts at the current position, that is, the place in the hierarchy reached by the previous call.
- Parentage: The search for the requested segment is limited to the dependents of the lowest segment most recently accessed by a GU or GN call.

In other words, positioning determines the start of the search, and parentage determines the end of the search. There is always some positioning in effect. If positioning has not been established from a DL/I call, current position is immediately preceding the root segment. Parentage, on the other hand, is in effect only following a successful GU or GN call.

An unqualified GNP retrieves the first dependent segment occurrence under the current parent. If your current position is already on a dependent of the current parent, an unqualified GNP retrieves the next segment occurrence.

For example, to answer this processing request:

> We need the complete record for Kate Bailey.
> Her patient number is 09080.

You want to retrieve only the dependent segments of the patient whose name is Kate Bailey, and whose patient number is 09080. You don't want to retrieve all of the dependents of each patient. To do this, you would use a GU call to establish your position on the PATIENT segment for Kate Bailey. Then you would continue issuing an unqualified GNP until DL/I had returned all the dependent segments of that PATIENT segment.

```
GU   PATIENTb(PATNObbb=b09080)
GNP
```

A GE status code would indicate that you have retrieved all the dependent segments for the PATIENT segment whose key is 09080.

Using SSAs with GNP

You can include one or more SSAs in a GNP call; the SSAs may be qualified or unqualified. Without SSAs, a GNP call retrieves the next sequential dependent of the established parent. The advantage of using SSAs with GNP is that they allow you to point DL/I to a specific dependent or dependent type of the established parent.

A GNP with an unqualified SSA sequentially retrieves the dependent segment occurrences of the segment type you've specified under the established parent. For example, if you wanted to answer the request:

> Which doctors have been prescribing acetaminophen
> for headaches?

Suppose that, for this example, the key of ILLNESS is ILLNAME, and the key of TREATMNT is MEDICINE. In this example, you want to sequentially process the ILLNESS segments whose illness is headache, and you want to retrieve each TREATMNT segment where the treatment was acetaminophen. (Note that "acetaminophen" is abbreviated to "ACETAMINOP" because this field is defined as 10 bytes long.) The name of the doctor who prescribed the treatment is part of the TREATMNT segment:

```
GN   ILLNESSb(ILLNAMEbEQHEADACHEbb)
GNP  TREATMNTbb(MEDICINE=bACETAMINOP)
```

To process this, you would loop back to the GNP call until DL/I returned a a GE (not found) status code, then you would retrieve the next headache segment and retrieve the TREATMNT segment for it. You would do this until there were no more ILLNESS segments where the ILLNAME was headache.

Notice in this call that if you didn't want to see the ILLNESS segment, you could issue one GN call qualified with both SSAs to retrieve the TREATMNT segment.

A GNP with a qualified SSA describes to DL/I the segment you want retrieved, or the segment that is to become part of the hierarchic path to the segment you want retrieved. A qualified GNP describes a unique segment only if it's qualified on a unique key field, and not a data field or a nonunique key field.

A GNP with multiple SSAs defines the hierarchic path to the segment you want. The SSAs must be for segments lower than the established parent, and the last SSA describes the segment you want. Multiple unqualified SSAs establish the first occurrence of the specified segment type as part of the path to the segment you want. If there are missing SSAs between the parent and the requested segment in a GNP call, they are generated internally as unqualified SSAs. This means that DL/I includes the first occurrence of the segment from the missing SSA as part of the hierarchic path to the segment you have requested.

Using Command Codes with GNP

You can use any of the command codes except N in a GNP.

GNP and Parentage

GNP affects parentage only if it includes a P command code. Otherwise, GNP does not establish or change parentage.

GNP Status Codes

bb Blanks. Call was successful; the segment(s) you requested have been returned to the I/O area.

GA The segment retrieved as a result of an unqualified GNP is at a higher level in the hierarchy than the previous segment retrieved, but it is still below the current parent. This status code warns you that the position in the data base has changed.

GE DL/I did not find the segment you described in one or more SSAs (qualified or unqualified) under the established parent. If the requested segment exists, this could mean that the segment is behind the current position, or it could mean that the segment does not exist under the established parent. "Current Position after Unsuccessful Calls" explains where your position in the data base is after a GE status code.

GG DL/I returns a GG status code to a program with a processing option of GOT or GON when the segment that the program is trying to retrieve contains an invalid pointer. Position in the data base after a GG status code is just before the first root segment occurrence in the hierarchy. The PCB key feedback area will contain the length of the key of the last root segment accessed.

GK DL/I has returned a segment to your I/O area that's a dependent of the established parent and is part of the same higher level path as the previously returned segment, but the new segment is a different segment type. DL/I returns this

status code after unqualified GNP calls as a warning to you that you are working with a new segment type.

GP This status code usually means that there was no parentage in effect when you issued the GNP call; DL/I does not return a segment to your I/O area. This status code can also mean that the segment specified in the SSA is at a level equal to or higher than the currently established parent. These errors are usually programming errors. If parentage is not established, this can be caused by one of the following reasons:

- Neither a GU nor a GN call has been issued to establish parentage.
- A GU or GN call was issued, but was unsuccessful.
- The established parent was just deleted.

USING THE RIGHT RETRIEVAL CALL

Sometimes more than one of the retrieval calls will accomplish the same thing. When you think that you have a choice about which retrieval call to use, you should take the following considerations into account:

- If you want only particular segments, issue fully qualified GUs for these segments instead of GNs.
- If you want to retrieve a specific segment occurrence or obtain a specific position within the data base, use GU.
- If you're moving forward in the data base, even if you're not retrieving every segment in the data base, use GN and GNP with SSAs.
- Be careful when you use GN because it's possible to use SSAs with it that would force DL/I to search to the end of the data base without retrieving a segment. This is particularly true with the "not equal" or "greater than" relational operators.
- Don't use a GN call that doesn't limit the number of root segments that can be accessed.
- Use GNP when possible; the more you limit the search by giving DL/I information, the less your program has to do, and the more efficiently DL/I will process your call.

UPDATING INFORMATION

There are two calls your program can issue to change the data in the data base. A replace call, abbreviated REPL, replaces the current data in a particular segment with the new data you supply in your program's I/O area. A delete call, abbreviated DLET, removes the segment occurrence(s) you specify from the data base.

BEFORE YOU UPDATE: GET HOLD CALLS

When you want to delete or replace a segment, you need to first obtain the segment and indicate to DL/I that you are about to change the segment in some way. You do this by issuing a get call with a "hold" before you issue the DLET or REPL call. Once you have issued a successful get hold call for the segment you want to change, you can delete the whole segment or change one or more fields (except the key field) in the segment.

There are three get hold calls, one for each of the get calls. Get hold unique, or GHU, is the hold form for a GU call; get hold next, or GHN, is the hold form for a GN call; and get hold next within parent, or GHNP, is the hold form for a GNP call.

The only difference between the get calls with a hold and get calls without a hold is that the hold calls can be followed by a REPL or DLET call. If a hold call is not immediately followed by a DLET or REPL call, the hold status on the retrieved segment is canceled and must be reestablished before reissuing the DLET or REPL call.

If, after issuing a get hold call, you find out that you don't have to update it after all, you can go on to other processing without releasing the segment. The segment will be freed as soon as the current position changes—when you issue another call to the same PCB you used for the get hold call. In other words, a get hold call must precede a REPL or DLET call; however, issuing a get hold call doesn't require you to replace or delete the segment.

REPLACING SEGMENTS: REPL

A REPL call must be preceded by one of the three get hold calls. After you retrieve the segment, you modify it in the I/O area, then issue a REPL call to replace it in the data base. You cannot change the lengths of any of the fields of the segment in the I/O area before you issue the REPL call.

How You Use REPL

A REPL call must be preceded by a get hold call; DL/I replaces the segment in the data base with the segment you modify in the I/O area.

You can replace more than one segment at a time by using the D command code in the get hold call, then issuing the REPL call. If, in the path of segments you're replacing, you don't want to replace one or more of the segments, you can issue the REPL call with unqualified SSAs containing the N command code. The N command code indicates to DL/I that you don't want to replace the segment described by the SSA containing the N. If you don't use the D command code in the get hold call, but you do use multiple SSAs, the REPL call replaces only the last segment successfully retrieved from the data base with the segment in the I/O area.

REPL Examples

The following are two examples of the REPL call. The first example replaces only one segment; the second example replaces two segments in a path of segments.

- "We have received a payment for $65.00 from a patient whose name is Margaret Collins, and whose patient number is 08642. Update her billing record and her payment record with this information, and print a current bill for her."

  There are four parts to satisfying this processing request:

  1. Get the BILLING and PAYMENT segments for Margaret Collins.
  2. Calculate the new values for these segments by subtracting $65.00 from the value in the BILLING segment, and adding $65.00 to the value in the PAYMENT segment.
  3. Replace the values in the BILLING and PAYMENT segments with the new values.
  4. Print a bill for Margaret Collins showing her name, number, and address; the current amount of her bill; and the amount of her payments to date.

  To retrieve the BILLING and PAYMENT segments, you issue a get hold call. You can retrieve both segments with one call by using the D command code in the call. Because you will also

need the PATIENT segment when you print the bill, you can include the D command code in the SSA for the PATIENT segment, and in the SSA for the BILLING segment:

```
GHU  PATIENTb*D(PATNObbb=b08642)
     BILLINGb*D
     PAYMENTbb
```

DL/I always returns the segment described by the lowest SSA, so you don't have to use the D command code for the PAYMENT segment.

After you have calculated the current bill and payment, you can print the bill, then replace the billing and payment segments in the data base. Before issuing the REPL call, you change the segments in the I/O area.

Since you haven't changed the PATIENT segment, you don't need to replace it when you replace the BILLING and PAYMENT segments. To indicate to DL/I that you don't want to replace the PATIENT segment, you use an unqualified SSA with the N command code for PATIENT when you issue the REPL call.

```
REPL  PATIENTb*N
```

This call tells DL/I to replace the BILLING and PAYMENT segments, but not to replace the PATIENT segment. The SSA with the N command code for PATIENT tells DL/I that you don't want to replace that segment.

- "Steve Arons, patient number 10250, has moved to a new address in this town. His new address is 4638 Brooks Drive, Lakeside, California. Update the data base with his new address."

  In this example, you need to retrieve the PATIENT segment for Steve Arons and replace the address portion of the segment. To retrieve the PATIENT segment, you can use the get hold call below:

```
GHU  PATIENTb(PATNObbb=b10250)
```

  Since you aren't replacing the first two fields of the PATIENT segment (PATNO and NAME), you don't have to change them in the I/O area. You place the new address in the I/O area following the PATNO and NAME fields. Then you issue the REPL call. In this example, you don't need any SSAs:

```
REPL
```

  In the call itself, you would reference the I/O area that contains the PATIENT segment, in addition to the DB PCB.

Using SSAs with REPL

After retrieving several segments using a get hold call with the D command code, you can use one or more SSAs in a REPL call to indicate to DL/I the segments in the path that you don't want replaced. You code the N command code on the SSA for each segment that you haven't changed. DL/I will not replace these segments if you do this.

If you issue a REPL call with an unqualified SSA containing an N command code after a get hold call that did not use the D command code, DL/I simply ignores the SSA in the REPL call. This means that you can use the same SSA in the REPL call regardless of whether or not you issued a get hold call with the D command code.

But if you include a qualified SSA in a REPL call, DL/I returns an AJ status code to inform you of this.

Using Command Codes with REPL

N is the only command code that is valid in a REPL call. DL/I rejects any other status codes that are included in a REPL call.

REPL and Parentage

Unless you are using a secondary index, REPL does not affect parentage. If you are using a secondary index, and you replace the indexed segment, parentage is lost. "How Secondary Indexing Affects Your Program" explains how using a secondary index affects your program.

REPL Status Codes

bb The segment has been successfully replaced.

DA The segment has not been replaced, because the REPL call attempted to change the key field.

DJ The segment has not been replaced, because the program did not issue a get hold call before issuing the REPL call.

VI The segment has not been replaced, because the length of the variable-length segment you supplied in the I/O area is invalid.

DELETING SEGMENTS: DLET

Like a REPL call, a DLET call must be immediately preceded by one of the three get hold calls. When you issue the DLET call, DL/I deletes the held segment, along with all its physical dependents, from the data base regardless of whether or not your program is sensitive to all these segments. In other words, be careful when using this call.

How You Use DLET

DL/I rejects the DLET call if the call immediately preceding it was not a get hold call. If the DLET call is successful, the previously retrieved segment and all of its dependents are removed from the data base and cannot be retrieved again.

DLET Examples

This is an example of the DLET call.

- "Evelyn Parker has moved away from this area. Her patient number is 10450. Delete her record from the data base."

  In this example, you want to delete all the information about Evelyn Parker from the data base. To do this, you only have to delete the PATIENT segment; when you do this, DL/I deletes all the dependents of that segment. This is exactly what you want DL/I to do—there wouldn't be any reason to keep the ILLNESS and TREATMNT segments for Evelyn Parker after Evelyn Parker was no longer one of the clinic's patients.

  Before you can delete the PATIENT segment, you have to retrieve it with a get hold call:

```
GHU  PATIENTb(PATNObbb=b10450)
```

To delete this patient's data base record, you issue a DLET call and reference the same DB PCB and I/O area that you referenced in the get hold call. In this example you don't need any SSAs:

```
DLET
```

Using SSAs with DLET

Unless the get call that precedes the DLET call is a path call, no SSAs are allowed in a DLET call. If the get hold call was a path call, however, you indicate to DL/I which one of the retrieved segments (and its dependents, if any) you want deleted by specifying an unqualified SSA for that segment. This is the only situation in which an SSA is allowed in a DLET call.

Using Command Codes with DLET

None of the command codes are valid in a DLET call.

DLET and Parentage

A DLET call does not affect parentage unless the DLET call eliminates the established parent. If this happens, you will need to reestablish parentage before issuing a GNP call.

DLET Status Codes

bb DL/I has deleted the segment or segments you specified.

DJ DL/I did not delete the segment, because the program did not issue a get hold call before issuing the DLET call.

INSERTING INFORMATION

You use the same call to add information to an existing data base and to initially load a data base. This is the ISRT call. The call looks the same in either case; the way it's used is determined by the processing option in the PCB.

ADDING INFORMATION TO AN EXISTING DATA BASE

ISRT can add new occurrences of an existing segment type to a HIDAM, HISAM, or HDAM data base.

New segments cannot be added to an HSAM data base unless you reprocess the whole data base or you add the new segments to the end of the data base.

How You Use ISRT to Add Segments

Before you issue the ISRT call, you must build the new segment in the I/O area. The fields of the segment you build in the I/O area must be in the same order and of the same length as defined for the segment. The DBD defines the fields that a segment contains, and the order in which they appear in the segment. If you are adding a root segment occurrence, DL/I places it in the correct sequence in the data base by using the key you supply in the I/O area. If the segment you are inserting is not a root, but you have just inserted its parent, you can insert the child segment by issuing an ISRT call with an unqualified SSA. You must build the new segment in your I/O area before you issue the ISRT call. You also use an unqualified SSA when you insert a root. When you are adding new segment occurrences to an existing data base, the segment type must have been defined in the DBD. You can add new segment occurrences directly or sequentially after you have built

them in the program's I/O area. At least one SSA is required in an ISRT call; the last (or only) SSA specifies the segment being inserted. To insert a path of segments, you can set the D command code for the highest level segment in the path.

If the segment type you are inserting has a unique key field, where DL/I adds the new segment occurrence depends on the value of its key field. If the segment doesn't have a key field, or if the key is not unique, you can control where the new segment occurrence is added by specifying either the FIRST, LAST, or HERE insert rule. The rules are specified on the RULES parameter of the DBD generation for this data base. These rules are as follows:

FIRST DL/I inserts the new segment occurrence before the first existing occurrence of this segment type. If this segment has a nonunique key, DL/I inserts the new occurrence before all existing occurrences of that segment that have the same key field.

LAST DL/I inserts the new occurrence after the last existing occurrence of the segment type. If the segment occurrence has a nonunique key, DL/I inserts the new occurrence after all existing occurrences of that segment type that have the same key.

HERE If HERE is specified, DL/I assumes you have a position on the segment type from a previous DL/I call and places the new occurrence before the segment occurrence that was retrieved or deleted by the last call—in other words, immediately before current position. If current position is not within the occurrences of the segment type being inserted, DL/I adds the new occurrence before all existing occurrences of that segment type. If the segment has a nonunique key and current position is not within the occurrences of the segment type with equal key value, DL/I adds the new occurrence before all existing occurrences that have equal key fields.

If HERE has been specified, and you use an F or L command code in the SSA, the command code will override the insert rule and determine where the new segment will be added. This is also true if the insert rule is first, and you use the L command code.

ISRT (add) Example

This is an example of using the ISRT call and the medical data base.

"Add information to the record for Chris Edwards about his visit to the clinic on February 1, 1980. His patient number is 02345. He had a sore throat."

First you need to build the ILLNESS segment in your program's I/O area. Your I/O area for the ILLNESS segment would look like this:

```
02011980SORETHROAT
```

The call you would use to add this new segment occurrence to the data base is:

```
ISRT   PATIENTb(PATNObbb=b02345)
       ILLNESSb
```

You would reference the I/O area that contains the ILLNESS segment in your call. Notice that the ILLNESS SSA must be unqualified.

Using SSAs with ISRT

An ISRT call must have at least one unqualified SSA for each segment being added to the data base. Unless the ISRT is a path call, the lowest level SSA specifies the segment being inserted;

this SSA must be unqualified. If you use the D command code, all of the SSAs after the SSA containing the D command code must be unqualified.

You should provide qualified SSAs for higher levels to establish the position of the segment being inserted. Qualified and unqualified SSAs may be used to specify the path to the segment, but the last SSA must be unqualified. This final SSA names the segment type to be inserted.

If you supply only one unqualified SSA for the new segment occurrence, you must be sure that current position is at the right place in the data base so that the new segment's logical location in the data base can be found by searching forward or backward in the current record.

If you use multiple SSAs, you can mix qualified and unqualified SSAs, but the last SSA must be unqualified. If the SSAs are unqualified, DL/I satisfies each unqualified SSA with the first occurrence of the segment type, assuming that the path is correct. If you omit SSAs in the path, DL/I generates internal SSAs based on current position—just as though you had included the U command code for those levels. If a higher level SSA has changed current position, DL/I develops the missing SSAs for the first occurrence of the segment types that fall within the new path.

One of the best ways to use SSAs with ISRT is to check for the parent segments of the new segment you want to insert. You can't add a segment unless all its parent segments exist in the data base. To check for the parents, you don't have to issue calls for them.

Instead, you can define a fully qualified set of SSAs for all of the parents and issue the ISRT call for the new segment. If DL/I returns a GE status code, at least one of the parents doesn't exist. You can then check the segment level number in the DB PCB to find out which of the parents is missing. If the level number in the DB PCB is 00, DL/I didn't find any of the segments you specified. An 01 means that DL/I found only the root segment; an 02 means that the lowest level segment that DL/I found was at the second level; and so on.

ISRT and Command Codes

You can use all the command codes except N and P with ISRT.

ISRT and Parentage

The only time an ISRT call affects parentage occurs when you insert a segment that is not a dependent of the parent that was established when you issued the call. In this case, the ISRT call destroys parentage. If you issued a GNP call after this kind of an ISRT, DL/I would return a GP status code. GP means that no parentage has been established.

ISRT Status Codes

These are some of the status codes that apply to ISRT:

GE DL/I did not find the path you specified with multiple SSAs. This status is usually returned after a call with one or more qualified SSAs that define the path to the insertion. GE can also be returned when only unqualified SSAs are used if no parent exists.

II The segment you're trying to insert already exists in the data base. This can be returned if you haven't established a path for the segment before trying to insert it. The segment you're trying to insert might match a segment with the same key in another hierarchy or data base record.

VI You gave an invalid length for the variable-length segment you were trying to insert.

INITIALLY LOADING A DATA BASE

Once the DBA has defined the data base, you load the data base by writing an application program using the ISRT call in load mode. You specify load mode by specifying "L" as the processing option for the program in the PROCOPT parameter in the PSB for the program. The only time you use this processing option is when you initially load a data base. The FIRST, LAST, and HERE insert rules do not apply when you are loading a data base—unless you're loading an HDAM data base. In this case, the rules determine how segments with nonunique sequence fields will be chained.

A data base load program builds each segment in the program's I/O area, then loads it into the data base by issuing an ISRT call for it. The ISRT call is the only DL/I call a data base load program issues. If you're using HSAM, you are loading an output data base as you do this. A load program must be a batch program.

Most comprehensive data bases are loaded in stages by segment type or by groups of segment types, since there are usually too many segments to load using only one application program. This means that you need several programs to do the loading. Each load program after the first load program is technically an "add" program, not a load program. If you are writing an add type of load program to load a data base, be sure to have the DBA review the program to make sure that the program's performance will be acceptable. It usually takes longer to add a group of segments than to load them.

For HSAM, HISAM, and HIDAM, the root segments that the application program inserts must be presorted by key fields of the root segments. The dependents of each root segment must follow the root segment in hierarchic sequence, and key values within segment types. In other words, you insert the segments in the same sequence in which your program would retrieve them if it retrieved them sequentially. Other than the fact that you must load segments in hierarchic sequence (children after their parents, data base records in order of their key fields), parentage has no significance in a load program.

If you're loading a HDAM data base, you don't have to presort root segments by their key fields.

Using SSAs in a Load Program

When you are loading segments into the data base, you don't have to worry about position, because DL/I inserts one segment after another. The most important part of loading a data base is the order in which you build and insert the segments. This is the DBA's responsibility.

Because you don't have to worry about position in a data base load program, you don't have to use SSAs for the parents of the segment you're inserting; you don't have to worry about establishing position. You can use SSAs, just as long as any qualified SSAs use only the equal (EQ, =b, or b=) relational operator. You must also use the key field of the segment as the comparative value.

The only SSA you have to supply is the unqualified SSA giving the name of the segment type you're inserting.

For HISAM and HIDAM, the key X'FFFF' is reserved for IMS/VS. IMS/VS returns a status code of LB to you if you try to insert a segment with this key.

Loading a Sequence of Segments

You can insert a path of segments with one ISRT call by concatenating the segments in the I/O area and supplying DL/I with a list of unqualified SSAs. You must include the D command code with the first SSA. The path that the SSAs define must lead down the hierarchy, with each segment in the I/O area being the child of the one before it.

When you build segments in the I/O area, segments must be in the order in which you're inserting them. If the segment has a key field, the key must be in the correct location within the segment in the I/O area. Segments that contain keys must be loaded in the sequence of their keys. If you load segments without keys, DL/I loads them into the data base in the order in which you supply them.

Load Command Codes

The only command code you can use in a load program is D. You use this to load a sequence of segments with one ISRT call.

Status Codes for Load Programs

The status codes below are important to a load program. Like other DL/I calls, a successful ISRT call in a load program receives a blank status code.

LB The segment you are trying to load already exists in the data base. DL/I only returns this status code for segments with key fields.

LC The segment you are trying to load is out of key sequence.

LD No parent exists for this segment. This status code usually means that the segment types you're loading are out of sequence.

LE In an ISRT call with multiple SSAs, your SSAs are out of sequence.

VI You've supplied a variable-length segment whose length is invalid.

DETERMINING YOUR POSITION IN THE DATA BASE

Positioning means that DL/I keeps track of your place in the data base after each call that you issue. If DL/I didn't do this for you, you would have to start from the beginning of the data base each time you retrieved or updated a segment; you would not be able to process one segment after the other without starting from the beginning of the data base each time you issued a retrieval call.

There are three ideas about positioning that you need to understand:

- When it's important, or, in other words, what calls are affected by where your position in the data base is **before** you issue the call
- Where position is after any kind of successful call—in other words, how each type of DL/I call affects current position **after** you issue the call
- Where position is after an unsuccessful call—this is important when you issue a retrieval call and DL/I can't find the segment you've specified; and when you issue an ISRT call with qualified SSAs to define the path to the new segment, and

DL/I can't find one of the segments in the path. These are not-found calls—calls that receive a GE status code.

WHEN POSITION IS IMPORTANT

Position is important to you when you process the data base sequentially—when you issue GN and GNP calls, and GHN and GHNP. Current position is the starting place of DL/I's search for the segments that you specify in these calls.

Before you've issued the first call to the data base, current position is immediately preceding the first root segment occurrence in the data base. Saying that current position is immediately preceding this segment means that if you issue an unqualified GN call, DL/I retrieves the first root segment occurrence. Current position is the place just before the segment occurrence that DL/I would retrieve if you immediately issued an unqualified GN call; it's the next segment occurrence in the hierarchy defined by the DB PCB you referenced.

When you issue a GU, your current position in the data base doesn't affect the way that you code the GU or the SSAs you use. If you issue the same GU at different points during program execution (when you have different positions established), you'll receive the same results each time you issue the call. If you've coded the call correctly, DL/I will return the segment occurrence you requested regardless of whether the segment is before or after current position. (There is an exception: if the GU doesn't have SSAs for each level in the call, it is possible for DL/I to return a different segment at different points in your program. This is based on the position at each level, described later in this section.)

For example, suppose you issue the call below against the data structure that's shown in Figure 53. This structure contains six segment types: A, B, C, D, E, and F. Figure 53 shows one data base record—the root of the record is A1.

```
GU   Abbbbbbb(AKEYbbbb=bA1)
     Bbbbbbbb(BKEYbbbb=bB11)
     Dbbbbbbb(CKEYbbbb=bD111)
```

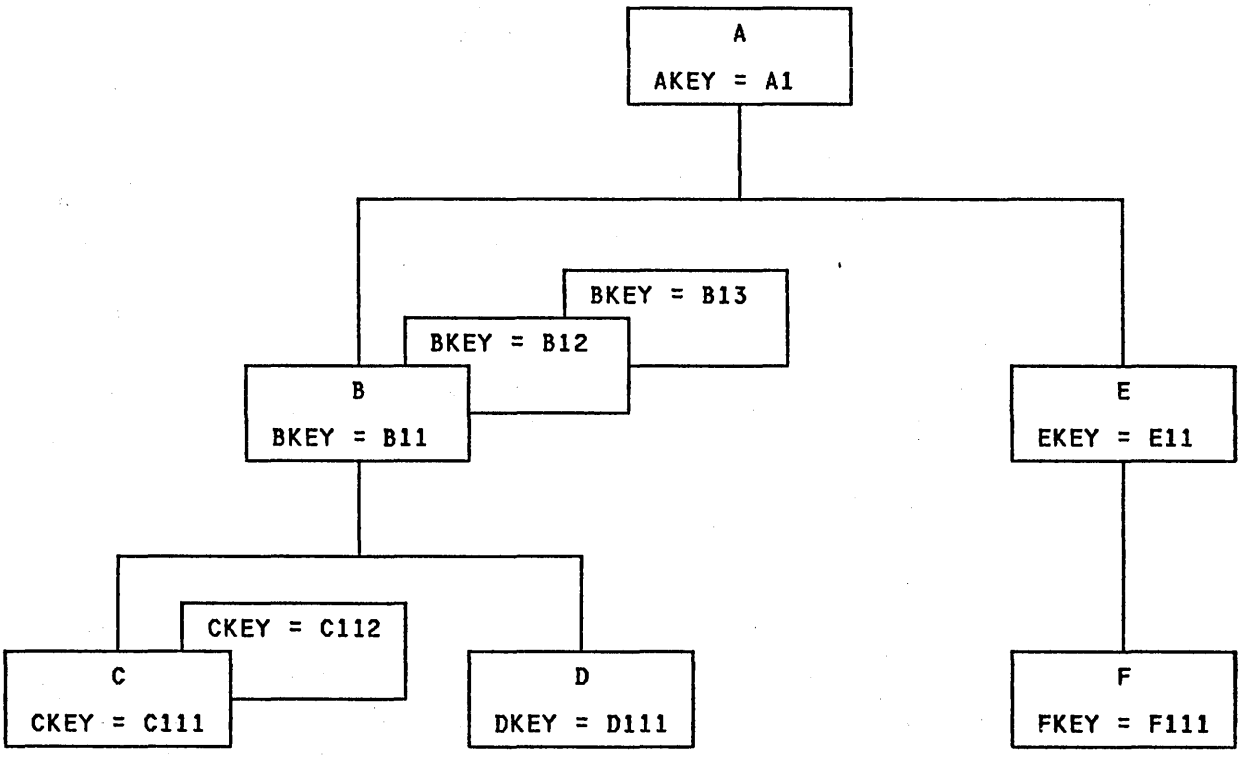

Figure 53. Current Position Hierarchy

When you issue this call, DL/I returns the D segment with the key of D111, regardless of where your position is when you issue the call. If this is the first call your program issues (and if this is the first data base record in the data base), current position before you issue the call is immediately before the first segment occurrence in the data base—just before the A segment with the key of A1. But even if current position is past segment D111 when you issue the call, for example, just before segment F111, DL/I still returns the segment D111 to your program. This is also true if current position is in a different data base record.

But when you issue GN and GNP calls, current position in the data base does affect the way that you code the call and the SSAs. That's because, when DL/I searches for a segment described in a GN or GNP call, DL/I starts the search from current position and can only search forward in the data base—DL/I can't look behind that segment occurrence to satisfy a GN or GNP. These calls can only move forward in the data base when trying to satisfy your call—unless you use the F command code. "Retrieving and Inserting the First Occurrence: F" explains how you use the F command code.

If you issue a GN call for a segment occurrence that you've already passed, DL/I starts searching at current position and stops searching when it reaches the end of the data base (status code GB); or when it determines, from your SSAs, that it can't find the segment you've requested (status code GE). "Current Position after Unsuccessful Calls" explains where your position is when your receive a GE status code.

Current position affects ISRT calls when you don't supply qualified SSAs for the parents of the segment occurrence that you're inserting. If you supply only the unqualified SSA for the segment occurrence you're inserting, then you have to be sure that your position in the data base is in the right place for the segment occurrence to be added in the right place.

CURRENT POSITION AFTER SUCCESSFUL CALLS

In order to process a data base sequentially, you need to understand where current position is after you issue each type of data base call.

Position after Retrieval Calls

After you issue any kind of successful retrieval call, position is immediately after the segment occurrence you just retrieved—or the lowest segment occurrence in the path, if you retrieved several segment occurrences using the D command code. When you use the D command code in a retrieval call, a successful call is one that DL/I completely satisfies.

For example, if you issue the call below against the data base shown in Figure 53, DL/I returns the C segment occurrence with the key of C111. If you then issue an unqualified GN, DL/I returns the C112 segment to your program.

```
GU   Abbbbbbb(AKEYbbbb=bA1)
     Bbbbbbbb(BKEYbbbb=bB11)
     Cbbbbbbb(CKEYbbbb=bC111)
```

Your current position is the same after retrieving segment C111, regardless of whether you retrieve it with a GU, a GN, or a GNP call (or any of the get hold calls).

If you retrieve several segment occurrences by issuing a get call with the D command code, current position is immediately after the lowest segment occurrence that you retrieved. If you issue the same GU call that was shown above, but you include the D command code in the SSAs for segments A and B, current position is still immediately after segment C111. C111 is the last segment that DL/I retrieves for this call. With the D command code, the call looks like this:

```
GU   Abbbbbbb*D(AKEYbbbb=bA1)
     Bbbbbbbb*D(BKEYbbbb=bB11)
     Cbbbbbbb(CKEYbbbb=bC111)
```

You don't need the D on the SSA for the C segment because DL/I always returns the segment occurrence described in the last SSA to your I/O area.

Position after DLET

After a successful DLET call, position is immediately after the segment occurrence you deleted. This is true when you delete a segment occurrence without dependents, and when you delete a segment occurrence with dependents.

For example, if you issue the call shown below to delete segment C111, current position is immediately after segment C111. If you then issue an unqualified GN, DL/I returns segment C112.

```
GHU   Abbbbbbb(AKEYbbbb=bA1)
      Bbbbbbbb(BKEYbbbb=bB11)
      Cbbbbbbb(CKEYbbbb=bC111)
DLET
```

Figure 54 shows what the hierarchy looks like after this call.

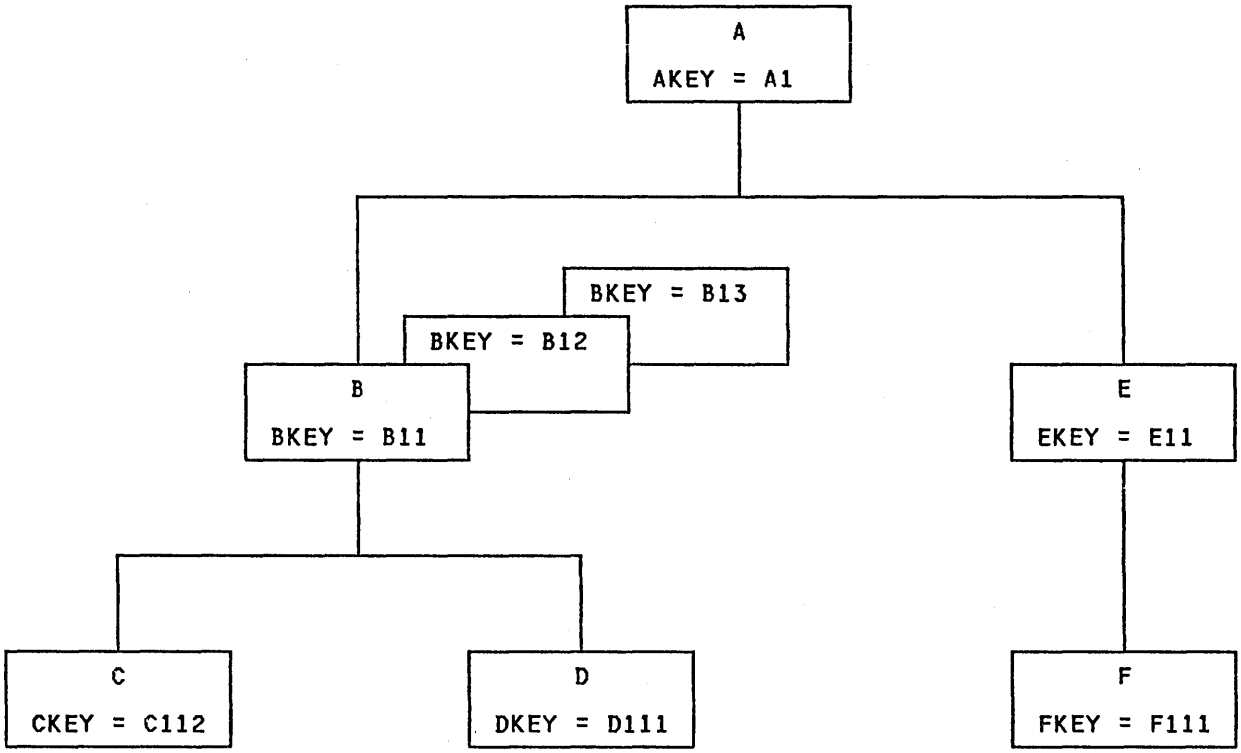

Figure 54. Hierarchy after Deleting a Segment

When you issue a DLET call for a segment occurrence that has dependents, DL/I deletes the dependents, as well as the segment occurrence. Current position is still immediately after the segment occurrence you deleted. An unqualified GN would return the segment occurrence following the segment you deleted.

For example, if you delete segment B11 in the hierarchy shown in Figure 53, DL/I deletes its dependents as well: segments C111, C112, and D111. Current position is immediately after segment B11, just before segment B12. If you then issue an unqualified GN, DL/I returns segment B12. Figure 55 shows what the hierarchy would look like after you issued this call.

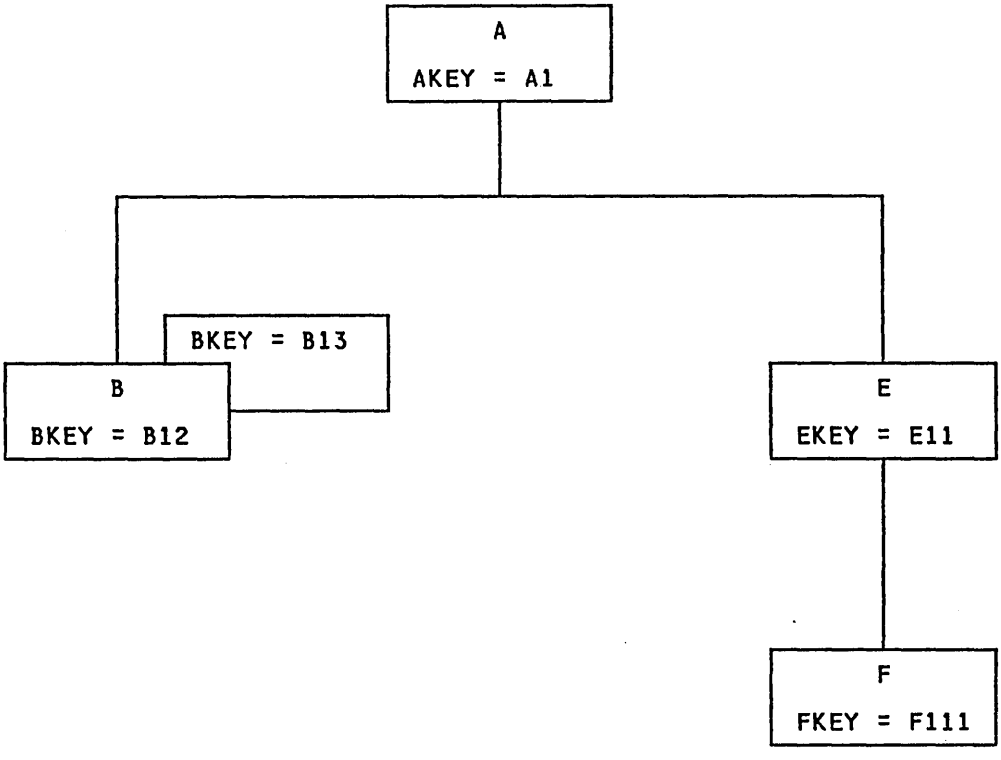

Figure 55. Hierarchy after Deleting a Segment and Dependents

Since DL/I deletes the segment's dependents, you can think of current position as being immediately after the last (lowest, rightmost) dependent. In the example in Figure 55, this is immediately after segment D111. But if you then issue an unqualified GN, DL/I still returns segment B12. You can think of position in either place—the results are the same either way.

Position after REPL

A REPL call doesn't change your position in the data base. After you issue a REPL call, current position is just where it was before you issued the REPL call—immediately after the lowest segment retrieved by the get hold call you issued before the REPL call.

For example, if you retrieve segment C111 using a GHU instead of a GU, change the segment in the I/O area, and then issue a REPL call, current position is immediately after segment C111—just where it is in the retrieval example above.

Position after ISRT

After you add a new segment occurrence to the data base, current position is immediately after the new segment occurrence. For example, if you issue the call below to add segment C113 to the data base, current position is immediately following segment C113. An unqualified GN would retrieve segment D111.

```
ISRT  Abbbbbbb(AKEYbbbb=bA1)
      Bbbbbbbb(BKEYbbbb=bB11)
      Cbbbbbbbb
```

If you're inserting a segment that has a unique key, DL/I places the new segment in key sequence. If you're inserting a segment

that has either a nonunique key, or no key at all, where DL/I places the segment you're adding depends on the rules parameter of the SEGM statement of the DBD for the data base you're processing. "How You Use ISRT to Add Segments"explains these rules.

If you insert several segment occurrences by using the D command code, current position is immediately after the lowest segment occurrence inserted.

Suppose you insert a new segment B (this would be B14), and a new C segment occurrence that's a dependent of B14. Figure 56 shows what the hierarchy looks like after you insert these segment occurrences. The call do to this looks like this:

```
ISRT   Abbbbbbb(AKEYbbbb=bA1)
       Bbbbbbbb*D
       Cbbbbbbbb
```

Notice that you don't need the D command code in the SSA for the C segment—on ISRT calls you only have to include the D in the SSA for the first segment you're inserting. After you issue this call, position is immediately after the C segment occurrence with the key of C141. If you then issue an unqualified GN, DL/I returns segment E11.

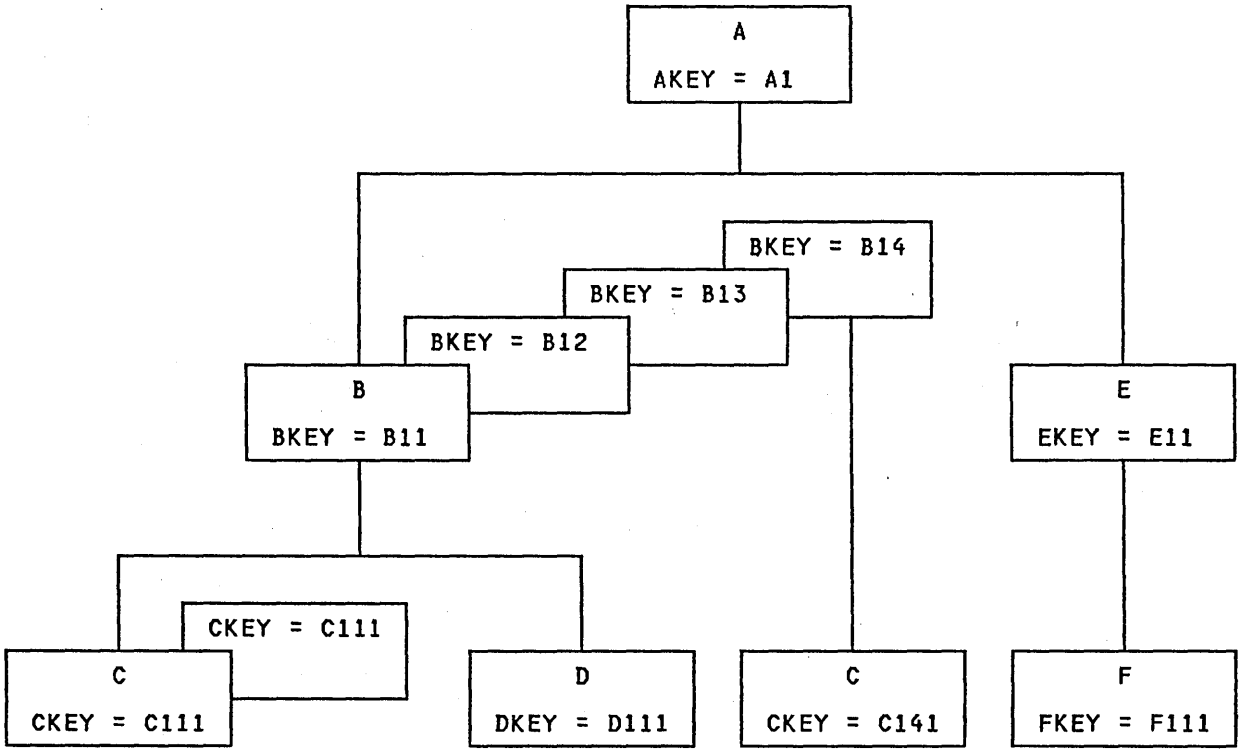

Figure 56. Hierarchy after Adding New Segments and Dependents

The position that's been explained so far is the starting place that DL/I uses for sequential processing. It's called current position. There is another kind of position, in addition to current position, that DL/I establishes when you issue get calls and ISRT calls. This is its position on one segment occurrence at each hierarchic level in the path to the segment you're retrieving or inserting. You need to know how DL/I establishes this position to understand the U and V command codes described in "Using Command Codes," and you need to understand where your position in the data base is when DL/I returns a not-found status code to a retrieval or ISRT call.

CURRENT POSITION AFTER UNSUCCESSFUL CALLS

An unsuccessful DLET or REPL call doesn't affect current position. Your position in the data base after you issue the call is just where it was before you issued the call. But an unsuccessful get call or ISRT call does affect your current position.

To understand where your position is in the data base when DL/I can't find the segment you've requested, you need to understand a little about how DL/I determines that it can't find your segment.

In addition to establishing current position after the lowest segment retrieved or inserted, DL/I maintains a position on one segment occurrence at each hierarchic level in the path to the segment you're retrieving or inserting. When DL/I searches for a segment occurrence to satisfy an SSA, it accepts the first segment occurrence it encounters that satisfies the call. As it does so, it places the key of that segment occurrence in the key feedback are of the DB PCB.

Current position after a retrieval or ISRT call that receives a GE status code depends on how far DL/I got in trying to satisfy the SSAs in the call. When DL/I processes an ISRT call, it checks for each of the parents of the segment occurrence you're inserting. An ISRT call is similar to a retrieval call, in that DL/I processes the call level by level, trying to find segment occurrences to satisfy each level of the call. When DL/I returns a GE status code on a retrieval call, it means that DL/I was unable to find a segment occurrence to satisfy one of the levels in the call. When DL/I returns a GE status code on an ISRT call, it means that DL/I was unable to find one of the parents of the segment occurrence you're inserting. These are called not-found calls.

When DL/I processes retrieval and ISRT calls, it tries to satisfy your call until it can determine that it can't. If, when DL/I first tries to find a segment matching the description you've given in the SSA, there isn't one under the first parent, DL/I will try to search for your segment under another parent. The way that you code the SSAs in the call determines whether or not DL/I can move forward and try again under another parent.

For example, suppose you issue the GN call below to retrieve the C segment with the key of C113 in the hierarchy shown in Figure 57.

```
GN   Abbbbbbb(AKEYbbbb=bA1)
     Bbbbbbbb(BKEYbbbb=bB11)
     Cbbbbbbb(CKEYbbbb=bC113)
```

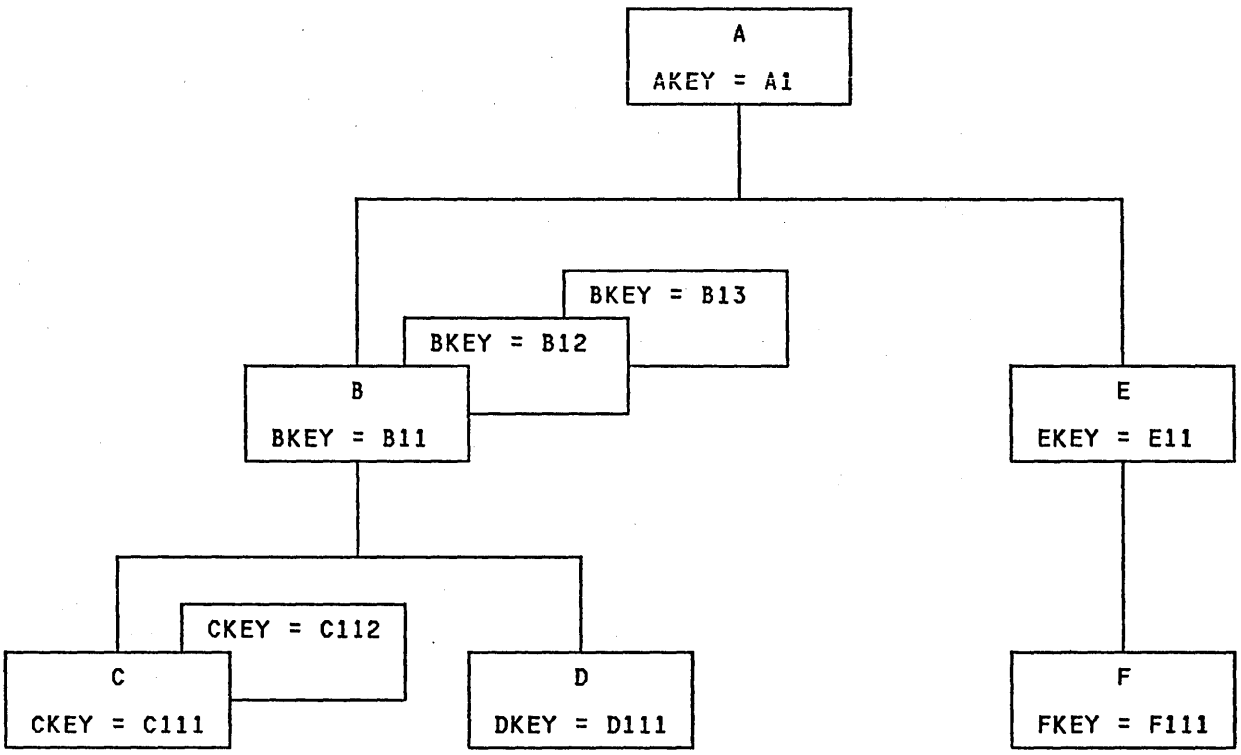

Figure 57. Position after Not Found Calls

When DL/I processes this call, it searches for a C segment with the key equal to C113. DL/I can only look at C segments whose parents meet the qualifications in the SSAs for the A and B segments. The B segment that's part of the path must have a key equal to B11, and the A segment that's part of the path must have a key equal to A1. DL/I looks at the first C segment. Its key is C111. The next C segment has a key of C112. DL/I looks for a third C segment occurrence under the B11 segment occurrence. There are no more C segment occurrences under B11. Because you've specified in the SSAs that the A and B segment occurrences in C's path must be equal to certain values, DL/I can't try to look for a C segment occurrence with a key of C113 under any other A or B segment occurrences. There are no more C segment occurrences under the parent B11; the parent of C must be B11, and the parent of B11 must be A1. DL/I determines that the segment you've specified doesn't exist, and returns a not-found (GE) status code to your program.

When you use an SSA that's qualified on a unique key field with the equal relational operator, DL/I searches for segments to satisfy lower-level SSAs only under the segment occurrence you've described. In this case, DL/I stops searching for C113 when it doesn't find one under B11 because you've told DL/I that the B parent must have a key equal to B11; and that the A parent must have a key equal to A1. If these keys are unique, and DL/I can't find a C113 segment under these parents, Dl/I can't move forward and continue searching under other parents because no other A and B segments can have keys equal to A1 and B11. If these keys are nonunique, DL/I can check for other A and B segments with the key values you specified. If DL/I finds A and B segments with those key values, it continues searching for C113 under those segments.

After this call, current position is immediately after the last segment occurrence that DL/I examined in trying to satisfy your call—in this case, C112. If you then issue an unqualified GN, DL/I rturns D111. The DB PCB key feedback area reflects the

positions that DL/I has established at the levels it was able to satisfy—in this case, A1 and B11.

When you receive the GE status code on this call, you can determine where your position is from the DB PCB key feedback area. If you issue an unqualified GN, you can expect DL/I to return segment D111.

Current position after this call is different if A and B have nonunique keys. Suppose A's key is unique and B's is nonunique. After DL/I searches for a C113 segment under B11 and is unable to find one, DL/I moves forward from B11 to look for another B segment with a key of B11. When DL/I finds there isn't one, DL/I returns a GE status code. Current position is further in the data base than it was when both keys were unique. Current position is immediately after segment B11. An unqualified GN would return B12.

If A and B both have nonunique keys, current position after this call is immediately after segment A1. Assuming there are no more segment A1's, an unqualified GN would return segment A2. If there were other A1's, DL/I would try to find a segment C113 under the other A1's.

But suppose you issue the same call with a greater-than-or-equal-to relational operator in the SSA for segment B:

```
GU   Abbbbbbb(AKEYbbbb=bA1)
     Bbbbbbbb(BKEYbbbb>=B11)
     Cbbbbbbb(CKEYbbbb=bC113)
```

DL/I establishes position on segment A1 and segment B11. Since A1 and B11 satisfy the first two SSAs in the call, DL/I places their keys in the DB PCB key feedback area. DL/I searches for a segment C113 under segment B11. There isn't one. But this time, DL/I can continue searching because the key of the B parent can be greater than or equal to B11. The next segment is B12. Since B12 satisfies the qualification in the SSA for segment B, DL/I places B12's key in the key feedback area. DL/I then looks for a C113 under B12 and doesn't find one. The same thing happens for B13: DL/I places the key of B13 in the key feedback area and looks for a C113 under B13.

When DL/I finds there are no more B segments under A1, it again tries to move forward to look for B and C segments that satisfy the call under another A parent. But this time it can't—the SSA for the A segment specifies that the A segment must be equal to A1. (If the keys were nonunique, DL/I could look for another A1 segment.) DL/I then knows that it can't find a C113 under the parents you've specified and returns a GE status code to your program.

In this example, you haven't limited DL/I's search for segment C113 to only one B segment because you've used the greater-than-or-equal-to operator. DL/I's position is further than you might have expected, but you can tell what the position is from the key feedback area. The last key in the key feedback area is the key of segment B13; DL/I's current position is immediately following segment B13. If you then issue an unqualified GN, DL/I returns segment E11.

Each of the B segments that DL/I examines for this call satisfies the SSA for the B segment, so DL/I places the key of each in the key feedback area. But if one or more of the segments DL/I examines don't satisfy the call, DL/I doesn't place the key of that segment in the key feedback area. This means that DL/I's position in the data base may be further than the position reflected by the key feedback area. For example, suppose you issue the same call, but you qualify the SSA for segment B on a data field, in addition to the key field. To do this, you include another qualification statement in the SSA for segment B. When you use multiple qualification statements in an SSA, you connect them

with a special kind of operator, called a Boolean operator. The two qualification statements in the SSA for segment B shown below are joined with the logical "and" (represented as *). The logical and says that, in order to satisfy the SSA, a segment occurrence must satisfy both qualification statements (as opposed to one or the other). There is another Boolean operator called the logical "or"; it and the logical and are explained in "Using Multiple Qualification Statements."

Assume the data field you're qualifying the call on is called BDATA. For the example, assume the value you want is 14—but that none of the B segments under A1 contains a value in BDATA of 14:

```
GU   Abbbbbbb(AKEYbbbb=bA1)
     Bbbbbbbb(BKEYbbbb>=B11*BDATAbbb=b14)
     Cbbbbbbb(CKEYbbbb=bC113)
```

The way that DL/I processes this call is similar to the way it processes the call above, but there is an important difference: because there is not a B segment greater than or equal to B11 with a data field containing the value 14, the only key that DL/I places in the key feedback area is A1. But when DL/I processes this call, it examines the same segments that it examined in the call above. The difference is that of the segments DL/I examines, only one of them (A1) satisfies an SSA in the call. So that's all the key feedback area contains after this call. But in reality, DL/I's current position is immediately after segment B13; an unqualified GN would return segment E11.

When you use a greater-than or greater-than-or-equal-to relational operator, you don't limit DL/I's search. If you get a GE status code on this kind of call, and if one or more of the segments DL/I examines doesn't satisfy an SSA, DL/I's position in the data base may be further than the position reflected in the DB PCB. If, when you issue the next GN or GNP call, you want DL/I to start searching from the position reflected in the DB PCB key feedback area instead of from its "real" position, you can either:

- Issue a fully qualified GU to reestablish position where you want it.
- Issue a GN or GNP with the U command code. Including a U command code on an SSA tells DL/I to use the first position it established at that level as qualification for the call. It's like supplying a qualified SSA with the equal relational operator for the segment occurrence that DL/I has position on at that level.

For example, suppose that you issue the GU call with the greater-than-or-equal-to relational operator in the SSA for segment B, then you issue this GN call:

```
GN   Abbbbbbb*U
     Bbbbbbbb*U
     Cbbbbbbbb
```

The U command code tells DL/I to use segment A1 as the A parent, and segment B11 as the B parent. DL/I returns segment C111. But if you issue the same call without the U command code, DL/I starts searching from segment B13 and moves forward to the next data base record until it encounters a B segment. DL/I would return the first B segment it encountered.

TECHNIQUES TO MAKE PROGRAMMING EASIER

Deciding how your program will read and update a DL/I data base involves more than deciding which call it will use, and where. How you design your program—in other words, the number, type, and sequence of calls your program issues—can have a significant effect on the efficiency of your program. The number of DL/I calls your program issues and the I/Os that each DL/I call requires are performance considerations.

A program that is poorly designed will run as long as it's coded correctly. IMS/VS won't find design errors for you. The suggestions in this chapter are provided to help you in developing the most efficient design possible for your application program. Inefficiently designed programs can adversely affect performance, and they are hard to change. Being aware of how certain call combinations and call formats affect performance helps you to avoid these problems and design a more efficient program.

In addition to some general guidelines on call sequence, this section tells how the use of some DL/I tools and options affects your program's performance. The tools are SSAs and command codes; the options are multiple PCBs and multiple positioning.

USING SSAS

Using SSAs can simplify your programming, because the more information you can give DL/I so that DL/I does the searching for you, the less program logic you need to analyze and compare segments in your program.

Using SSAs doesn't necessarily reduce the system overhead (for example, internal logic and I/Os) required to obtain a specific segment. To locate a particular segment without using SSAs, you can issue DL/I calls and include program logic to examine key fields until you find the segment you want. By using SSAs in your DL/I calls, you can reduce the number of DL/I calls issued in the programs and the program logic needed to examine key fields. When you use SSAs, DL/I does this for you.

Guidelines on Using SSAs

The list below gives some recommendations about using SSAs. These are not DL/I rules; they are guidelines about using SSAs efficiently in your program.

- Use qualified calls with qualified SSAs whenever possible. This gives DL/I the necessary search information and it is a good way to document your program.
- When you use multiple SSAs in one call, don't leave out SSAs on any levels; this forces DL/I to generate them internally.
- When you use multiple SSAs, give the SSA for the root segment qualified for the key using the equal relational operator. This gives DL/I a specific value instead of forcing DL/I to process a lot of the root segment occurrences.
- It's a good idea, as stated above, to use fully qualified SSAs for all calls, where allowed. This is most important when you're using the ISRT call to add segments. If you don't qualify the ISRT call completely, DL/I can add the segment you've supplied in a position completely different from the correct position. This can happen if another program has changed the hierarchy in some way that affects your call.
- When you use qualified SSAs, use the key field whenever you can in the qualification statement. If you use a data field, there is no sequence for DL/I to use while searching for the segment you want. DL/I then has to search through all occurrences of the segment type you've specified before it can determine whether or not the segment you're requesting exists.

  For example, suppose you want to find the record for a patient by the name of Ellen Carter. As a reminder, the patient segment in the examples contains three fields: the patient number, which is the key field; the patient name; and the patient address. The fact that patient number is the key field means that DL/I stores the patient segments in order of their patient numbers. The best way to get the record for Ellen

Carter is to supply her patient number in the SSA. If her number was 09000, you would use this call and SSA:

```
GU   PATIENTb(PATNObbb=b09000)
```

If you supplied an invalid number, or if someone had deleted Ellen Carter's record from the data base, DL/I would stop searching for the record as soon as it reached a patient number higher than 09000. A higher patient number would be a signal to DL/I that the segment it was looking for did not exist.

If you didn't have the number, however, and had to give the name instead, DL/I would have to search through all the patient segments and read each patient name field until it found Ellen Carter, or until it reached the end of the patient segments.

Using Multiple Qualification Statements

When you use a qualified SSA, you can do more than give DL/I a field value with which to compare the fields of segments in the data base; you can give several field values to establish limits for the fields you want DL/I to compare. There is no set limit for the number of qualification statements you can include in an SSA, but there is a limit on the maximum size of an SSA. You specify this size on the SSASIZE parameter of the PSBGEN statement. For information on this parameter, see the IMS/VS Utilities Reference Manual.

You do this by using qualification statements connected with a special operator. You can indicate to DL/I that you are looking for a value that, for example, is greater than A **and** less than B; or you can indicate that you are looking for a value that is equal to A **or** greater than B. The operators you use to do this are called Boolean operators. They are:

"*" or "&" Logical "and." For a segment to satisfy this request, the segment has to satisfy both qualification statements that are connected with the logical and.

"+" or "|" Logical "or." For a segment to satisfy this request, the segment can satisfy either of the qualification statements that are connected with the logical or.

There is one more Boolean operator: this is the independent "and." It is used with secondary indexes. "How Secondary Indexing Affects Your Program" describes how you use it.

For a segment to satisfy an SSA with multiple qualification statements, the segment can satisfy any set of qualification statements. A set is any qualification statements that are joined by an "and." To satisfy a set, in turn, a segment must satisfy each of the qualification statements within that set. Each "or" starts a new set of qualification statements. When processing multiple qualfication statements, DL/I reads them left to right and processes them in that order.

When you include multiple qualification statements for a root segment, the fields you name in the qualification statements affect the starting point of DL/I's search for your segment. If one or more of the sets don't include at least one statement qualified on the key field of the root segment, DL/I starts searching for a segment that meets the qualification with the first root segment occurrence of the hierarchy. But if all of the sets have one or more statements qualified on the key field of the root segment, then DL/I uses the lowest key field value as the starting place for its search. For example, if you included the statement below, DL/I would start its search with the root segment occurrence with the key of 4:

```
ROOTKEYb>b06*FIELDbbb=bVALUE+ROOTKEYb=04*FIELDbbb=bVALUE
```

When DL/I processes a call containing multiple qualification statements, it searches forward sequentially in the data base, similar to the way it processes GN calls. DL/I examines each root it encounters to determine whether or not the search can continue.

Note: In HDAM, root segments are not stored in key sequence, so using multiple qualification statements for root segments in an HDAM data base may not give the results you want. "HDAM and HIDAM Data Bases" in Chapter 4 (Part 1), "Choosing a Data Base Type," explains this in more detail.

When you use multiple qualification statements segments that are part of logical relationships, there are some additional considerations. "How Logical Relationships Affect Your Programming" explains these considerations.

The easiest way to understand multiple qualification statements is to look at an example:

"Did we see patient #04120 during 1979?"

To find the answer to this question, you need to give DL/I more than the patient's name; you want DL/I to search through the ILLNESS segments for that patient, read each one, and return any that have a date in 1979. The call you would issue to do this is:

```
GU   PATIENTb(PATNObbbEQ04120)
     ILLNESSb(ILLDATEb>=01011979&ILLDATEb<=12311979)
```

In other words, you want DL/I to return any ILLNESS segment occurrences under patient number 04120 that have a date after or equal to January 1, 1979, and before or equal to December 31, 1979.

USING COMMAND CODES

There are nine command codes that you can use in DL/I calls. What they do for you, and when you might want them, really depends on the processing you're doing. Command codes make it possible for you to significantly change how your calls work.

You can use all the command codes except N with the get calls, and you can use all of them except N and P with ISRT. There is only one command code you can use with REPL; and none of the command codes are valid with DLET.

Retrieving and Inserting a Sequence of Segments: D

Using the D command code reduces the number of DL/I calls your program issues, because it lets you retrieve or insert several segments with one get or ISRT call. Calls with the D command code are called "path calls." To use the D command code, your program must have the P processing option specified in its PCB.

RETRIEVING SEGMENTS WITH D: When you use the D command code with retrieval calls, DL/I places the segments that satisfy the SSAs in the retrieval call in your I/O area. The segments in the I/O area are placed one after the other, left to right, starting with the first SSA you supplied. To have DL/I return each segment in the path, you have to include the D command code in each SSA. You can, however, have intervening SSAs without the D command code. You don't have to include the D on the last segment in the path; DL/I always returns the last segment in the path to your I/O area.

The D command code has no effect on DL/I's retrieval logic; the only thing it does is to cause each segment to be moved to your I/O area. If DL/I is unable to find the lowest segment you've requested, DL/I returns a GE (not-found) status code, just as it does if you don't use the D command code and DL/I is unable to find the segment you've requested. This is true even if DL/I reaches the end of the data base before finding the lowest segment

you requested. If DL/I reaches the end of the data base without satisfying any levels of a path call, DL/I returns a GB status code. But if DL/I returns one or more segments to your I/O area and is unable to find the lowest segment requested, DL/I returns a GE status code, even if DL/I has reached the end of the data base.

The advantages of using the D command code are significant enough that even if you're not positive you'll need the dependent segment returned by D, sometimes it's worth using it. For example, suppose there was only about ten percent chance that, after examining the dependent segment, you would need to use the dependent segment. It's still more efficient to issue the D command code so that you have the segment if you need it, than it is to have to issue another call for the segment when you don't use the D command code.

As an example of the D command code, suppose you had this request: "Compute the balance due for each of the clinic's patients by subtracting the payments received from the amount billed and print bills to be mailed to each patient."

To process this request, you need to know, for each patient, the patient's name and address, what the charges are for the patient, and the dollar amount of the payments that the patient has made. You would issue this call until you received a GE status code indicating that there were no more patient segments.

```
GN   PATIENTb*D
     BILLINGb*D
     PAYMENTbb
```

Each time you issued this call, your I/O area would contain the patient segment, the billing segment, and the payment segment for a particular person.

INSERTING WITH D: With ISRT calls, you can use the D command code to insert a path of segments at once. When you're using D with ISRT you don't have to include the D for each SSA in the path; you just specify the D on the first segment that you want DL/I to insert. DL/I then inserts all the segments following that in the path. You cannot use the D command code to insert segments if there is a logical child segment in the path. "Processing Segments in Logical Relationships" explains how logical relationships affect your programming.

Retrieving and Inserting the First Occurrence: F

The F command code is an efficient tool to use, because it limits the search for DL/I. You can use F for GN, GNP, and ISRT calls. (There is no reason to use it with GU because a GU can back up in the data base anyway.) When you use F you are indicating to DL/I that you want the search to start with the first occurrence of the segment type you indicate under its parent in attempting to satisfy this level of the call.

USING F WITH GN AND GNP: When you use the F command code with GN or GNP, you are able to back up in the data base; something you can only do with GU if you don't use F. When you use F, you can back up to the first occurrence of the segment type that has current position, or you can back up to a segment type that is before current position in the hierarchy.

The only restriction is that the segment you're backing up to must be in the same hierarchic path. DL/I disregards F when you supply it at the root level and with a GU or GHU.

USING F WITH ISRT: When you use F with an ISRT (add) call, you are indicating to DL/I that you want DL/I to insert the segment you've supplied as the first segment occurrence of its segment type. You would only use F with segments that have either no key at all or a nonunique key, and that have HERE specified for them on the RULES operand of the SEGM statement in the DBD. If you've specified HERE

in the DBD, the F command code will override this and DL/I will insert the new segment occurrence as the first occurrence of that segment type.

Retrieving and Inserting the Last Occurrence: L

The L command code is the opposite of the F command code: L indicates to DL/I that you want DL/I to retrieve the last segment occurrence, or that you want DL/I to insert the segment occurrence you're supplying as the last occurrence of that segment type. Like F, L simplifies your programming because you can go directly to the last occurrence of a segment type without having to examine the previous occurrences with program logic, if you know that it is the last segment occurrence that you want. You should use the L command code whenever appropriate. Again, DL/I disregards L if you use it for a root segment. L is not disregarded with GU or GHU, because you can use it to indicate to DL/I that you are looking for the last occurrence of a segment type. Without the L, this would not be obvious on a GU.

USING L WITH RETRIEVAL CALLS: Using an L with GU, GN, and GNP indicates to DL/I that you want the last occurrence of the segment type that satisfies the qualification you've provided. The qualification is the SSA: either the segment type, or the qualification statement as well. If you have supplied just the segment type (an unqualified SSA), DL/I retrieves the last occurrence of this segment type under its parent.

For example, suppose you had this request using the medical hierarchy:

> "What was the illness that brought Jennifer Thompson, patient number 10345, to the clinic most recently?"

In this example, assume that RULES=LAST on ILLNESS. You would issue the call below to retrieve this information:

```
GU  PATIENTb(PATNObbb=b10345)
    ILLNESSb*L
```

The first SSA gives DL/I the number of the particular patient that you're working with, and the second SSA asks for the last (in this case, this is the first occurrence chronologically) occurrence of the ILLNESS segment for this patient.

USING L WITH ISRT: You only use L with ISRT when the segment has no key or a nonunique key, and the insert rule for the segment is either FIRST or HERE. L overrides both FIRST and HERE.

Using Concatenated Keys in SSAs: C

You can use the C command code for all of the get calls and the ISRT call. Using C indicates to DL/I that instead of a qualification statement, you're supplying the concatenated key of a segment as a means of identifying it. When you code the concatenated key, you enclose it in parentheses following the *C, in the same position that would otherwise contain the qualification statement.

Using C is sometimes more convenient than a qualification statement because it's easier to just use the concatenated key than to move each part of the qualification statement to the SSA area during program execution. Using the segment's concatenated key is the equivalent of giving all of the SSAs in the path to the segment qualified on their keys.

You can only have one SSA with a concatenated key per call; the SSA containing the concatenated key must be the first SSA in the call. DL/I returns an AM status code if the SSA with the concatenated key is not the first SSA in the call.

Setting Parentage Where You Want It: P

The P command code lets you set parentage at the level at which you want it. Ordinarily, DL/I sets parentage at the level of the lowest segment accessed during a call. When you use P in a retrieval call, you can set parentage at a higher level than the level at which DL/I would set it. You can only use P with the get calls.

The parentage that you set with P works just like the parentage that DL/I sets: it remains in effect for subsequent GNP calls, and is not affected by ISRT, DLET, or REPL calls. It's only affected by GNP if you use the P command code in the GNP call. Parentage is destroyed by a subsequent GU, GHU, GN, or GHN.

Use the P command code at only one level of the call. If you use P in multiple levels of a call by mistake, DL/I sets parentage at the level of the lowest call that includes P.

If DL/I cannot fully satisfy the call that uses P (for example, DL/I returns a GE status code), but the level that includes the P is satisfied, the P is still valid. If DL/I cannot fully satisfy the call including the level that contains the P, then DL/I doesn't set any parentage at all. You would receive a GP (no parentage established) if you then issued a GNP.

If you use P with a GNP call, DL/I processes the GNP call with the parentage that was already set by preceding calls. DL/I then resets parentage with the parentage you specified using P after DL/I has satisfied the GNP call.

As an example of using the P command code, look at the example below. The example uses the P command code with the D command code.

> "Send a current bill to all of the patients we have seen this month."

In this request the determining value is in the ILLNESS segment; you want to look at only patients whose ILLNESS segments have dates after the first of the month. For the patients who have been to the clinic during the month, you need to look at their addresses and the amount of charges in the BILLING segment so that you can print a bill. For this example, assume the date is March 31, 1980. You would issue the two calls below to process this information:

```
GN    PATIENTb*PD
      ILLNESSb(ILLDATEb>=03011980)
GNP   BILLING
```

Once you locate a patient who has been to the clinic during the month, you issue the GNP call to retrieve that patient's BILLING segment. Then you repeat the GN call to find another patient who has been to the clinic during the month until DL/I returns a GB status code.

Using DL/I's Positions as Qualifications: U

As DL/I satisfies each level in a retrieval or ISRT call, DL/I establishes a position on the segment occurrence that satisfies that level. For example, suppose you want to find out about the illness that brought a patient named Mary Warren to the clinic most recently, and about the treatments that she received for that illness. Figure 58 shows the PATIENT, ILLNESS, and TREATMNT segments for Mary Warren.

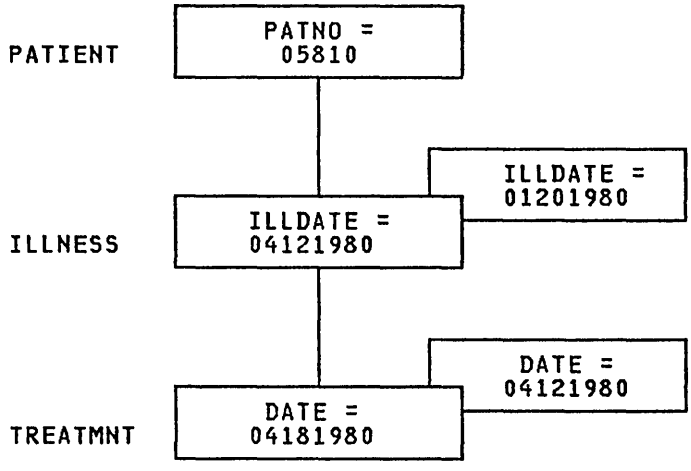

Figure 58. U Command Code Example

To retrieve this information, you need to retrieve the first ILLNESS segment and the TREATMNT segments associated with that ILLNESS segment. To retrieve the most recent ILLNESS segment, you can issue the GU call below:

```
GU   PATIENTb(PATNObbb=b05810)
     ILLNESSb
```

After you issue this call, DL/I has position established at the root level on the PATIENT segment with the key 05810, and on the first ILLNESS segment with the key 04121980. Since it's possible that there are other ILLNESS segments with the key 04121980, you can think of this one as the first. The next thing you want to do is to retrieve the TREATMNT segment occurrences associated with that ILLNESS segment. You can do this by issuing the GN call below with the U command code:

```
GN   ILLNESSb*U
     TREATMNTb
```

In this example, the U command code indicates to DL/I that you want only TREATMNT segments that are dependents of the ILLNESS segment on which DL/I has established position. Issuing the GN call above the first time retrieves the TREATMNT segment with the key of 04181980, and issuing the GN call the second time retrieves the TREATMNT segment with the key 04121980. If you issue the call a third time, DL/I retrieves a not-found status code. The U command code tells DL/I that, if it does not find a segment that satisfies the lower qualification under this parent, it cannot continue looking under other parents.

Qualifying the Search on the Current Path: V

When you use the V command code, it's as though you had included a U command code in each of the SSAs above the SSA that includes the V. You use V to indicate to DL/I that you want each of the segment occurrences in the path to the segment you're retrieving or inserting to be part of the qualification for that segment.

Preventing a Segment from Being Replaced: N

You use the N command code only with REPL calls. You use it when you are going to replace a path of segments that you have retrieved with a get hold call. If there is a segment in the path of segments that you do not want to replace, you can use the N

command code to indicate to DL/I which of the segments you haven't changed, and therefore don't need to be replaced. The N command code is a good performance tool because, if you are not going to change any of the segments in the path, you can indicate this to DL/I so that DL/I doesn't needlessly replace the same segment.

Reserving a Place for Command Codes: Null

There is one more value that you can use as a command code in SSAs: null. If you use a null ("-") in an SSA with which you normally use a command code, you can subsequently substitute the command code you want without having to change the SSA. This method simplifies maintenance of SSAs using command codes, because you can set aside a fixed number of bytes for command codes. Then, you can turn them on and off by using the hyphen.

USING PARALLEL PROCESSING

Sometimes during your processing, you need to maintain two or more independent positions in one hierarchic path or one data base record. For example:

- Suppose you have to reread a segment before replacing it and you have to access another segment between the get hold call and the REPL call. If you don't have a way to keep both places in the data base (and to save the get hold call so that DL/I doesn't cancel it), you have to reissue the get hold call before you can successfully issue the REPL call.
- Suppose you need to save your place in multiple data base records. You need to look at the data in one segment before you know what segment you want to look at next. If you can't hold your place in both records, then you have to continually save the segments and reissue calls for them.

Using one of the methods below saves you from having to keep issuing GU calls to switch back and forth from one data base record or hierarchic path to another in order to reestablish position.

Using Multiple DB PCBs

The easiest way to do this is to use two or more PCBs for the same hierarchic view. When a program has multiple PCBs, it usually means that the program accesses several data bases, but this can mean that you need several positions in one data base record. To DL/I, each time you reference a different PCB, this is like referencing a different data base, so DL/I maintains a position for you in each data base represented by a PCB.

Multiple PCBs can keep your position in more than one hierarchic path and in more than one data base record. When you use multiple PCBs, you can save your place in one data base record or hierarchic path by using the second PCB in the next call. For example, suppose you were processing the data base record for Patient A, then you wanted to look at the record for Patient B, but be able to come back to your position for Patient A. If your program was using multiple PCBs for the medical data base, you would issue the first call for Patient A using PCB1, then issue the next call, for Patient B, using PCB2. To return to Patient A's record, you would issue the next call using PCB1 and you would be back where you left off in that data base record.

Using Multiple Positioning

The other way to save your place in a hierarchic path is to use multiple positioning. Multiple positioning is an option that you can specify on the POS parameter of the PCB statement in the PSB for your application program. One of the main differences between

using multiple PCBs and multiple positioning is that with multiple PCBs you can maintain separate positions in two or more data base records or in two or more hierarchic paths; but with multiple positioning you can keep multiple positions only in multiple hierarchic paths. You can't keep separate positions in multiple data base records.

You should always use qualified SSAs with multiple positioning because without them you can't indicate to DL/I which path contains the segment you want. If you want to sequentially process the segments in the hierarchy, but are using multiple positioning, DL/I will satisfy each subsequent GN call by using the current position established by the previous call.

You can reset the position for any path by issuing a GU call to a new root segment, or, to reestablish position in the data base record you've been processing, to the root segment you've been using.

PROGRAMMING GUIDELINES

Once you have a general call sequence mapped out for your program, look over the guidelines on call sequence in this section to see if you can improve the sequence. Sometimes you can find a way to avoid issuing one or two calls in your program by making better use of command codes, the PCB mask, and parallel processing. Usually an efficient call sequence causes efficient internal DL/I processing.

- Use the simplest call. Qualify your calls to narrow the search for DL/I, but don't use more qualification than you have to. For example, Boolean operators and some of the command codes increase the complexity of your program without noticeably improving its efficiency.
- Always use the call or sequence of calls that will give DL/I the shortest path to the segment you want.
- Use the fewest number of calls possible in your program. Each DL/I call your program issues uses system time and resources. The first thing to do when you look at call sequence is to determine whether or not you can eliminate any calls. These are some ways in which you may be able to do this:
  - Use the D command code if you're retrieving more than one segment in the same path. If you are using more than one call to do this, you are issuing unnecessary calls. The D command code retrieves multiple segments with only one retrieval call. The same goes for inserting, replacing, and deleting segments. When you insert segments, use the D command code to insert a whole path of segments. If you have to delete or replace a path of segments, retrieve them with a get hold path call, then issue one DLET or REPL call to update them. Use the N command code if you need to.
  - If your program retrieves the same segment more than once during program execution, you are issuing an unnecessary call. You can change the sequence so that your program saves the segment in a separate I/O area, then gets it from that I/O area the second time it needs the segment.
  - Anticipate and eliminate needless and nonproductive calls, such as calls that result in GB, GE, and II status codes. For example, if you're issuing GNs for a particular segment type and you know how many occurrences there are of that segment type, don't issue the GN that will result in a GE status code. You can keep track of the number of occurrences your program retrieves, then continue with other processing when you know you have retrieved all of the occurrences of that segment type.

- If you retrieve a dependent segment and its parents, but you only need the keys of the parents (instead of the whole segments), all you have to do is to issue a call for the dependent. DL/I returns the concatenated key for each segment to the key feedback area of the PCB. To use the keys of the parents, you have to know the length of each of the keys so that you can know where one key stops and the next key begins.

- When you're inserting segments, you can't insert dependents unless the parents exist. But instead of issuing get calls for the parents to make sure that they exist, you can save calls by issuing an ISRT call with a fully qualified SSA for each parent. If DL/I returns a GE status code, at least one of the parents doesn't exist. You can then check the PCB mask to determine which parent doesn't exist.

- If you're trying to find a particular segment, use qualified SSAs instead of retrieving several segments and using program logic to find out if that's the segment you want. It's more efficient to let DL/I do as much of the searching as possible for you.

- Keep the main section of the program logic together. For example, branch to conditional routines, such as error and print routines, in other parts of the program, instead of having to branch around them to continue normal processing.

- Use call sequences that make good use of the physical placement of the data. In other words, access segments in hierarchic sequence as much as possible. Avoid moving backward in the hierarchy. To use the physical placement of segments in the data base, you need to know these things about the data base:

  - data base volumes

  - dependent segment frequencies

  - where logical relationships exist and what the logical linkage paths are

- Process data base records in order of the key field of the root segments. (For HDAM data bases, this order depends on the randomizing routine that is used. Check with your DBA for this information.)

- Try to avoid constructing the logic of the program and the structure of calls in a way that depends heavily on the data base structure. Depending on the current structure of the hierarchy takes a lot of flexibility away from the program.

- If your program calls other programs that are OS/VS subtasks of your application program, these programs should not issue DL/I calls. Attaching a region controller to an application program is your installation's responsibility.

CHECKING STATUS CODES

To give you information about the results of each call, DL/I places a 2-character status code in your program's PCB after each DL/I call your program issues. Your program should check the status code after every DL/I call it issues. If it doesn't, it can continue processing even though the last call caused an error. How your program tests the status code depends on the type of call just issued. What you want to do is to test for the status codes that indicate exceptional conditions. There are some status codes that you can expect for certain calls. If you find that DL/I has returned a status code code other than one you had expected, you should branch to an error routine.

When DL/I returns two blanks after a call (indicated as "bb" in this manual) the call was completely successful. Your program should always check for this status code before any of the others for every call. It means that you can continue processing without any further action.

EXCEPTIONAL CONDITIONS

Some status codes do not necessarily mean that your call was successful or unsuccessful; they just give you information about the results of the call. Your program will have to use this information to determine what to do next. What these informational status codes are depends on the call.

For example, if your program is sequentially processing the data base using GN calls, you can expect, at some point, to reach the end of the data base. DL/I returns the status code GB to indicate this to you. In this case, your program should test first for blanks, then for a GB in the DB PCB after each GN call.

In a typical program, the status codes that you should test for apply only to the get calls. There are status codes that can indicate exceptional conditions for other calls; the most common exceptional conditions are GA, GB, GE, and GK. When your program is retrieving segments, these are situations that you should expect and for which you should provide other routines than error routines.

GA GA means that DL/I has returned a segment, but that the segment is at a higher level in the hierarchy than the last segment returned was. DL/I returns GA only for unqualified GNs.

GB GB means that DL/I reached the end of the data base while trying to satisfy a GN and did not return a segment to your program's I/O area.

GE GE means that DL/I could not find the segment you asked for. You can receive a GE after a GU, a GN, or a GNP.

GK GK means that DL/I has returned a segment that satisfies an unqualified GN or GNP, but the segment is of a different segment type (but at the same level) than the last segment returned.

What your program does after receiving one of these calls depends on your particular program. If you are processing a data base sequentially and you receive a GB, this tells you that you have finished processing and you would probably want to terminate.

ERROR ROUTINES

If, after checking for blanks and exceptional conditions in the status code, you find that there has been an error, your program should branch to an error routine and print as much information as possible about the error before terminating. You should print the status code as well. Some of the information that can be helpful in finding out about the error is the call that was being executed when the error occurred; the SSAs and command codes, if any, that the call included; and the contents of the DB PCB.

There are two kinds of errors that can occur in your program. The first kind—programming errors—are usually your responsibility; they're the ones you can find and fix. These errors are caused by things like an invalid SSA, an invalid call, or an I/O area that is too long. The other kind of error is not usually something you can fix; this is a system or I/O error. When your program has this kind of error, you will usually have to ask the system programmer or the equivalent specialist at your installation for help.

Because every application program should have an error routine available to it, and because each installation has its own ways of finding and debugging programs, installations usually provide their own standard error routines.

If you don't have one available to you, the assembler language routine in "Appendix E. Sample Status Code Error Routine (DFSOAER)" is provided as a sample routine. This routine is part of the IMS/VS Primer function that is included with IMS/VS.

To call this sample routine, your program issues a call similar to a DL/I call. In the call you must specify a label that identifies the DL/I call that preceded the error call. You can specify nine areas in your program that you want the routine to print. The routine will print the first 76 characters in each area. Your call must specify one area; the other eight are optional.

TAKING CHECKPOINTS

Batch programs should issue checkpoint calls so that they can be restarted from a place other than the beginning of the program. This is important if your program terminates abnormally, or if the system goes down while your program is executing. This is particularly important in long-running batch programs because it means that the program can pick up processing where it left off instead of having to duplicate the processing it has already done. Avoiding duplicate processing in these situations can save a lot of time and resources.

Checkpoint calls must refer to the I/O PCB, so you must have the compatibility option (CMPAT=YES) specified for your program in your program's PSB. When you specify this option, IMS/VS gives your batch program a dummy PCB that acts as an I/O PCB.

The decisions involved in taking checkpoints have effect beyond your own application program. Because checkpoints are a big part of an installation's recovery plan, installations usually establish checkpoint standards for application programs in these areas:

- Which checkpoint call to use—symbolic or basic
- Which areas of your program to checkpoint if you use symbolic checkpoint and restart
- The type of checkpoint ID to use
- How often to issue checkpoints in your program

There are two checkpoint calls you can use: basic and symbolic. The mnemonic for both of them is "CHKP". When you use symbolic CHKP there is a call that you issue to restart your program. The mnemonic for this call is "XRST".

Unless your program accesses OS/VS files that cannot be converted to GSAM, you should use symbolic checkpoint and restart. "Identifying Recovery Requirements"explains what is involved in this high-level decision. This decision, more than any of the others involved in taking checkpoints, is usually an installation standard. Regardless of which checkpoint call you use, you must use only that kind of checkpoint call in your program; you cannot mix basic and symbolic checkpoint calls.

When your program issues either checkpoint call, IMS/VS does three things:

- Writes all modified data base buffers to DASD.
- Writes a log record containing the checkpoint identification given in the call to the system log tape. To print the checkpoint log records, you can use an IMS/VS utility called the File Select and Formatting Print Program (DFSERA10). With

this utility you can select and print log records on the basis of their type, the data they contain, or their sequential positions in the data set. Checkpoint records are type 18 log records. The IMS/VS Utilities Reference Manual describes this program in Chapter 9, "Log Data Formatting Utilities."

- Sends a message containing the checkpoint identification given in the call to the system console operator and to the master terminal operator.

CHECKPOINT IDS

Each checkpoint call your program issues must have an identification, or ID. One of the parameters of both kinds of checkpoint calls is the address in your program of this ID. Checkpoint IDs must be 8 bytes long. Because IMS/VS uses this ID as a means of identifying the checkpoint to the MTO and on the system log, checkpoint IDs should be EBCDIC characters.

When you want to restart your program, you must supply the ID of the checkpoint from which you want the program to be started. This ID is important because when your program is restarted, IMS/VS then reads the system log tape forward and searches for a checkpoint log record with an ID matching the one you have supplied. The first matching ID that IMS/VS encounters becomes the restart point for your program. **This means that checkpoint IDs must be unique both within each application program and among application programs.** If checkpoint IDs are not unique, you can't be sure that IMS/VS will restart from the checkpoint you wanted.

One way to make sure that checkpoint IDs are unique within and among programs is to use IDs made up of the following:

- 3 bytes of information that uniquely identify your program, followed by
- 5 bytes of information to serve as the ID within the program, for example, a value that is incremented by one for each checkpoint call, or a portion of the system time obtained at program start by issuing the TIME macro

WHERE TO USE CHECKPOINTS

The whole idea of issuing checkpoint calls is to indicate to IMS/VS that you have completed a unit of work and have reached a synchronization point in your processing. The best place to issue a checkpoint call is right after you complete a unit of work, before you start the next one. Since a GU call usually starts a new unit of processing, issuing a checkpoint call just before a GU is a good idea. Another reason to issue a checkpoint call just before a GU is that a checkpoint call makes you lose your position in the data base. Your current position in the data base after a successful checkpoint call is just before the first root segment occurrence in the hierarchy. You must reestablish your position in all data bases (except GSAM) after each checkpoint call.

The first call your program issues should be a checkpoint call (or second, if you're using symbolic CHKP and XRST. This is explained below.). If your program doesn't issue a checkpoint call until after it has finished the first unit of work, but then terminates abnormally before it reaches the first checkpoint call, it would have to duplicate the processing it had already done.

HOW OFTEN TO USE CHECKPOINTS

How often you issue checkpoints in your program depends on the type of processing your program does; there are no hard and fast rules, but there are some guidelines. You should specify checkpoint frequency in your program in a way that makes the frequency easy to modify in case the frequency you specify

initially is too high or too low. Some examples of ways to do this are:

- Use a counter in your program to keep track of elapsed time and issue a checkpoint call after a certain time interval.
- Use a counter to keep track of the number of root segments your program accesses. Issue a checkpoint after a certain number of root segments.
- Use a counter to keep track of the number of updates your program has performed. Issue a checkpoint after a certain number of updates.

The main thing to consider when you decide how often to issue checkpoints is how long it would take to back out and reprocess each unit of work if you had to. A general recommendation is one checkpoint call every ten minutes. IMS/VS will back out your program to the specified checkpoint ID, or to the most recent checkpoint if you haven't supplied a checkpoint ID.

SYMBOLIC CHKP

A symbolic CHKP does two things for you that a basic CHKP does not. It can record as many as seven data areas of your program that you specify in the CHKP call, and it works with the XRST call to restart your program.

To indicate to IMS/VS the areas of your program that you want saved, you give the addresses of the areas as parameters in the symbolic CHKP. All of these areas are optional. The address of the I/O area containing the checkpoint ID for the call is required. Some areas you might want to record are counters, control information, and totals from your processing data. IMS/VS records these areas on the system log as special records and restores them to the recorded values when you restart your program. The address of the I/O area containing the checkpoint ID is required on the CHKP call. "Symbolic CHKP and XRST Call Formats" shows how you code the CHKP call.

USING XRST

Only programs that use symbolic CHKP can issue XRST. XRST is required in these programs. The XRST call must be the first call your program issues. This is the only time your program issues it. Batch programs should issue a CHKP call immediately after the XRST call. This is so that is your program terminates abnormally before reaching the first CHKP call, you can still restart and back out the data base updates from the beginning of the program.

When you are starting your program normally, XRST is simply a signal to IMS/VS that you are using symbolic, not basic, CHKP in the program. At this time, the I/O area pointed to in the XRST call must contain blanks. Your program should test this area after issuing XRST. IMS/VS does not change the area when you're starting the program normally.

You should also check the status code in the I/O PCB after issuing the XRST call. The only successful status code for a XRST call is a blank status code. You can also get an AD status code for a restart call. AD means that the call function is invalid. This is the only error status code IMS/VS returns after a XRST call. If IMS/VS detects any other kind of error while processing the XRST call, it terminates your program abnormally.

To restart your program, you have to give IMS/VS the checkpoint ID from which you want the program restarted. You do this in the PARM field of the EXEC statement in the JCL. IMS/VS places this ID in the I/O area pointed to by the XRST call. IMS/VS then reads forward on the system log tape defined in the //IMSLOGR DD statement and searches for the checkpoint records with the ID you

supplied. You must add the //IMSLOGR DD statement to the JCL for the batch region. When it finds this ID, IMS/VS restores the areas you specified in symbolic CHKP and XRST. "Symbolic CHKP and XRST Call Formats" shows how you code the XRST call.

After your program has issued the XRST call and is being restarted, your program must establish position in the data base in order to keep processing. To do this, you can retrieve, from the key feedback area of the DB PCB, the identity of the last segment occurrence that was processed. Then, you can issue a GU, specifying the SSA that points to that occurrence. (This does not apply to GSAM data bases; IMS/VS repositions GSAM data sets for sequential processing.)

It is possible for the segment occurrence identified in the DB PCB to have been deleted by another application program between the time your program terminated abnormally and the time when you restarted your program. If the record is not found, IMS/VS will not prime the key feedback area with the key of the last-processed record. You should include code in your program to handle this situation. For example, if you want to continue by processing the next sequential segment occurrence, you can issue a GU with th SSA qualified with the greater-than-or-equal-to (>= or => or GE) the last-processed key.

If you want to use your own restart method, you can use XRST to reposition GSAM data bases by placing the checkpoint ID in the I/O area before you issue the XRST call. You can supply either the checkpoint ID or the 12-byte YYDDD/HHMMSS ID. If you supply both the parameter specification and the work area specification, IMS/VS will use the parameter specification.

BASIC CHKP

The only reason to use basic CHKP instead of symbolic CHKP is if your program accesses OS/VS files that cannot be converted to GSAM. (Symbolic CHKP can checkpoint GSAM data bases, but not OS/VS files. Basic CHKP is just the opposite.) Symbolic CHKP is better than basic CHKP because basic CHKP does not let you checkpoint additional areas of your program, and it cannot work with the XRST call.

The only way to restart your program with basic CHKP is to use the OS/VS option on the call. This option lets you request an OS/VS checkpoint and subsequently use OS/VS restart. If you do not use OS/VS restart, then you have to supply your own method of restart. You cannot use the XRST call with basic CHKP. "Basic CHKP Call Format" shows how you code the basic CHKP call.

There is a disadvantage to using OS/VS restart, however: you cannot change your program between the time it terminates abnormally and the time you restart it. So, if it terminates abnormally because of a programming error, OS/VS checkpoint and restart won't help too much, because you can't fix the problem before restarting the program.

To indicate to IMS/VS that you want to use OS/VS checkpoint, you use one of the parameters on basic CHKP and supply data set DD cards. IMS/VS issues the OS/VS checkpoint of your region, and then takes its own checkpoint. You can specify one or two DD names. The names of the DCBs that IMS/VS supplies are CHKDD and CHKDD2. If you specify both, they are used alternately, giving you the advantage of saving the two most recent checkpoint data sets instead of only the most recent one. Each OS/VS checkpoint record overrides the one preceding it. Batch programs written in assembler can specify their own DCBs.

If IMS/VS receives a return code other than 0 or 4 from OS/VS, your application program will be abnormally terminated with a code of 0475. This means that IMS/VS failed while processing a batch program that requested an OS/VS checkpoint. See the IMS/VS Messages and Codes Reference Manual for more details.

When restarting your application program under OS/VS restart, follow the OS/VS restart procedures with the following restrictions:

- You cannot make program changes between the time of the failure and the restart.
- At restart for VS1, the same address space must be available.
- OS/VS does not reposition DASD files.

Note: If a BMP requesting OS/VS checkpoints uses the OS/VS timer by issuing OS/VS STIMER macros, the program should issue the OS/VS TTIMER CANCEL macro before requesting the OS/VS checkpoint. If the program must restart the timer between OS/VS checkpoints, the program should issue a TTIMER CANCEL macro before each OS/VS checkpoint, and an STIMER macro after each OS/VS checkpoint. If you use the STIMER macro in a BMP, you must also set the STIMER parameter in the IMSBATCH procedure to 0. This turns STIMER off until you reset it in your program. OS/VS2 MVS Supervisor Services and Macro Instructions describes these macros.

In general, IMS/VS application programs should avoid issuing OS/VS STIMER macros. The reason for this is that IMS/VS uses STIMER for control purposes. If an application program issues an STIMER before the time interval set by the IMS/VS STIMER has expired, the application program's STIMER cancels the STIMER issued by IMS/VS. MVS then resets the time interval for IMS/VS, and the results are unpredictable.

If, for some reason, your program must use STIMER, you need to set the STIMER parameter to 0. This means that you don't want IMS/VS to issue STIMERs. When you do this, there is no IMS/VS timing during the scheduling of your application program. For an MPP, you set the STIMER parameters in the DFSMPR procedure; for a BMP, you set the STIMER parameter in the IMSBATCH procedure; and for a Fast Path program, you set the STIMER parameter in the IMSFP procedure.

USING SECONDARY INDEXING AND LOGICAL RELATIONSHIPS

In addition to all of the DL/I calls and system service calls described in this chapter, DL/I has some techniques available that give you more flexibility in how your program views the data. The decision about whether or not to use these options is made by the DBA. Secondary indexing and logical relationships are techniques that can change your application program's data structure. Examples of when you use these techniques are:

- If an application program needs to access a segment type in a sequence other than the sequence specified by the key field, secondary indexing can be used. Secondary indexing can also change the application program's view of the hierarchic structure or access based on a condition in a dependent segment.
- Logical relationships allow an application program to establish relationships between segments in different data bases. The use of logical relationships provides the application program with a logical structure that may contain segments from more than one data base.

What you need to know about each of these techniques, when you design a program that uses them, is why they might be used, and how each of them affects the way you design your program.

HOW SECONDARY INDEXING AFFECTS YOUR PROGRAM

One of the situations in which someone designing a data base might choose to use a secondary index occurs when an application program needs to select data base records in a sequence other than that defined by the root key. Since DL/I stores root segments in the

sequence of their key fields, the program that was not accessing root segments in the order of their key fields would not be a very efficient program. You can index any field in a segment; you do so by defining an XDFLD statement for the field in the DBD for the data base. If the call isn't qualified on the key, but uses some other field, DL/I has to search all of the data base records to find the correct record. With secondary indexing, DL/I can go directly to a record based on a field value that is not the key field. "Using a Different Key" explains why you use secondary indexes and gives some examples. This section explains how secondary indexing affects your programming.

Using SSAs with Secondary Indexes

If your program uses a secondary index, you can use the name of an indexed field to qualify your SSAs. When you do this, DL/I tries to satisfy your call by going directly to the secondary index and finding the pointer segment with the value you specify. Then it locates the segment that the index segment points to in the data base and returns the segment to your program.

To use an indexed field name in the qualification statement of an SSA, follow these guidelines:

- The indexed field must be defined in the DBD for the primary data base. This is done in the XDFLD statement during DBD generation.
- Use the name that has been given on the XDFLD statement as the field name in the qualification statement.
- Specify the secondary index as the processing sequence during PSB generation for your program. This is done by specifying the name of the secondary index data base on the PROCSEQ parameter on the PCB for the data base during PSB generation.

Also, if you modify the XDFLD of the indexed segment (using the REPL call), any parentage that you had established before issuing the REPL call is lost after you issue REPL. The key feedback area is no longer valid after a successful REPL call.

For example, if you index the PATIENT segment on the NAME field, you define it in the DBD for the medical data base. You use the XDFLD statement for this. If the name of the secondary index data base is INDEX, you specify PROCSEQ=INDEX in the PCB. Then, to issue an SSA that identifies a PATIENT by the NAME field instead of PATNO, you use the name that you've specified on the XDFLD statement. If the name of the XDFLD is XNAME, you use XNAME in the SSA. This is illustrated in Figure 59.

In the DBD:	In the PSB:	In the program:
XDFLD NAME=XNAME	PROCSEQ=INDEX	GU PATIENTb(XNAMEbbb=bJBBROKEbbb)

Figure 59. Using an SSA with Secondary Indexing

What DL/I Returns with a Secondary Index

In the example above, the PATIENT segment that DL/I returns to the application program's I/O area looks just as it would if you hadn't used secondary indexing. The DB PCB key feedback area, however, contains something different. The concatenated key that DL/I returns is the same except that, instead of giving you the key of the segment you requested (the key of the PATIENT segment), DL/I gives you the key of the pointer segment (the segment in the INDEX data base). DL/I places this key in the position where the

root key would be located if you had not used a secondary index—in the left-most bytes of the key feedback area. "Using a Different Key" gives some examples of this.

If you try to insert or replace a segment that contains a secondary index source field that is a duplicate of one already reflected in the secondary index, DL/I returns an NI status code to you.

PROCESSING SEGMENTS IN LOGICAL RELATIONSHIPS

Sometimes an application program needs to process a hierarchy made up of segments that already exist in two or more separate data base hierarchies. Logical relationships make it possible to establish hierarchic relationships between these segments. When you use logical relationships, the result is a new hierarchy—one that doesn't exist in physical storage, but can be processed by application programs as though it does. This type of hierarchy is called a logical structure.

An advantage of using logical relationships is that programs can access the data as though it exists in more than one hierarchy, but it's only stored in one place. An alternative to using logical relationships when two application programs need to access the same segment through different paths is storing the segment in both hierarchies. The problem with this is that you have to update the data in two places to keep it current.

Processing segments in logical relationships isn't very different from processing other segments. "Creating a New Hierarchy: Logical Relationships" explains the application requirements that logical relationships can satisfy. This section uses the example about the inventory application program that's explained in that section to explain how processing segments that are part of logical relationships affects your programming.

In this example, the hierarchy that an inventory application program needs to process contains four segment types:

- An ITEM segment containing the name and an identification number of a medication that is used at a medical clinic
- A VENDOR segment that contains the names and addresses of vendors that supply the item
- A SHIPMENT segment that contains information such as quantity and date for each shipment of the item that the clinic receives
- A DISBURSE segment that contains information about the disbursement of the item at the clinic, such as the quantity, the date, and the doctor that prescribed it

The TREATMNT segment in the medical data base used throughout this chapter contains the same information that the inventory application program needs to process in the DISBURSE segment. Rather than store this information in both hierarchies, you can define a logical relationship between the SHIPMENT segment in the item hierarchy and the TREATMNT segment in the patient hierarchy. Doing this makes it possible to process the TREATMNT segment through the item hierarchy as though it's a child of SHIPMENT. TREATMNT then has two parents: ILLNESS is TREATMNT's physical parent, and SHIPMENT is TREATMNT's logical parent. There are three types of segments involved in a logical relationship. ILLNESS, SHIPMENT, and TREATMNT (called DISBURSE in the item hierarchy) are the three segments that this logical relationship affects. Figure 60 shows the item hierarchy on the right. The SHIPMENT segment points to the TREATMNT segment in the patient hierarchy shown on the left. (The patient hierarchy is part of the medical data base.)

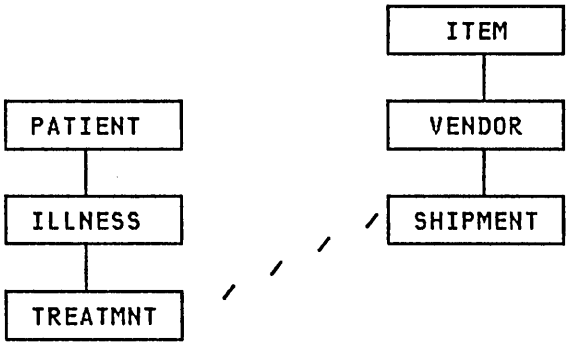

Figure 60. Patient and Item Hierarchies

There are three types of segments in a logical relationship:

- TREATMNT is called a logical child segment. It's a physical dependent of ILLNESS, but it can be processed as though it's a dependent of SHIPMENT. The logical child segment is the segment that can be accessed through both hierarchies, but is stored in only one place.
- ILLNESS is called the physical parent segment. The physical parent is the parent of the logical child in the physical data base hierarchy.
- SHIPMENT is called the logical parent segment. It's the parent of the logical child in the logical structure.

Because a logical child segment has two parents, there are two paths through which a program can access it.

- When a program accesses it through the physical path, that means it reaches it through the segment's physical parent. Accessing the TREATMNT segment through ILLNESS is accessing it through its physical path.
- When a program accesses the logical child through the logical path, that means it reaches the segment through the segment's logical parent. Accessing the TREATMNT segment through SHIPMENT is accessing through its logical path.

How Logical Relationships Affect Your Programming

The calls you issue to process segments in logical relationships are the same calls that you use to process other segments. Processing segments in logical relationships is different from processing other segments in the way the logical segment looks in your I/O area, what the DB PCB mask contains after a retrieve call, and how you can replace, delete, and insert physical and logical parent segments. Because it's possible to access segments in logical relationships through the logical path or the physical path, the segments have to be protected from being updated by unauthorized programs. When the DBA defines the logical relationships, the DBA defines a set of rules that determine how the segments can be deleted, replaced, and inserted. Defining these rules is a data base design decision. If your program processes segments in logical relationships, you should have the following information from the DBA (or the person at your installation responsible for data base design):

- What segments will look like in your I/O area when you retrieve them

- Whether or not your program is allowed to update and insert segments
- What to do if you receive a DX, RX, or IX status code

There is also something you should know about inserting a logical child segment. If you insert the logical child segment through the physical path, your I/O area has to contain the concatenated key of the logical parent, followed by the logical child segment. If you insert the logical child segment through the logical path, your I/O area has to contain the concatenated key of the physical parent, followed by logical child data.

If you use multiple qualification statements in a call to retrieve a concatenated segment, you can include field names in the qualification statements of either of the two physical segments that make up the concatenated segment. But if you include the key field of the logical parent, DL/I treats it as a data field, and does not use it as a sequence field. This is because DL/I doesn't look at the logical parents in sequence when accessing them through a logical child.

Status Codes for Logical Relationships

These are status codes that apply specifically to segments involved in logical relationships. These are not all of the status codes that you can receive when processing a logical child segment or a physical or logical parent; there are others that you can receive for these segments for the same reason that you receive them in other situations (for example, a GE status code). If you receive one of these status codes, it means that you are trying to update the data base in a way that you're not allowed to. You should check with the DBA, or the person responsible for implementing logical relationships at your installation, to find out what the problem is.

DX DL/I did not delete the segment because the physical delete rule was violated. If the segment is a logical parent, is still has active logical children. If the segment is a logical child, it has not been deleted through its logical path.

IX You tried to insert either a logical child segment or a concatenated segment. If it was a logical child segment, the corresponding logical or physical parent segment doesn't exist. If it was a concatenated segment, either the insert rule was physical and the logical or physical parent doesn't exist, or the insert rule is virtual and the key of the logical or physical parent in the I/O area doesn't match the concatenated key of the logical or physical parent.

RX The physical replace rule has been violated. The physical replace rule was specified for the destination parent and an attempt was made to change its data. When a destination parent has the physical replace rule, it can be replaced only through the physical path.

PLANNING AHEAD FOR BATCH-TO-BMP CONVERSION

A batch message program, abbreviated "BMP", is a batch program that runs online. You can have a program that runs as a batch program or a BMP; you just have to code checkpoints in a way that makes them easy to modify because batch programs require fewer checkpoints than BMPs. It has characteristics of both batch programs and message processing programs (MPPs). Like batch programs, a BMP is started by JCL and can access OS/VS files. Like MPPs, BMPs can access the message queues and they access online data bases.

There are two kinds of BMPs: batch-oriented and transaction-oriented. Both can send output to the message queue. A batch-oriented BMP is just a batch program that accesses online

data bases at the same time that other BMPs and MPPs are accessing the same data bases. What makes a transaction-oriented BMP different is that it can access the message queue for its input. This means that an MPP can gather processing requests and send them to a message queue for a BMP to process when all of the requests are collected.

There are a lot of reasons why you might want to convert a batch program to a BMP. Some common reasons are:

- You want to run more than one batch program against the same data base.

  When you run a batch program, there are no other batch programs accessing the data base at the same time. This is because DL/I has no way of protecting the data from the errors that can occur if two programs are updating the data base concurrently. For example, if program A updates segment 1, but terminates abnormally, then program B uses the incorrect data in segment 1, and its results will be incorrect. If you have several batch programs to run, this can take a lot of time.

  When a BMP accesses the data base, however, it does it through the IMS/VS control region. The control region controls which programs access and update which segments. It puts a hold on the segments that a BMP or MPP updates until the BMP or MPP indicates that the results are valid. This means that several programs can access the same data bases at once, because IMS/VS will make sure that program B cannot use segment 1 until program A has indicated that the results are correct. Even if program A terminates abnormally soon afterward, if program A has indicated that its update of segment 1 is accurate, other programs can use this segment.

- There is not enough offline time to run your batch program.

  As an installation develops more and more applications for IMS/VS, sometimes there may be less and less offline time—the time that the online system is not up—to do the batch processing. Converting the batch program to a BMP, so that it too can be run online, solves this problem.

- The batch program needs access to data that is in online data bases.

  A batch program cannot access online data bases. The control region handles all access to online data bases, and batch programs do not use the control region. Converting the batch program to a BMP makes it possible for it to access online data bases.

- Users need the data to be more current.

  Since you can't run a batch program until the online system is down (usually during the night or on weekends), the updates that the batch program performs are not in the data base until that time. This means that people using the data base are using data that is not current. If you convert the batch program to a BMP, however, it's almost like having direct online update, because you can run the BMP as soon as there are enough updates for it to perform.

The main advantages of running a batch program as a BMP are that the data is more current and that you can run several programs at once instead of only one at a time. There is one more advantage; logging and recovery procedures for a BMP are much simpler than they are for a batch program. When a batch program terminates abnormally, it has to be backed out using one of the IMS/VS utilities and its own log tape. But when a BMP terminates abnormally, IMS/VS automatically backs it out using the dynamic log.

Converting a batch program to a BMP involves two additional requirements. The requirements are that you have an I/O PCB for the program by specifying the compatibility option in the PSB for the program and that you use checkpoint calls frequently in your program.

THE COMPATIBILITY OPTION

For a batch program to run as a BMP it must have an I/O PCB. To get an I/O PCB for your program, all you have to do is specify the compatibility option in the PSB for your program. You do this by specifying CMPAT=YES on the CMPAT keyword in the PSBGEN statement of the PSB. It's possible to run your program in a BMP region without specifying this option (the default is CMPAT=NO), but you must have an I/O PCB for checkpoint calls, and a BMP should issue checkpoint calls.

Even if you aren't going to convert your program to a BMP, you should make sure that you have the compatibility option for your program because it is required by checkpoint calls. You must also have an I/O PCB if you want to use the LOG call in your program.

CHECKPOINT FREQUENCY

Checkpoints in BMPs are important for reasons beyond recovery and restart. In a BMP, checkpoint calls indicate to IMS/VS that your program has finished a unit of processing and that the results thus far are accurate. Then, even if your program terminates abnormally, IMS/VS knows that the updates your program has done up to that checkpoint call are accurate and that other programs can use those updates in their processing. "Chapter 10. Structuring and Coding a Batch Message Program"explains additional considerations in checkpointing a BMP.

DESIGNING A PROGRAM THAT USES GSAM

If your program accesses GSAM data bases, there are a few things that you need to take into consideration as you design your program. A DL/I program can retrieve records and add records to the end of the GSAM data base, but the program can't delete or replace any records in the data base. There are separate calls that you use to access GSAM data bases; and there are some additional checkpoint and restart considerations involved in using GSAM. Your program must use symbolic CHKP and XRST if it uses GSAM; basic CHKP cannot checkpoint GSAM data bases.

When a DL/I program uses a GSAM data set, the program treats it like a sequential nonhierarchic data base. Batch programs and BMPs can use GSAM. The OS/VS access methods that GSAM can use are the basic sequential access method (BSAM) on direct access storage devices (DASD), unit record, and tape devices; and the virtual storage access method (VSAM) on DASD devices. VSAM data sets must be nonkeyed, nonindexed, entry-sequenced data sets (ESDS) and must reside on DASD to be used with GSAM. VSAM does not support temporary, SYSIN, SYSOUT, and unit-record files.

Since GSAM is a sequential nonhierarchic data base, it has no segments, keys, or parentage.

ACCESSING GSAM DATA BASES

The calls you use to access GSAM data bases are different from those you use to access DL/I data bases. You can use GSAM data bases for input and output. For example, your program could read input from a GSAM data base sequentially, then load another GSAM data base with the output data. Programs that retrieve input from a GSAM data base usually retrieve GSAM records sequentially and process them. Programs that send output to a GSAM data base must add output records to the end of the data base as the program

processes the records. You cannot delete or replace records in a GSAM data base, and you cannot add records randomly to a GSAM data base; any records that you add to a GSAM data base must go at the end of the data base.

When you issue a GSAM call, IMS/VS first checks to find out whether or not the call is for a GSAM data base. If it is, IMS/VS passes control to GSAM. If it isn't, IMS/VS proceeds with DL/I processing.

PCB Masks for GSAM Data Bases

For the most part, you process GSAM data bases in the same way you process DL/I data bases. You communicate your requests in calls that are very similar to DL/I calls, and GSAM describes the results of those calls in a GSAM DB PCB. The DB PCB mask for a GSAM data base serves the same purpose that it does for DL/I data bases: the program references the fields of the DB PCB for the GSAM data base through the GSAM DB PCB mask. The GSAM DB PCB mask that a program references must contain the same fields as the GSAM DB PCB, and of the same length.

There are some differences between a DB PCB for a GSAM data base and one for a DL/I data base. Some of the fields are different, and the GSAM DB PCB has one field that a DL/I DB PCB doesn't. Figure 61 shows the order and lengths of these fields. Some of the fields in a PCB mask for a DL/I data base don't have meanings in a PCB mask for a GSAM data base. This is because GSAM is a nonhierarchical data base; there are no segments or hierarchic levels in a GSAM data base. The fields that are not used when you access GSAM data bases, but are used when you access DL/I data bases, are: the second field (the segment level number), the sixth field (the segment name), and the eighth field (the number of sensitive segments). Even though GSAM doesn't use these fields, you need to define them in the order and length shown in Figure 61 in the GSAM DB PCB mask.

1. Data Base Name 8 bytes
2. Not Used by GSAM 2 bytes
3. Status Code 2 bytes
4. Processing Options 4 bytes
5. Reserved for DL/I 4 bytes
6. Not Used by GSAM 8 bytes
7. Length of Key Feedback Area and Undefined-Length Records Area 4 bytes
8. Not Used by GSAM 4 bytes
9. Key Feedback Area 8 bytes
10. Undefined-Length Records Area 4 bytes

Figure 61. GSAM DB PCB Mask

When you code a DB PCB mask, you also give it a name, but this is not a field in it. The name you give to the GSAM DB PCB is the name of the area that contains all the fields in the GSAM DB PCB. In COBOL and assembler language programs, you list the names you've given all your DB PCBs in the entry statement; in PL/I programs, you list the pointers to your DB PCBs. DB PCBs don't have names assigned to them in the PSB. In your entry statement, you associate the name in your program with a particular DB PCB based on the order of all of the PCBs in the PSB. In other words, the first PCB name in the entry statement corresponds to the first PCB; the second name in the entry statement corresponds to the second PCB; and so on. The DB PCBs for a particular application program are contained in the PSB for the application program.

A GSAM DB PCB mask contains the following fields:

1. **Data Base Name**

   This is the name of the GSAM DBD. This field is 8 bytes long and contains character data.

2. **Not Used by GSAM**

   This field is not used by GSAM. You should code it, however; it is 2 bytes long.

3. **Status Code**

   DL/I places a 2-character status code in this field after each call to a GSAM or DL/I data base. This code describes the results of the call. DL/I updates it after each call and does not clear it between calls. The application program should test this field after each call to find out whether or not the call was successful. If the call was completely successful, this field contains blanks. This field is 2 bytes long.

4. **Processing Options**

   This is a 4-byte field containing a code that tells DL/I what type of calls this program can issue. It is a security mechanism, in that it can prevent a particular program from updating the data base, even though the program can read the data base. This value is coded in the PROCOPT parameter of the PCB statement when the PSB for the application program is generated. The value does not change.

5. **Reserved for DL/I**

   This 4-byte field is used by DL/I for internal linkage. It is not used by the application program.

6. **Not Used by GSAM**

   This field is not used by GSAM. It should still be coded as part of the GSAM DB PCB mask, however; it is 8 bytes long.

7. **Length of Key Feedback Area and Undefined-Length Records Area**

   This is a 4-byte field that contains the decimal value of 12 in binary. This is the sum of the lengths of the key feedback and the undefined-length record areas described below.

8. **Not Used by GSAM**

   This field is not used by GSAM, but it should be coded as part of the GSAM DB PCB mask. This field is 4 bytes long.

9. **Key Feedback Area**

   After a successful retrieval call, GSAM places the address of the record returned to your program in this field. This is called a record search argument, or RSA; you can use it later on if you want to retrieve that record directly by including it as one of the parameters on a GU call. This field is 8 bytes long.

10. **Undefined-Length Records Area**

    If you use undefined-length records (RECFM=U), the length in binary of the record you're processing is passed between your program and GSAM in this field. This field is 4 bytes long. When you issue a GU or GN call, GSAM places the binary length of the record retrieved in this field. When you issue an ISRT call, you must put the binary length of the record you're inserting in this field before issuing the ISRT call.

Retrieving and Inserting GSAM Records

To retrieve GSAM records sequentially, you use the GN call. The only required parameters on the call are the I/O area for the segment and the GSAM PCB. To process the whole data base, you issue the GN call until you get a GB status code in the GSAM PCB. This means that you have reached the end of the data base. GSAM automatically closes the data base when you reach the end of the data base.

To add new records to the end of the data base you use the ISRT call. GSAM adds the records sequentially, in the order in which you supply them. The output data base that you build can be a data base previously created with BSAM, QSAM, or VSAM. Once you have created a GSAM data base, other programs using these OS/VS access methods can access that GSAM data base.

There is a way to retrieve records directly from a GSAM data base, but before you can ask GSAM to retrieve the record, you have to supply the record's address. To do this, you use something called a record search argument, or RSA. An RSA is similar to an SSA, but it contains the exact address of the record that you want to

retrieve. The specific contents and format of the RSA depend on the access method GSAM is using. For BSAM tape sets and VSAM data sets, the RSA contains the relative byte address (RBA). For BSAM disk data sets, the RSA contains the disk address in the TTR format (track, track, record).

Before you can give GSAM the RSA, you have to get the RSA yourself. To do this, you have to know ahead of time what records you'll want to retrieve directly later on. When you're retrieving records sequentially or adding records to the end of the GSAM data base, you can include a parameter on the GN or ISRT call that tells GSAM to return the address of that record to a certain area in your program. Then you can save this address until you want to retrieve that particular record. At that time, you issue a GU call for the record and you give the address of its RSA as a parameter of the GU call. GSAM returns the record you requested to the I/O area that you have named as one of the call parameters. If you do this at all, you should only do it on DASD with tape accessing.

You can also use a GU call and an RSA to position yourself at a certain place in the GSAM data base. If you place a doubleword of F'1,0' in the RSA and issue a GU using that RSA, GSAM repositions you to the first record in the data base.

Explicitly Opening and Closing a GSAM Data Base

There are two DL/I calls you can use to explicitly open and close a data base: OPEN and CLSE. Although GSAM automatically opens the data base when you issue the first call and closes it when you reach the end of the data or the program terminates, you can use these calls to initiate or terminate data base operations at other times. An application program that loads a GSAM data base can read the data base without terminating between loading the data base and reading it. If you do this, you must issue a CLSE call before you start reading the data base.

When you issue an OPEN call, you have the option of specifying the kind of data set you're using. These are your options:

- INP for an input data set
- OUT for an output data set
- OUTA for an output data set with ASA control characters for a printer
- OUTM for an output data set with machine control characters for a punch

GSAM RECORD FORMATS

If you're using VSAM, you can use fixed- or variable-length records, blocked or unblocked. If you're using BSAM, you can use fixed-length, variable-length, or undefined-length records, blocked or unblocked. Whatever you use, you must specify this on the RECFM keyword in the DATASET statement of the GSAM DBD. You can override this in the RECFM statement of the DCB parameter in the JCL, however; you can include carriage control characters in the JCL for all formats. "GSAM JCL Restrictions"explains what you use to override each type of record format.

GSAM records must be nonkeyed. For variable-length records you must include the record length as the first two bytes of the record. Undefined-length, like fixed-length records, contain only data (and control characters, if you wish). If you use undefined-length records, the record length is passed between your program and GSAM in the 4-byte field in the GSAM DB PCB that follows the key feedback area. This is the tenth field in Figure 61. It is called the undefined-length records area. When you issue an ISRT call, you supply the length; when you issue a GN or GU, GSAM places the length of the returned record in this

field. The advantage of using undefined-length records is that you don't have to include the record length at the beginning of the record, yet your records don't have to be fixed-length. The length of any record must be less than or equal to the logical record length and greater than 11 bytes (the OS/VS convention).

GSAM STATUS CODES

Your program should test the status code after each GSAM call just as it does after each DL/I or system service call. There are two differences between the way DL/I status codes work and the way GSAM status codes work. First, GSAM initializes the status code to blanks before processing each GSAM call; and second, GSAM never returns any data with a nonblank status code.

If, after checking the status code, you find that you have an error and you terminate your program, be sure to save the PCB address before you terminate; the GSAM PCB address will be helpful in problem determination. When a program that uses GSAM terminates abnormally, GSAM issues PURGE and CLOSE calls internally; this changes the PCB pointer.

The status codes that have specific meanings for GSAM are:

AF GSAM detected a BSAM variable-length record that was formatted invalidly. You should terminate your program.

AH You have not supplied an RSA for a GU call.

AI There has been a data management OPEN error.

AJ One of the parameters on the RSA you supplied is invalid.

AM You have issued an invalid request against a GSAM data base; or you issued a call to a GSAM dummy data set.

AO An I/O error occurred when the data set was accessed or when GSAM entered the CLOSE SYNAD routine. This happens when the last block of records was written prior to closing the data set.

GB You reached the end of the data base and GSAM has closed the data base. The next position is the beginning of the data base.

IX You issued an ISRT call after receiving an AI or AO status code. You should terminate your program.

USING SYMBOLIC CHKP AND XRST WITH GSAM

To checkpoint GSAM data bases you must use symbolic CHKP and XRST. This is why most people use GSAM: They can convert OS/VS files to GSAM, then use symbolic CHKP and XRST to make their programs restartable. When you use the XRST call, IMS/VS repositions GSAM data bases for processing. These restrictions apply to restarting GSAM data bases:

- You cannot use temporary data sets with CHKP and XRST calls.
- A SYSOUT data set at restart time can give duplicate output data.
- You cannot restart a program that is loading a GSAM/VSAM data base.

When IMS/VS restores the data areas specified in the symbolic CHKP and XRST calls, it also repositions any GSAM data bases that your program was using when it issued the symbolic CHKP. If your program was loading GSAM data bases when the symbolic CHKP was taken, IMS/VS repositions them if they are accessed by BSAM. IMS/VS does this by internally issuing a GU for the last record

processing at that point. Your program will not receive any data from this GU, but the key feedback area of the GSAM PCB will contain the positioning information you need.

PROCESSING FAST PATH DATA BASES

Fast Path data bases are designed to give you a higher-than-normal performance rate. Because of this, Fast Path data bases are different from DL/I data bases, as is the way that you process them. There are two kinds of Fast Path data bases:

- Main storage data bases, or MSDBs, are data bases that contain only root segments in which you store the data that you access the most frequently. For example, a bank might store various totals, counts and status for all active tellers in an MSDB. The kind of data that you store in an MSDB is data that you reference quite often. The teller data is referenced by all the transactions that involve cash withdrawals and deposits. MSDBs reside in main storage, and thereby reduce I/O activity. By doing this, they can give faster processing than DL/I data bases can.

- Data entry data bases, or DEDBs, are data bases that contain a root segment and one level of dependent segments. A DEDB can have up to seven dependent segment types. You use DEDBs primarily for data collection processes.

Because Fast Path data bases are online data bases, batch programs cannot process them. Both types of Fast Path programs (message driven and nonmessage driven) can process Fast Path data bases. MPPs and BMPs can also process Fast Path data bases. Because MSDBs and DEDBs answer different application processing requirements, you need to understand the differences between them to understand how you process them.

PROCESSING MSDBS

MSDBs contain only root segments; this means that each segment is like a data base record, in that the segment contains all the information about a particular subject. In a DL/I hierarchy, a data base record is made up of a root segment and all its dependents. For example, in the medical hierarchy, a particular PATIENT segment and all the segments underneath that PATIENT segment comprise the data base record for that patient. But in an MSDB, the segment is the whole data base record; the data base record contains only of the fields that the segment contains. MSDB segments are fixed length.

Types of MSDBs

There are two kinds of MSDBs: in one kind of MSDB each segment is "owned" by one logical terminal. The segment that is owned can be updated only by that terminal. This type of MSDB is called terminal related. For clarity, this chapter calls terminal related MSDBs related MSDBs. Further, related MSDBs can be fixed or dynamic. You can add segments to and delete segments from dynamic related MSDBs; you can't add segments to or delete segments from fixed related MSDBs.

The second kind of MSDB is called nonterminal related, or, for purposes of clarity, a nonrelated MSDB. The segments in nonrelated MSDBs aren't owned by logical terminals. One way to understand the differences between these types of data bases, and why you would use each one, is to look at some examples of each.

RELATED MSDBS: One type of data that you might store in a fixed related MSDB is summary data about a particular teller at a bank. For example, you could have an identification code for the

teller's terminal, then you could keep a count of that teller's transactions and the teller's balance for the day. This segment would contain three fields:

- TELLERID: a 2-character code that identifies the teller
- TRANCNT: the number of transactions the teller has processed
- TELLBAL: the balance for the teller

Figure 62 shows what the segment for this type of application could look like.

Figure 62. Teller Segment in Fixed Related MSDB

Some of the characteristics of fixed related MSDBs are:

- You can only read and replace segments; you cannot delete or insert segments. In the bank teller example, a teller would only want to replace values in segments; there would be no need to add or delete segments.
- Each segment is assigned to one logical terminal. Only the owning terminal can change a segment, but other terminals can read the segment. In the bank teller example, you would not want tellers to update the information about other tellers; tellers would be responsible only for their own transactions.
- The name of the logical terminal that owns the segment is the segment's key. Unlike DL/I segments, however, the key isn't a field of the segment. It's used as a means of storing and accessing segments.
- A logical terminal can own only one segment in any one MSDB.

The type of data that you might store in a dynamic related MSDB is summary data about the activity of all the bank tellers at one branch. For example, this segment might contain the following:

- BRANCHNO: the identification number for the branch
- TOTAL: the bank's current balance
- TRANCNT: the number of transactions for that day
- DEPBAL: the deposit balance giving the total dollar amount of deposits for the branch
- WTHBAL: the withdrawal balance, giving the dollar amount of the withdrawals for the branch

Figure 63 shows what this segment could look like.

Figure 63. Branch Summary Segment in Dynamic Related MSDB

Dynamic related MSDBs have the same characteristics as fixed related MSDBs except:

- The owning logical terminal can delete and insert segments.

- The MSDB can have a pool of unassigned segments. This kind of segment is assigned to a logical terminal when the logical terminal inserts it, and is returned to the pool when the logical terminal deletes it.

NONRELATED MSDBS: A nonrelated MSDB is used to store data that is updated by several terminals—such as data that a lot of people need to access and update during the same time period. For example, you might store data about individuals' bank accounts in a nonrelated MSDB segment. This is data that a lot of people might need to access quite often. This segment could contain the following fields:

- ACCNTNO: the account number
- BRANCH: the name of the branch where the account is
- TRANCNT: the number of transactions for this account this month
- BALANCE: the current balance

Figure 64 shows what the segment for this application could look like.

ACCNTNO	BRANCH	TRANCNT	BALANCE

Figure 64. Account Segment in Nonrelated MSDB

The characteristics of nonrelated MSDBs are:

- Segments are not owned by terminals as they are in related MSDBs. This means that IMS/VS programs and Fast Path programs can update these segments; updating them is not restricted to the owning logical terminal.
- You cannot add segments to or delete segments from nonrelated MSDBs.
- Segment keys can be the name of a logical terminal; this is a nonrelated MSDB with terminal-related keys. The segments still aren't owned by the logical terminals; the logical terminal name is used to identify the segment.
- If the key isn't the name of a logical terminal, it can be any value and it's in the first field of the segment. In this case, segments are loaded in key sequence.

Reading Segments in an MSDB: GU and GN

To retrieve segments from an MSDB, you can issue get calls just as you do to retrieve segments from DL/I data bases. Since MSDBs contain only root segments, you don't have any reason to use GNP; you only use GU and GN (and GHU and GHN when you're going to update a segment). Also, you can use only 1 SSA in a call to an MSDB. There are some other differences between calls to DL/I data bases and MSDBs: MSDB calls can't use command codes, nor can they use multiple qualification statements (Boolean operators). (Also, the maximum length for a segment's key in a DL/I data base is 255 bytes. The maximum length for a segment's key in an MSDB is 240 bytes.)

Updating Segments in an MSDB: REPL, DLET, ISRT, and FLD

Three of the calls that you can use to update an MSDB are the same calls that you can use to update a DL/I data base: REPL, DLET, and ISRT. You can issue a REPL call to a related MSDB or nonrelated

MSDB, and you can issue DLET, and ISRT calls to a dynamic MSDB. When you issue REPL or DLET calls against either type of MSDB, you must first issue a get hold call for the segment you want to update, just as you do when you replace or delete segments in DL/I data bases.

There is one call that you can use against MSDBs that you can't use against DL/I data bases. This is the field call, or FLD. The FLD call makes it possible for you to access the contents of a field within a segment, and then to change that field based on its contents. The FLD call does two things for you: it compares the value of a field to the value you supply—this is called FLD/VERIFY—and it changes the value of the field in the way that you specify—this is called FLD/CHANGE. FLD does in one call what a get hold call and a REPL call do in two calls.

For example, using the ACCOUNT segment shown in Figure 64, a bank would have to perform the following processing in order to find out whether or not a customer could withdraw a certain amount of money from a bank account:

1. Retrieve the segment for the customer's account.
2. Verify that the balance in the account is more than the amount that the customer wants to withdraw.
3. If the amount of the balance is more than the amount of the withdrawal, update the balance to reflect the withdrawal.

Without using the FLD call, a program would issue a GU call to retrieve the segment, then verify its contents with program logic, and issue a REPL call to update the balance to reflect the withdrawal. But if you use the FLD call, you use FLD/VERIFY to retrieve the segment and compare the BALANCE field to the amount of the withdrawal. FLD/CHANGE can update the BALANCE field if the compare was satisfactory. Here's how the FLD call works.

CHECKING A FIELD'S CONTENTS: FLD/VERIFY: A FLD/VERIFY call compares the contents of a specified field in a segment to the value that you supply. The way that FLD/VERIFY compares the two depends on the operator you supply. When you supply the name of a field and the value with which you want to compare that field's value, you can compare them in any of the following ways:

- Is the field value equal to the value you've supplied?
- Is the field value greater than the value you've supplied?
- Is the field value greater than or equal to the value you've supplied?
- Is the field value less than the value you've supplied?
- Is the field value less than or equal to the value you've supplied?
- Is the field value not equal to the value you've supplied?

After IMS/VS performs the comparison that you've asked for, IMS/VS returns a status code, in addition to the status code in the PCB, to tell you the results of the compare.

You specify the name of the field and the value to which you want its value compared in a field search argument, or FSA. The FSA is also where IMS/VS returns the status code to you. You place the FSA in an I/O area before you issue a FLD call, then reference that I/O area in the call—just as you do an SSA in a DL/I call. An FSA is similar to an SSA in that you use it to give information to IMS/VS about the information you want to retrieve from the data base. An FSA, however, contains more information than an SSA does—and it does more for you.

Figure 65 shows the structure and format of an FSA.

FLD NAME	SC	OP	FLD VALUE	CON
8	1	1	8	1

Figure 65. FSA Structure

There are 5 fields in an FSA. They are:

- **Field Name**: This is the name of the field that you want to update. The field must be defined in the DBD.
- **Status Code**: This is where IMS/VS returns the status code for this FSA. If IMS/VS successfully processes the FSA, IMS/VS returns a blank status code to the FSA. If not, you receive one of the status codes listed below. If IMS/VS returns a nonblank status code to you in the FSA, IMS/VS returns an FE status code to the PCB to indicate this to you. The FSA status codes that IMS/VS might return to you on a FLD/VERIFY call are:

  B The length of the data that you supplied in "field value" is invalid.

  D The verify check was unsuccessful; in other words, the answer to your question is no.

  E The "field value" contains invalid data. This means that the data you supplied in this field is not the same type of data that is defined for this field in the DBD.

- **Operator**: This tells IMS/VS how you want the two values compared. For a FLD/VERIFY call, you can specify:

  E Verify that the value in the field is equal to the value you've supplied in the FSA.

  G Verify that the value in the field is greater than the value you've supplied in the FSA.

  H Verify that the value in the field is greater than or equal to the value you've supplied in the FSA.

  L Verify that the value in the field is less than the value you've supplied in the FSA.

  M Verify that the value in the field is less than or equal to the value you've supplied in the FSA.

  N Verify that the value in the field is not equal to the value you've supplied in the FSA.

- **Field Value**: This area contains the value that you want IMS/VS to compare to the value in the segment field. The data that you supply in this area must be the same type of data that's in the field you've named in the first field of the FSA. There are five types of data: hexadecimal, packed decimal, alphameric or a combination of data types, binary fullword, and binary halfword. The length of the data that you supply in this area must be the same as the length that's defined for this field in the DBD, with two exceptions:

  - If you're processing hexadecimal data, the data in the FSA must be in hexadecimal. This means that the length of the data in the FSA will be twice the length of the data in the field in the data base. IMS/VS checks the characters in hexadecimal fields for validity before that data is translated to data base format. (Only 0 to 9 and A to F are valid characters.)

- For packed decimal data, you don't have to supply the leading zeros in the field value. This means that the number of digits in the FSA might be less than the number of digits in the corresponding data base field. The data that you supply in this field must be in valid packed decimal format and end in a sign digit.

When IMS/VS processes the FSA, IMS/VS does logical comparisons for alphameric and hexadecimal fields, and arithmetic comparisons for packed decimal and binary fields.

- **Connector:** if this is the only or last FSA in this call, this area contains a blank. If another FSA follows this one, this area contains an asterisk (*). You can include several FSAs in one FLD call, just as long as all of the fields that the FSAs reference are in the same segment. If you get an error status code for a FLD call, you should then check the status codes for each of the FSAs in the FLD call to find out where the error is.

Once you've verified the contents of a field in the data base, you can change the contents of that field in the same call. To do this, you supply an FSA that specifies a change operation for that field.

CHANGING A FIELD'S CONTENTS: FLD/CHANGE :To indicate to IMS/VS that you want to change the contents of a particular field, you use an FSA, just as you do in a FLD/VERIFY call. The difference is in the operators that you can specify, and the FSA status codes that IMS/VS can return to you after the call. Using Figure 65, FLD/CHANGE works like this:

1. You specify the name of the field that you want to change in the first field of the FSA—Field Name
2. You specify an operator in the third field of the FSA—Operator indicates to IMS/VS how you want to change that field.
3. You specify the value that IMS/VS is to use in changing the field in the last area of the FSA—the Field Value

By specifying different operators in a FLD/CHANGE call, you change the field in the data base in these ways:

- Add the value that I've supplied in the FSA to the value in the field.
- Subtract the value I've supplied in the FSA from the value in the field.
- Set the value in the data base field to the value I've supplied in the FSA.

You code these operators in the FSA with these symbols:

- To add: +
- To subtract: -
- To set the old value equal to the new one: =

You can add and subtract values only when the field in the data base contains arithmetic (packed decimal, binary fullword, or binary halfword) data.

Some of the status codes that IMS/VS can return to the FSA for a change operation are the same, but there are also some different ones. The status codes you can receive on a FLD/CHANGE FSA are:

A Invalid operation; for example, you specified the "" operator for a fieldthat contains character data.

B Invalid data length. The data you supplied in the FSA is not the length that is defined for that field in the DBD.

C You attempted to change the key field in the segment; changing the key field is not allowed.

E Invalid data in the FSA. The data that you supplied in the FSA is not the type of data that's defined for this field in the DBD.

F You tried to change an unowned segment. This status code applies only to related MSDBs.

G An arithmetic overflow occurred when you changed the data field.

AN EXAMPLE OF USING FLD/VERIFY AND FLD/CHANGE: Using FLD/VERIFY and FLD/CHANGE, you could perform the bank account processing that was described earlier in this chapter. This example uses the bank account segment shown in Figure 64. Assume that a customer wants to withdraw $100 from a checking account. The number of the checking account is 24056772. To find out whether or not the customer can withdraw this amount, you have to check the current balance. If the current balance is greater than $100, you want to subtract $100 from the balance, and add 1 to the transaction count in the segment.

You can do all of this processing using one FLD call and three FSAs. The FSAs and what each of them does are as follows:

1. Verify that the value in the BALANCE field is greater than or equal to $100. For this verification, you specify the BALANCE field, the H operator for greater than or equal to, and the amount. The amount is specifed without a decimal point. You also have to leave a blank between the field name and the operator for the FSA status code. This FSA looks like this:

   BALANCEbbH10000*

   The last character in the FSA is an asterisk because this FSA will be followed by other FSAs.

2. Subtract $100 from the value in the BALANCE field if the first FSA was successful. If the first FSA is unsuccessful, IMS/VS doesn't continue processing. To subtract the amount of the withdrawal from the amount of the balance, you use this FSA:

   BALANCEbb-10000*

   Again, the last character in the FSA is an asterisk because this FSA is followed by a third FSA.

3. Add 1 to the transaction count for the account. To do this, use this FSA:

   TRANCNTbb+001b

   In this FSA, the last character is a blank (b) because this is the last FSA for this call.

   When you issue the FLD call, you don't reference each FSA individually; you reference the I/O area that contains all of them.

Sync Point Processing in an MSDB

When you update a segment in an MSDB, IMS/VS doesn't apply your updates immediately. Updates don't go into effect until your program reaches a sync point. A sync point in a message-driven Fast Path program occurs each time the program issues a call to the message queue for a new message. A sync point in a nonmessage-driven Fast Path program occurs each time the program

issues a CHKP call or a synchronization, or SYNC, call. SYNC is a call that nonmessage-driven Fast Path programs can issue simply to invoke a sync point. A synchronization interval is the time between sync points in the application program's processing. When a program reaches a sync point, IMS/VS tries to apply the updates (and send output messages) that have been generated during that sync interval.

As a result of this, you can receive different results if you issue the same call sequence against a DL/I data base and an MSDB. For example, if you issue a GHU and a REPL for a segment in an MSDB, then you issue another get call for the same segment in the same sync interval, the segment that IMS/VS returns to you the "old" value, not the updated one. If, on the other hand, you issue the same call sequence for a segment in a DL/I data base, the second get call returns the updated segment.

When the program reaches a sync point, IMS/VS also reprocesses FLD/VERIFY calls to see if they are still accurate. If another program has updated the field that a FLD/VERIFY call was verifying, the program's processing could be inaccurate.

PROCESSING DEDBS

A DEDB contains a root segment and up to seven dependent segments. One of these can be a sequential dependent; the other six are direct dependents. Sequential dependent segments are stored in chronological order, regardless of the root that they're dependent on. Direct dependent segments are stored hierarchically. A DEDB is good for gathering detailed information and summary information.

Using DL/I Calls with DEDBs

The DL/I calls that you can issue against a root segment are the GU and GN (GNP has no meaning for a root segment), DLET, ISRT, and REPL. You can issue all of the DL/I calls against a direct dependent segment, and you can issue get calls and ISRT calls against sequential dependent segments.

There are three restrictions on using these calls with a DEDB:

- The call can contain only one SSA.
- The call cannot use command codes.
- The call cannot use Boolean qualification statements.

The POS Call

There is one call that you cannot use with IMS/VS data bases that you can use with DEDBs. This is the position call, or POS. This is a system call that retrieves the location of a specific sequential dependent segment or the location of the last inserted sequential dependent segment. You can also use the POS call to find out how much unused space there is within a DEDB area. "POS Call Format" explains how you code the POS call and what the I/O area for POS looks like. The POS call also tells you the amount of unused space within each DEDB area. For example, you can use the position information that IMS/VS returns for a POS call to scan or delete the sequential dependent segments for a particular time period.

LOCATING A SPECIFIC SEQUENTIAL DEPENDENT: When you have position on a particular root segment, you can retrieve the position information and the area name of a specific sequential dependent of that root. If you have a position established on a sequential dependent segment, the search starts from that position. IMS/VS returns the position information for the first sequential dependent segment that satisfies the call. To retrieve this information, you issue a POS call with a qualified or unqualified

SSA containing the segment name of the sequential dependent. Current position after this kind of POS call is in the same place that it is after a GNP call.

After a successful POS call, the contents for each position in the I/O area are as follows:

LL A 2-byte field giving the total length of the data in the I/O area, in binary

Area Name An 8-byte field giving the name of the area

Position An 8-byte field containing the position information for the requested segment

Unused CIs An 8-byte field containing the number of unused CIs in sequential dependent part

Unused CIs An 8-byte field containing the number of unused CIs in the independent overflow part

LOCATING THE LAST INSERTED SEQUENTIAL DEPENDENT SEGMENT: You can also retrieve the position information for the most recently inserted sequential dependent segment of a given root segment. To do this, you issue a POS call with an unqualified or qualified SSA containing the root segment as the segment name. Current position after this type of call follows the same rules as position after a GU.

After a successful call, the contents of the I/O area are as follows:

LL A 2-byte field containing the total length of the data, in binary

Area Name An 8-byte field giving the area name

Position An 8-byte field containing the position information for the most recently inserted sequential dependent segment. This field contains zeros if there is no sequential dependent for this root.

Unused CIs An 8-byte field containing the number of unused CIs in the sequential dependent part

Unused CIs An 1-byte field containing the number of unused CIs in the independent overflow part

IDENTIFYING FREE SPACE: To retrieve the area name and the next available position within the sequentia part from all online areas, you can issue an unqualified POS call. This type of call also retrieves the unused space in the independent overflow and sequential dependent parts.

After a successful unqualified POS call, the I/O area contains as many entries as there are areas within the data base. Each entry contains the following fields:

LL A 2-byte field containing the length of the data in the I/O area, in binary. The length includes 2 bytes for the LL field, plus the number of entries times 24.

Area Name An 8-byte field containing the area name

Position An 8-byte field giving the next available position within the sequential dependent part

Unused CIs An 8-byte field containing the number of unused CIs in the sequential dependent part

Unused CIs An 8-byte field containing the number of unused CIs in the independent overflow part.

Sync Point Processing in a DEDB

Sync point processing is performed after a message GU call, a SYNC call, or CHKP call. Fast Path keeps data base updates in main storage until the program reaches a sync point. IMS/VS saves updates to a DEDB in Fast Path buffers and are not applied to the data base. The data base updates are not applied to the DEDB until after the program has successfully completed sync point processing.

CHAPTER 7. CODING THE DL/I PORTION OF A PROGRAM

Before you start to code your program, make sure you've made all the design decisions explained in "Chapter 6. Structuring the DL/I Portion of a Program," such as when to use a qualified or unqualified call and what the most efficient call sequence is for your program. If everything is in order, coding the program is a matter of implementing the decisions that you have made. This chapter tells you how to code the part of the program that contains the program logic, and how to code the data area of the program. Before you start to code at all, however, you should understand what each of these parts of the program contains, and how the DL/I tools fit together. You should also make sure that you have all the necessary information to code the program. This chapter explains these preliminary considerations, and how you code your program, in this order:

- **Before You Code**

  This section shows you some sample skeleton programs and gives you some guidelines about the kind of information you need to code your program.

- **Coding the Program Logic**

  This section describes how you code an entry statement, DL/I calls, and system service calls; and how you check status codes.

- **Coding the Data Area**

  This section tells you how to code SSAs, DB PCB masks, I/O areas, function codes, and checkpoint IDs.

- **Coding Fast Path Data Base Calls**

  This section tells you what calls you can issue against each kind of Fast Path data base.

BEFORE YOU CODE

Before you start coding you need to understand two things: the order in which the parts of your program will appear, and the information that you must have to code the program.

PARTS OF A DL/I PROGRAM

The program logic contains the entry statement, the DL/I calls, system service calls, the processing logic, and the return statement. The data area contains the I/O area, the PCB mask, SSA definitions, and other data definitions.

Detailed program structure depends on the programming language you are using. The following are skeleton DL/I programs in COBOL, PL/I, and assembler language. Each of these programs retrieves data from the DETAIL data base and updates the MASTER data base. These programs don't show any of the processing logic; they show only the major parts of the program.

Note: These programs are provided as examples only. They will not run because they are incomplete; they are only skeletons that are intended to show you the overall structure of an IMS/VS batch application program.

COBOL DL/I Program Structure

The program in Figure 66 is a skeleton program in COBOL. It shows you how the parts of a DL/I program written in COBOL fit together. The numbers to the right of the program refer to the notes that follow the program. "Appendix A. Sample Batch Program" contains a sample COBOL batch program.

```
ENVIRONMENT DIVISION.
      .
      .
DATA DIVISION.
WORKING-STORAGE SECTION.
    77 FUNC-GU              PICTURE XXXX VALUE 'GU  '.
    77 FUNC-GHU             PICTURE XXXX VALUE 'GHU '.
    77 FUNC-GN              PICTURE XXXX VALUE 'GHN '.
    77 FUNC-GHN             PICTURE XXXX VALUE 'GHN '.
    77 FUNC-GNP             PICTURE XXXX VALUE 'GNP '.
    77 FUNC-GHNP            PICTURE XXXX VALUE 'GHNP'.
    77 FUNC-REPL            PICTURE XXXX VALUE 'REPL'.
    77 FUNC-ISRT            PICTURE XXXX VALUE 'ISRT'.
    77 FUNC-DLET            PICTURE XXXX VALUE 'DLET'.
    77 COUNT                PICTURE S9(5)VALUE +4 COMPUTATIONAL.

    01 UNQUAL-SSA.
       02 SEG-NAME          PICTURE X(08) VALUE  '        '.
       02 FILLER            PICTURE X     VALUE  ' '.

    01 QUAL-SSA-MAST.
       02 SEG-NAME-M        PICTURE X(08) VALUE 'ROOTMAST'.
       02 BEGIN-PAREN-M     PICTURE X     VALUE '('.
       02 KEY-NAME-M        PICTURE X(08) VALUE 'KEYMAST '.
       02 REL-OPER-M        PICTURE X(02) VALUE ' ='.
       02 KEY-VALUE-M       PICTURE X(06) VALUE 'vvvvvv'.
       02 END-PAREN-M       PICTURE X     VALUE ')'.
    01 QUAL-SSA-DET.
       02 SEG-NAME-D        PICTURE X(08) VALUE 'ROOTDET '.
       02 BEGIN-PAREN-D     PICTURE X     VALUE '('.
       02 KEY-NAME-D        PICTURE X(08) VALUE 'KEYDET  '.
       02 REL-OPER-D        PICTURE X(02) VALUE ' ='.
       02 KEY-VALUE-D       PICTURE X(06) VALUE 'vvvvvv'.
       02 END-PAREN-D       PICTURE X     VALUE ')'.
    01 DET-SEG-IN.
       02 --
       02 --
    01 MAST-SEG-IN.
       02 --
       02 --
LINKAGE SECTION.
    01 DB-PCB-MAST.
       02 MAST-DBD-NAME     PICTURE X(8).
       02 MAST-SEG-LEVEL    PICTURE XX.
       02 MAST-STAT-CODE    PICTURE XX.
       02 MAST-PROC-OPT     PICTURE XXXX.
       02 FILLER            PICTURE S9(5) COMPUTATIONAL.
       02 MAST-SEG-NAME     PICTURE X(8).
       02 MAST-LEN-KFB      PICTURE S9(5) COMPUTATIONAL.
       02 MAST-NU-SENSEG    PICTURE S9(5) COMPUTATIONAL.
       02 MAST-KEY-FB       PICTURE X---X.
    01 DB-PCB-DETAIL.
       02 DET-DBD-NAME      PICTURE X(8).
       02 DET-SEG-LEVEL     PICTURE XX.
       02 DET-STAT-CODE     PICTURE XX.
       02 DET-PROC-OPT      PICTURE XXXX.
       02 FILLER            PICTURE S9(5) COMPUTATIONAL.
       02 DET-SEG-NAME      PICTURE X(8).
       02 DET-LEN-KFB       PICTURE S9(5) COMPUTATIONAL.
       02 DET-NU-SENSEG     PICTURE S9(5) COMPUTATIONAL.
       02 DET-KEY-FB        PICTURE X---X.
```

1 2 3 4 5

Figure 66 (Part 1 of 2). COBOL DL/I Skeleton Program

```
PROCEDURE DIVISION.
      ENTRY 'DLITCBL' USING DB-PCB-MAST, DB-PCB-DETAIL.          6
       .
       .
       .
      CALL 'CBLTDLI' USING FUNC-GU, DB-PCB-DETAIL,               7
           DET-SEG-IN, QUAL-SSA-DET.
       .
       .
      CALL 'CBLTDLI' USING COUNT, FUNC-GHU, DB-PCB-MAST,         8
           MAST-SEG-IN, QUAL-SSA-MAST.
       .
       .
      CALL 'CBLTDLI' USING FUNC-GHN, DB-PCB-MAST,                9
           MAST-SEG-IN.
       .
       .
      CALL 'CBLTDLI' USING FUNC-REPL, DB-PCB-MAST,               10
           MAST-SEG-IN.
       .
       .
  GOBACK.                                                        11

  COBOL LANGUAGE INTERFACE                                       12
```

Figure 66 (Part 2 of 2). COBOL DL/I Skeleton Program

Notes:

1. You define each of the DL/I call functions the program uses with a 77 level or 01 level working storage entry. Each picture clause is defined as 4 alphameric characters and has a value assigned for each function. If you want to include the optional parmcount field, you can initialize count values for each type of call. You can also use COBOL COPY to include these standard descriptions in the program.

2. A 9-byte area is set up for an unqualified SSA. Before the program issues a call that requires an unqualified SSA, it moves the segment name to this area. If a call requires two or more SSAs, you may need to define additional areas.

3. A 01 level working storage entry defines each qualified SSA used by the application program. Qualified SSAs have to be defined separately because the values of the key fields are different.

4. A 01 level working storage entry defines I/O areas that will be used for passing segments to and from the data base. You can further define I/O areas with 02 level entries. You can use separate I/O areas for each segment type, or you can define one I/O area that you use for all segments.

5. A 01 level linkage section entry defines a mask for each of the DB PCBs that the program requires. The DB PCBs represent both input and output data bases. After issuing each DL/I call, the program checks the status code through this linkage. You define each field in the DB PCB so that you can reference them in the program.

6. This is the standard entry point in the procedure division of a batch program. After DL/I has loaded the PSB for the program in the DL/I region, DL/I passes control to the application program. The PSB contains all of the PCBs that the program

uses. The USING statement at the entry point to the batch program references each of the PCBs by the names that the program has used to define the PCB masks in the linkage section.

7. This call retrieves data from the data base using a qualified SSA. Before issuing the call, the program must initialize the key value of the SSA so that it specifies the particular segment to be retrieved. The program should test the status code in the DB PCB that was referenced in the call immediately after issuing the call.

8. This is another retrieval call that contains a qualified SSA.

9. This is an unqualified retrieval call.

10. The REPL call replaces the segment that was retrieved in the most recent get hold call with the data that the program has placed in the I/O area referenced in the call (MAST-SEG-IN).

11. The program issues the GOBACK statement when it has finished its processing.

12. IMS/VS supplies a language interface module (DFSLI000). This module must be link-edited to the batch program after the program has been compiled. It gives a common interface to IMS/VS and DL/I. If you use the IMS/VS-supplied procedures (IMSCOBOL or IMSCOBGO), IMS/VS link-edits the language interface with the application program. IMSCOBOL is a two-step procedure that compiles and link-edits your program. IMSCOBGO is a three-step procedure that compiles, link-edits, and executes your program in a DL/I batch region. For information on how to use these procedures, see "The IMS/VS Procedure Library" in the IMS/VS System Programming Reference Manual.

PL/I DL/I Program Structure

The program in Figure 67 is a skeleton program in PL/I. It shows you how the parts of a DL/I program written in PL/I fit together. The numbers to the right of the program refer to the notes that follow the program.

Note: IMS/VS application programs cannot use PL/I tasking. This is because all tasks operate as subtasks of a PL/I control task when you use multitasking.

```
/*                                                              */
/*                    ENTRY POINT                               */
/*                                                              */
DLITPLI: PROCEDURE (DB_PTR_MAST,DB_PTR_DETAIL)                          1
              OPTIONS (MAIN);

/*                                                              */
/*                DESCRIPTIVE STATEMENTS                        */
/*                                                              */
DCL   DB_PTR_MAST POINTER;
DCL   DB_PTR_DETAIL POINTER;
DCL   FUNC_GU      CHAR(4)     INIT('GU  ');
DCL   FUNC_GN      CHAR(4)     INIT('GN  ');
DCL   FUNC_GHU     CHAR(4)     INIT('GHU ');
DCL   FUNC_GHN     CHAR(4)     INIT('GHN ');                        2
DCL   FUNC_GNP     CHAR(4)     INIT('GNP ');
DCL   FUNC_GHNP    CHAR(4)     INIT('GHNP');
DCL   FUNC_ISRT    CHAR(4)     INIT('ISRT');
DCL   FUNC_REPL    CHAR(4)     INIT('REPL');
DCL   FUNC_DLET    CHAR(4)     INIT('DLET');

DCL 1     QUAL_SSA             STATIC UNALIGNED,

          2    SEG_NAME        CHAR(8) INIT('ROOT    '),
          2    SEG_QUAL        CHAR(1) INIT('('),
          2    SEG_KEY_NAME    CHAR(8) INIT('KEY     '),
          2    SEG_OPR         CHAR(2) INIT(' ='),                  3
          2    SEG_KEY_VALUE   CHAR(6) INIT('vvvvvv'),
          2    SEG_END_CHAR    CHAR(1) INIT(')');
DCL 1     UNQUAL_SSA           STATIC UNALIGNED,
          2    SEG_NAME_U      CHAR(8) INIT('NAME    '),
          2    BLANK           CHAR(1) INIT(' ');

DCL 1    MAST_SEG_IO_AREA,
         2 ---
         2 ---                                                      4
         2 ---
DCL 1    DET_SEG_IO_AREA,
         2 ---
         2 ---
         2 ---
DCL 1    DB_PCB_MAST           BASED (DB_PTR_MAST),
         2   MAST_DB_NAME      CHAR(8),
         2   MAST_SEG_LEVEL    CHAR(2),
         2   MAST_STAT_CODE    CHAR(2),
         2   MAST_PROC_OPT     CHAR(4),                             5
         2   FILLER           FIXED BINARY (31,0),
         2   MAST_SEG_NAME    CHAR(8),
         2   MAST_LEN_KFB     FIXED BINARY (31,0),
         2   MAST_NO_SENSEG   FIXED BINARY (31,0),
         2   MAST_KEY_FB      CHAR(*);
DCL 1    DB_PCB_DETAIL         BASE (DB_PTR_DETAIL),
         2   DET_DB_NAME      CHAR(8),
         2   DET_SEG_LEVEL    CHAR(2),
         2   DET_STAT_CODE    CHAR(2),
         2   DET_PROC_OPT     CHAR(4),
         2   FILLER           FIXED BINARY (31,0),
         2   DET_SEG_NAME     CHAR(8),
         2   DET_LEN_KFB      FIXED BINARY (31,0),
         2   DET_NO_SENSEG    FIXED BINARY (31,0),
         2   DET_KEY_FB       CHAR(*);
```

Figure 67 (Part 1 of 2). PL/I DL/I Skeleton Program

```
DCL     THREE    FIXED BINARY     (31,0)   INITIAL(3);
DCL     FOUR     FIXED BINARY     (31,0)   INITIAL(4);
DCL     FIVE     FIXED BINARY     (31,0)   INITIAL(5);
DCL     SIX      FIXED BINARY     (31,0)   INITIAL(6);
/*                                                    */
/*            MAIN PART OF PL/I BATCH PROGRAM         */
/*                                                    */
     CALL PLITDLI(FOUR,FUNC_GU,DB_PCB_DETAIL,
     DET_SEG_IO_AREA,QUAL_SSA);

     CALL PLITDLI(FOUR,FUNC_GHU,DB_PCB_MAST,
     MAST_SEG_IO_AREA,QUAL_SSA);

     CALL PLITDLI(THREE,FUNC_GHN,DB_PCB_MAST,
     MAST_SEG_IO_AREA;

     CALL PLITDLI(THREE,FUNC_REPL,DB_PCB_MAST,
     MAST_SEG_IO_AREA);

     RETURN;
     END DLITPLI;
```

6 7 8 9 10 11

PL/I LANGUAGE INTERFACE — 12

Figure 67 (Part 2 of 2). PL/I DL/I Skeleton Program

Notes:

1. After DL/I has loaded the application program's PSB, DL/I gives control to the application program through this entry point. PL/I programs must past the pointers to the PCBs, not the names, in the entry statement. The entry statement lists the PCBs that the program uses by the names that it has assigned to the definitions for the PCB masks. The order in which you refer to the PCBs in the entry statement must be the same order in which they've been defined in the PSB.

2. Each of these areas defines one of the call functions used by the batch program. Each character string is defined as 4 alphameric characters, with a value assigned for each function. You can define other constants in the same way. Also, you can store standard definitions in a source library and include them by using a %INCLUDE statement.

3. A structure definition defines each SSA the program uses. The unaligned attribute is required for SSAs. The SSA character string must reside contiguously in storage. You should define a separate structure for each qualified SSA, because the value of each one's key field is different.

4. The I/O areas that will be used to pass segments to and from the data base are defined as structures.

5. Level 1 declaratives define masks for the DB PCBs that the program uses as structures. These definitions make it possible for the program to check fields in the DB PCBs.

6. This statement defines the parmcount that is required in DL/I calls issued from PL/I programs. The parmcount gives the number of parameters in the call that follow parmcount itself.

7. This call retrieves data from the data base. It contains a qualified SSA. Before you can issue a call that uses a qualified SSA, you must initialize the key field of the SSA. Before you can issue a call that uses an unqualified SSA, you must initialize the segment name field. You should check the status code after each DL/I call that you issue.

   Although you must pass the PCB pointer in the entry statement to a PL/I program, you can pass either the PCB name or the PCB pointer in DL/I calls in a PL/I program.

8. This is another call that is qualified with a qualified SSA.

9. This call is an unqualified call that retrieves data from the data base. Because it is a get hold call, it can be followed by a REPL call or a DLET call.

10. The REPL call replaces the data in the segment that was retrieved by the most recent get hold call with the data in the I/O area referenced in the segment.

11. The RETURN statement returns control to DL/I.

12. IMS/VS provides a language interface module (DFSLI000) that gives a common interface to IMS/VS and DL/I. This module must be link-edited to the program. If you use the IMS/VS-supplied procedures (IMSPLI or IMSPLIGO), IMS/VS link-edits the language interface module to the application program. IMSPLI is a two-step procedure that compiles and link-edits your program. IMSPLIGO is a three-step procedure that compiles, link-edits, and executes your program in a DL/I batch region. For information on how to use these procedures, see "The IMS/VS Procedure Library" in the IMS/VS System Programming Reference Manual.

Assembler Language DL/I Program Structure

The program in Figure 68 is a skeleton program in assembler language. It shows how the parts of a DL/I program written in assembler language fit together. The numbers to the right of the program refer to the notes that follow the program. Throughout this program, you could replace ASMTDLI with CBLTDLI.

```
PGMSTART CSECT                                                    1
*                EQUATE REGISTERS
*   USEAGE OF REGISTERS
R1        EQU   1               ORIGINAL PCBLIST ADDRESS
R2        EQU   2               PCBLIST ADDRESS1
R5        EQU   5               PCB ADDRESSS
R12       EQU   12              BASE ADDRESS
R13       EQU   13              SAVE AREA ADDRESS
R14       EQU   14
R15       EQU   15
*
          USING PGMSTART,R12    BASE REGISTER ESTABLISHED
          SAVE  (14,12)         SAVE REGISTERS
          ST    R13,SAVEAREA+4  SAVE AREA CHAINING
          LA    R13,SAVEAREA    NEW SAVE AREA                       2
          USING PCBLIST,R2      MAP INPUT PARAMETER LIST
          USING PCBNAME,R5      MAP DB PCB
          LR    R2,R1           SAVE INPUT PCB LIST IN REG 2
          L     R5,PCBDETA      LOAD DETAIL PCB ADDRESS             3
          LA    R5,0(R5)        REMOVE HIGH ORDER END OF LIST FLAG
```

Figure 68 (Part 1 of 2). Assembler Language Skeleton Program

```
         CALL  ASMTDLI,(GU,(R5),DETSEGIO,SSANAME),VL                    4
*
*
         L     R5,PCBMSTA      LOAD MASTER PCB ADDRESS
         CALL  ASMTDLI,(GHU,(R5),MSTSEGIO,SSAU),VL                      5
*
*
         CALL  ASMTDLI,(GHN,(R5),MSTSEGIO),VL
*                                                                       6
*
         CALL  ASMTDLI,(REPL,(R5),MSTSEGIO),VL
*
*
         L     R13,4(R13)      RESTORE SAVE AREA                        7
         RETURN (14,12)      RETURN BACK
*
*      FUNCTION CODES USED
*
GU       DC    CL4'GU'                                                  8
GHU      DC    CL4'GHU'
GHN      DC    CL4'GHN'
REPL     DC    CL4'REPL'
*
*       SSAS
*
SSANAME  DS    0C
         DC    CL8'ROOTDET'
         DC    CL1'('                                                   9
         DC    CL8'KEYDET'
         DC    CL2' ='
NAME     DC    CL6'  '
         DC    C')'
*
SSAU     DC    CL9'ROOTMST'
*
MSTSEGIO DC    CL100' '                                                 10
DETSEGIO DC    CL100' '
SAVEAREA DC    18F'0'
*
PCBLIST  DSECT
PCBMSTA  DS    A               ADDRESS OF MASTER PCB
PCBDETA  DS    A               ADDRESS OF DETAIL PCB
*                                                                       11
PCBNAME  DSECT
DBPCBDBD DS    CL8             DBD NAME
DBPCBLEV DS    CL2             LEVEL FEEDBACK
DBPCBSTC DS    CL2             STATUS CODES
DBPCBPRO DS    CL4             PROC OPTIONS
DBPCBRSV DS    F               RESERVED
DBPCBSFD DS    CL8             SEGMENT NAME FEEDBACK
DBPCBMKL DS    F               LENGTH OF KEY FEEDBACK
DBPCBNSS DS    F               NUMBER OF SENSITIVE SEGMENTS IN PCB
DBPCBKFD DS    C               KEY FEEDBACK AREA
         END   PGMSTART
```

ASSEMBLER LANGUAGE INTERFACE — 12

Figure 68 (Part 2 of 2). Assembler Language Skeleton Program

Notes:

1. The entry point to an assembler language program can have any name.
2. When DL/I passes control to the application program, register 1 contains the address of a variable-length fullword

parameter list. Each word in this list contains the address of a PCB that the application program must save. The high-order byte of the last word in the parameter list has the 0 bit set to a value of 1 to indicate the end of the list. The application program uses these addresses later on in executing DL/I calls.

3. THe program loads the address of the DETAIL DB PCB.
4. The program issues a GU call to the DETAIL data base using a qualified SSA (SSANAME).
5. The program then loads the address of the MASTER DB PCB.
6. The next three calls that the program issues are to the MASTER data base. The first call is a GHU call that uses an unqualified SSA. The second call is an unqualified GHN. The REPL call replaces the segment retrieved with the GHN with the segment in the MSTSEGIO area.
7. The RETURN statement loads DL/I registers and returns control to DL/I.
8. The call functions are defined as 4-character constants.
9. The program defines each part of the SSA separately so that it can modify the SSAs fields.
10. The program must define an I/O area large enough to contain the largest segment it will retrieve or insert (or the largest path of segments if the program uses the D command code). This program's I/O areas are 100 bytes each.
11. A fullword must be defined for each DB PCB. The assembler language program can access status codes after a DL/I call by using the DB PCB base addresses.
12. The IMS/VS-supplied language interface module (DFSLI000) must be link-edited with the compiled assembler language program.

YOUR INPUT

In addition to knowing the design and processing logic for your program, you need to know about the data that your program will be processing, the PCBs it references, and the segment formats in the hierarchies your program processes. You can use the list below as a checklist to make sure that you're not missing any information. If you're missing information about your data, the DL/I options being used in your application, or segment layouts and your application's data structures, you should get this information from the DBA, or the equivalent specialist at your installation. You should also be aware of the programming standards and conventions that have been established at your installation.

Information You Need about the Program's Design

- The call sequence for your program
- For each DL/I call, the format of the call:
  - Does the call include any SSAs?
  - If so, are they qualified or unqualified?
  - Does the call include any command codes?
- The processing logic for the program
- The routine the program will use to check the DL/I status code after each call

- The error routine(s) the program will use

Information You Need about Checkpoints

- The type of checkpoint call to use (basic or symbolic)
- The identification to assign to each checkpoint call, regardless of whether the checkpoint call is basic or symbolic
- If you are going to use the symbolic checkpoint call, which areas, if any, of your program you will checkpoint
- If you are going to use the basic call, whether or not you are going to use the OS/VS option on that call

Information You Need about Each Segment

- Whether the segment is fixed or variable length
- The length of the segment (the maximum length, if the segment is variable length)
- The names of the fields that each segment contains
- Whether or not the segment has a key field. If it does, is the key field unique or nonunique? If it doesn't, what sequencing rule has been defined for it? (A segment's key field is defined in the SEQ keyword of the FIELD statement in the DBD. The sequencing rule is defined in the RULES keyword of the SEGM statement in the DBD.)
- The segment's field layouts: the byte location of each field, the length of each field, and the format of each field

Information You Need about the Program's Hierarchies

- Each data structure, your program processes has been defined in a DB PCB. All of the PCBs your program references are part of a PSB for your application program. You need to know the order in which the PCBs that your application program references are defined in the PSB. You will use this information in the entry statement for your program.
- The layout of each of the data structures your program processes.
- For each data structure, whether multiple or single positioning has been specified for it. This is specified in the POS keyword of the PCB statement during PSB generation.
- Whether any data structures use multiple DB PCBs.

CODING THE PROGRAM LOGIC

This section tells you how to code the main part of your program: the entry statement, DL/I calls, system service calls, error routines, and program termination. If you are writing your program in COBOL, this is the procedure division. If you are writing your program in PL/I, this is the code that comes after the data declarations. If you are writing your program in assembler language, this is the section before your data definitions.

Notes:

1. The parameter "parmcount" is required in all PL/I calls except the call that calls the status code error routine. It is optional for all COBOL calls (except the call to the status

code error routine, where it is not allowed), although it is not shown in any of the COBOL calls because it is never required. Parmcount can also be used instead of VL in any of the assembler language calls. These options are not shown on each COBOL and assembler language call in the interest of keeping the explanations of the call formats as simple as possible.

2. Also, for assembler calls, you can substitute CBLTDLI for ASMTDLI in any of the calls.

CODING AN ENTRY STATEMENT

The entry statement in a DL/I application program lists the PCBs that the application program uses. Your entry statement must refer to the DB PCBs in the same order in which they are defined in your program's PSB. If your program uses an I/O PCB, it must refer to the I/O PCB before referencing any of the DB PCBs.

PL/I programs must pass the pointers to the PCBs, not the PCB names, in the entry statement.

CODING DL/I CALLS

At a minimum, the parameters in a DL/I call indicate three things to DL/I: the type of call you're issuing, for example, a GU; the DB PCB that you want DL/I to use for this call; and the segment I/O area. The parameters giving this information are required in all of the DL/I calls in each language. You can also use SSAs in all of the DL/I calls. There are two additional parameters that are used: "parmcount" and "VL". Whether or not you use these parameters depends on the language you're using.

PL/I programs can reference either the PCB name or the PCB pointer in DL/I calls.

CODING SYSTEM SERVICE CALLS FOR RECOVERY

Three of the system service calls that a batch program can use are recovery tools: symbolic CHKP, XRST, and basic CHKP. Symbolic CHKP and XRST must be used together; the formats for these calls are identical. Basic CHKP has a different format and different parameters.

Checkpointing an application program is an important part of an installation's recovery procedures. Because of this, installations usually establish specific standards regarding checkpoint frequency, checkpoint IDs, and (if you use the symbolic call) which areas of your program to checkpoint. Be sure that you are following these standards before you code your CHKP calls and checkpoint IDs.

Each of these calls must reference the I/O PCB, rather than the DB PCB, as one of the call parameters. IMS/VS automatically generates a dummy PCB that can be used in a batch program as an I/O PCB if you specify the compatibility option (CMPAT=YES) on the PSB for your application program.

CODING SYSTEM SERVICE CALLS FOR MONITORING

Three of the system service calls that you can use in a batch program are monitoring tools: GSCD, LOG, and STAT. Like the CHKP and XRST calls, the LOG call must be issued against the I/O PCB, not the DB PCB. IMS/VS will automatically generate a dummy PCB that a batch program can use as an I/O PCB if you specify the compatibility option (CMPAT=YES) in the PSB for your batch program.

CHECKING STATUS CODES

Your program should include code to check the status code that DL/I returns in the PCB mask each time your program issues a DL/I call or a system service call. If DL/I returns an error status code, your program should branch to an error routine to terminate. Most installations have a standard error routine that is available to the application programs at the installation.

Not all status codes indicate errors. Because of this, your program should check for exceptional or warning status codes before it branches to an error routine. For example, "GE" means that DL/I has reached the end of the data base while searching for the segment you requested. You would not necessarily want to branch to an error routine in this case; the fact that DL/I reached the end of the data base could mean that it was time for the program to exit from a loop. Below is a COBOL example of how you might handle warning or exceptional status codes:

```
CALL 'CBLTDLI' USING GN-FUNC, DB-PCB, I/O-AREA.
IF PCB-STATUS EQ 'GE' PERFORM PRINT-NOT-FOUND.
IF PCB-STATUS NE 'bb' PERFORM STATUS-ERROR.
```

If the status code indicates an error—in other words, it is not just a warning or an indication of an exceptional condition—your program should be able to print out as much information as possible about the error before terminating. For example, if the error was caused by an invalid SSA, you would want to be able to look at the SSA that was being used when the program terminated. If your installation does not have a standard status code error routine, you might want to use the sample routine provided in "Appendix E. Sample Status Code Error Routine (DFSOAER)." Using this routine or a similar one can make debugging your program an easier job.

CODING THE DATA AREA

Each of the parameters of a DL/I call is the address of an area in your program in which you've defined the information that DL/I needs for the call. For example, for the SSA parameter, you give the name of the area in your program where you have defined the SSA. This section tells how to code these areas in the order in which they appear in a DL/I call: the parmcount, if you are using PL/I; the call function code; the DB PCB mask; the I/O area; and the SSAs, if you are using any.

CODING THE PARMCOUNT

Parmcount is the address of a 4-byte field that contains the number of parameters that follow in the call. Parmcount is required only in PL/I programs. It is optional in COBOL and assembler language programs. The value in parmcount is binary. The example below shows how you could code a the parmcount parameter when three parameters follow in the call:

```
DCL   THREE   FIXED BINARY    (31,0)   INITIAL(3);
```

CODING DL/I FUNCTION CODES

For each DL/I call that your program issues, you define the function code in the data area in your program. The label that you give to the definition of the function code can be anything you want it to be, including the same value as the function code itself, within the conventions of the programming language you are using. The order in which you define the function codes is not important. The following examples are from the skeleton programs shown in Figure 66, Figure 67, and Figure 68. Refer to these figures to see how the parameters appear in the DL/I calls.

GU Function Code for COBOL

```
WORKING-STORAGE SECTION.
     77 FUNC-GU          PICTURE XXXX VALUE 'GU  '.
```

Each call function used by your program should be defined as a 77 level or 01 level working storage entry. Each picture clause is defined as 4 alphameric characters and has a value assigned for each function. Notice that the value "GU" is padded with 2 blanks on the right to make it 4 bytes long.

ISRT Function Code for PL/I

```
/*      DESCRIPTIVE STATEMENTS      */
.
.
.
DCL  FUNC_ISRT      CHAR(4)      INIT('ISRT');
```

The function code "ISRT" is 4 bytes long, so you don't have to pad it.

REPL Function Code for Assembler Language

```
*      FUNCTION CODES USED
.
.
.
REPL         DC      CL4'REPL'
```

If you are using assembler language, define the call functions as 4-character constants. Notice that in this example the label is the same as the function code.

CODING DB PCB MASKS

To use the DB PCBs that define your program's data structures, your program defines a mask of the DB PCB; it doesn't use the actual DB PCB. Since PCBs don't reside in your program, the PCB mask should be defined in the linkage section if you're using COBOL, as a based variable if you're using PL/I, or as a dsect if you're using assembler language. You must have a PCB mask for each PCB that your program references. Although the entry statement must list the PCBs in the same order in which they are defined in the PSB, the PCB masks may appear in any order when you define them in your program.

CODING THE I/O AREA

The I/O area parameter on a DL/I call gives the address of the input/output area in your program. The length of this area is very important. If you don't use any path calls, the area must be as long as any segments that your program will retrieve or insert. If your program does issue path calls, this area must be long enough to hold the longest concatenated segment that might be retrieved using a path call, or the concatenated segment that you place in the I/O area before issuing an ISRT call.

When DL/I returns a segment to this area, the area is not cleared between calls; your program has to use the length field in the PCB to determine the length of the current segment in the I/O area. If this area is too small, the data from each call can overlay the data from the previous call. You can use separate I/O areas for each segment type you use, or you can use a single area.

CODING SSAS

There can be three parts to an SSA: a segment name, a command code, and a qualification statement. At the minimum, an SSA contains the name of the segment type that you want. Command codes and qualification statements are optional.

CODING CHECKPOINT IDS

This area is always 8 bytes long. The format of the area depends on the type of ID your installation uses. Checkpoint IDs must be unique both within your program and across application programs. Because of this and because checkpoint IDs are used in communicating between the MTO and programmers, most installations establish specific standards for checkpoint IDs. If you are unsure about the conventions at your installation, check with the person who handles recovery.

GSAM CODING CONSIDERATIONS

The calls your program uses to access GSAM data bases are not the same as the DL/I calls. This section tells how to code GSAM calls and GSAM data areas, and it describes the JCL restrictions that apply to GSAM. The system service calls that you can use with GSAM are symbolic CHKP and XRST.

Figure 69 summarizes the GSAM data base calls. There are five calls you can use to process GSAM data bases: OPEN, CLSE, GN, ISRT, and GU. The COBOL, PL/I, and assembler call formats and parameters for these calls are described below. GSAM calls don't look too different from DL/I calls. The main differences are that GSAM calls must reference the GSAM PCB, and that they do not use SSAs.

Function Code	Meaning	Use	Options	Parameters
OPEN	Open	Explicitly opens GSAM data base	Can specify printer or punch control characters	function, gsam pcb [,open option]
CLSE	Close	Explicitly closes GSAM data base	None	function, gsam pcb
GNbb	Get next	Retrieves next sequential record	Can supply address for RSA	function, gsam pcb, i/o area [,rsa name]
ISRT	Insert	Adds new record at end of data base	Can supply address for RSA	function, gsam pcb, i/o area [,rsa name]
GUbb	Get unique	Establishes position in data base or retrieves a unique record	none	function, gsam pcb, i/o area, rsa name

Figure 69. Summmary of GSAM Calls

CODING FAST PATH DATA BASE CALLS

You can use all of the DL/I calls except GNP and GHNP to access Fast Path data bases. You can also use two additional calls: FLD and POS. When you can use each of these calls depends on the type of Fast Path data base that you are processing. The following restrictions apply to DL/I calls that you issue to Fast Path data bases.

- The call can contain only one SSA.
- The SSA cannot have Boolean qualification statements.
- The SSA cannot use any command codes.

The calls you use to process MSDBs are:

- For nonterminal-related MSDBs:
  - FLD
  - GU and GHU
  - GN and GHN
  - REPL
- For terminal-related, fixed MSDBs:
  - FLD
  - GU and GHU
  - GN and GHN
  - REPL
- For terminal-related, dynamic MSDBs:
  - FLD
  - GU and GHU
  - GN and GHN
  - DLET
  - REPL
  - ISRT

You can use these calls to process a DEDB:

- GU and GHU
- GN and GHN
- REPL
- ISRT
- DLET
- POS

CHAPTER 8. STRUCTURING A MESSAGE PROCESSING PROGRAM

This chapter tells you how to use the DC calls to communicate with terminals and other programs, and it explains some additional tools that an MPP can use to process online data bases. To read and update a data base, a message processing program, or MPP, uses the calls and tools described in "Chapter 6. Structuring the DL/I Portion of a Program."

The decisions involved in structuring an MPP are decisions involving the messages that the program receives and sends. This chapter explains what an MPP can do, and the decisions that you have to make about the program's structure, in this order:

- **How You Send and Receive Messages: An Overview**

  This section explains what kinds of messages you can send and receive, and gives a general description of how you do it.

- **What Input Messages Look Like**

  This section explains the format of input messages.

- **What Output Messages Look Like**

  This section describes the format of output messages.

- **How You Edit Your Messages**

  This section explains how you edit input and output messages using MFS, and what basic edit does for you.

- **Retrieving Messages**

  This section explains the calls the program uses to receive messages, and what the message looks like when the program receives it.

- **Sending Messages: ISRT, CHNG, and PURG**

  This section explains the calls that an MPP uses to send messages once the MPP has done the required processing. The MPP doesn't necessarily reply to the terminal or program that sent the message; this section explains how an MPP can respond to an input message.

- **Communicating with Other IMS/VS Systems**

  If your installation has two or more IMS/VS DB/DC systems, it can use Multiple Systems Coupling to make it possible for a terminal in one system to communicate with terminals and programs in other systems. This section explains how this affects your programming.

- **Conversations**

  A conversation allows a terminal to have several interactions with one or more MPPs without having to start the transaction from the beginning with each interaction. This section tells you how to structure a conversational program.

- **Issuing Commands**

  This section explains how an MPP can issue commands. The most frequent use of these commands is in conjunction with the Automated Operator Interface.

- **Reserving and Releasing Segments**

  When an MPP accesses the data base, there may be several other application programs trying to access the same segments. This section explains how IMS/VS automatically keeps several programs from accessing one data base segment at a time, and how you can explicitly reserve segments by using the Q command code.

- **Backing Out Data Base Updates: ROLB and ROLL**

  Unlike a batch program, an MPP can back out the data base updates it has made since its last sync point while it is still executing. This section explains the calls that you use to do this, and the situations in which you would use these calls.

- **Considerations for Message-Driven Fast Path Programs**

  This section explains the differences between message-driven Fast Path programs and MPPs that affect your programming.

HOW YOU SEND AND RECEIVE MESSAGES: AN OVERVIEW

When a program accesses data in a DL/I data base, the program uses retrieval calls to read the data, and the update calls to add, replace, or delete data. In the same way, an MPP has two calls that it uses to retrieve messages, and three calls that it can use to send messages.

Just as you receive results from a data base call in your program's I/O area and the DB PCB mask, you receive results from a message call in your program's I/O area and an I/O PCB mask. IMS/VS places the input messages for your program in the I/O area, and, when you want to send a message, you build it in the I/O area before sending it.

When your program receives a message, IMS/VS returns the following information about the message to the I/O PCB:

- The name of the terminal that sent the message
- A 2-character status code describing the results of the call. For example, if the program receives a status code of QC after issuing a call to retrieve a message, that means that there are no more messages for the program.
- The current date, time, and sequence number for the message
- The userid of the person at the terminal, the logical terminal name, or the transaction code for the program that sent the message.

Figure 70 shows the parts of an MPP. The numbers to the right of the picture refer to the notes that follow it.

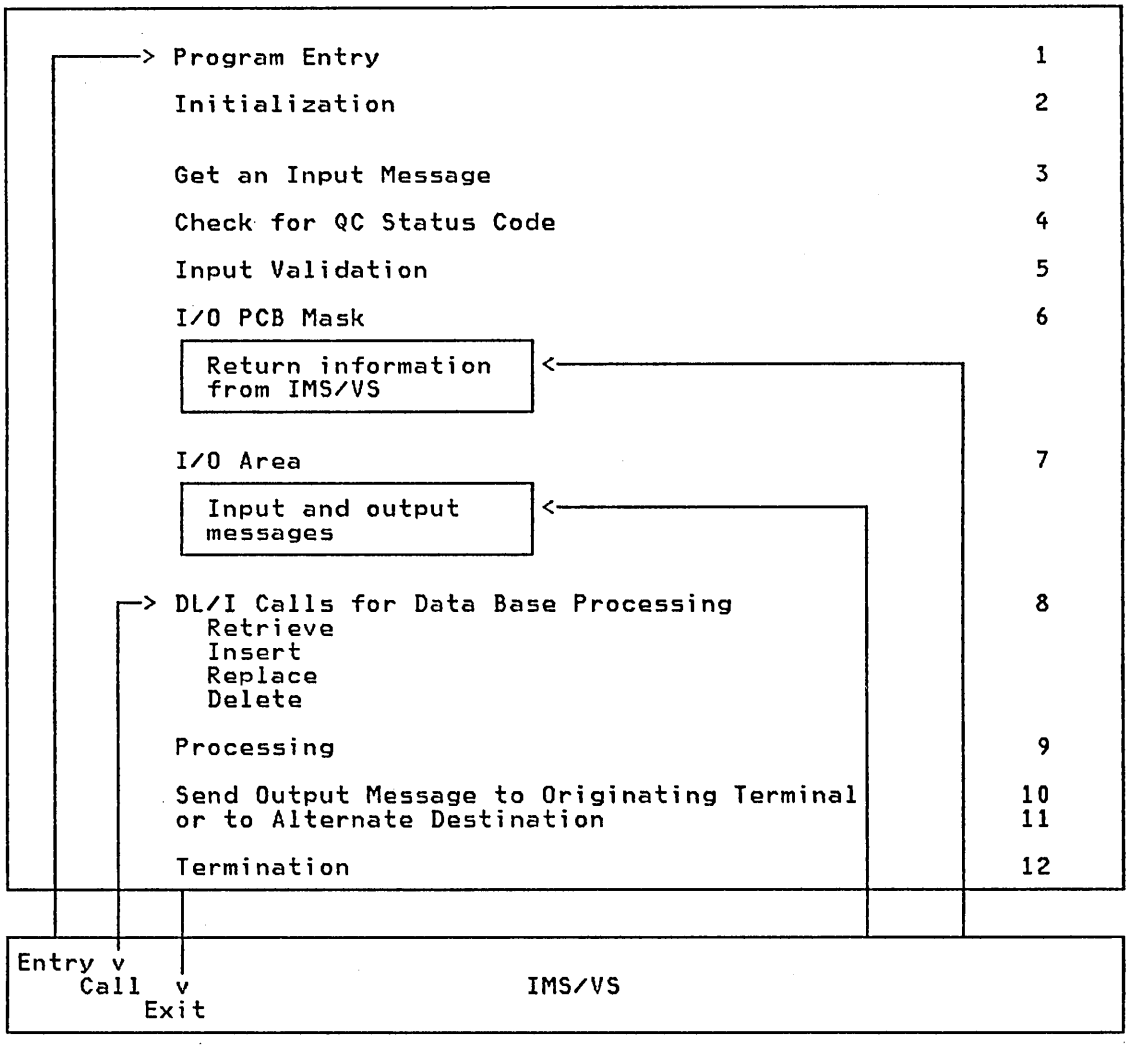

Figure 70. Basic MPP Structure

Notes:

1. **Program Entry.** IMS/VS passes control to the MPP when there are messages in its message queue.

2. **Initialization.** You can design your program to be reusable by initializing constants and switches that the program modifies at program entry. A reusable program is one that IMS/VS does not have to load each time it schedules the program. Instead, IMS/VS loads the program once, for the first message, and passes control to the program each time there is a new message for the program to process.

   To make your program reusable, try to avoid modifying constants (such as SSAs) and using switches. If, however, you want to use either of these techniques, make sure that you initialize the constants and switches you've modified at the beginning of the program, not at the end of the program.

3. **Get an Input Message.** The MPP issues a GU to retrieve the first segment in a message, and GN calls to retrieve the remaining segments in the message.

4. **Check for QC Status Code.** When a program issues a call for a new message, but there are no more messages for it to process, IMS/VS returns a QC status code to the I/O PCB.

5. **Input Validation.** To make sure that its input messages are valid, the MPP checks the format, value, and consistency of the input data fields. The program should do this as completely as possible, and before it issues any DL/I calls to the data base.

6. **I/O PCB Mask.** IMS/VS describes the results of each message call in the I/O PCB mask. Your program reads the information in this area after each DC call to find out whether or not the call was successful, and if there are any more messages to process.

7. **I/O Area.** IMS/VS returns input messages to the I/O area, and your program builds output messages in the I/O area.

8. **DL/I Calls for Data Base Processing.** If necessary to process the message, the program issues DL/I calls to read and update information in the data base.

9. **Processing.** The program does the required processing.

10. **Send Output Message to the Originating Terminal.** The MPP sends the response output message to the originating terminal by using the I/O PCB.

11. **Send Output Message to an Alternate Terminal.** The program can send a message to a terminal or program other than the terminal or program that sent the message.

12. **Terminate.** The program must terminate when there are no more input messages for it to process.

DC CALLS

An MPP retrieves and sends messages by issuing calls similar to DL/I calls. The calls used by an MPP are called data communication calls, or DC calls. IMS/VS messages are made up of one or more segments. The calls you use when you retrieve and send messages are:

GU A get unique call retrieves the first segment of a new message.

GN A get next call retrieves the remaining segments of that message.

ISRT An insert call sends one message segment to the destination represented by the PCB that you reference in the call.

CHNG A change call sets the destination of a modifiable alternate PCB to the logical terminal or transaction code that you specify.

PURG When you send messages using the ISRT call and you don't use PURG, IMS/VS groups the message segments into a message and sends the message when you retrieve the first segment of the next message (or when you reach a sync point). But if you issue a PURG call after issuing ISRT calls for one or more message segments, you can send that output message to its destination before you retrieve the next input message.

CMD A command call makes it possible for an application program to issue IMS/VS commands.

GCMD The get command call retrieves responses from IMS/VS to the command issued by the application program.

To retrieve messages, an MPP uses GU and GN.

A message contains one or more segments. If an input message contains only one segment, a GU call retrieves the entire message. If an input message contains more than one segment, a GU retrieves the first segment, and GN calls retrieve the remaining segments of the message.

To send a message, you use the ISRT call. Before issuing the call, you build the output message in an I/O area—just as you build a segment that you're adding to the data base in an I/O area. Then, when you issue the ISRT call, you reference that I/O area as one of the call parameters. To send the message to the terminal that sent the most recent input message, you reference the I/O PCB in the call.

To send a message to a different terminal or to an application program, you issue the ISRT call, but you reference an alternate PCB instead of the I/O PCB. Alternate PCBs can be defined for a particular terminal or program, or they can be defined as modifiable. If the alternate PCB is not modifiable, you just issue an ISRT call referencing the alternate PCB to send a message to the terminal or program that it represents. If the alternate PCB is modifiable, you set the destination for the alternate PCB before you issue the ISRT call. To do this, you use a CHNG call.

When you issue a PURG call after issuing one or more ISRT calls, the PURG call makes it possible to send several output messages to a destination while you're processing the same the same input message. If you include the address of an I/O area on a PURG call, PURG inserts the data in the I/O area as the first segment of a new message—in addition to sending the complete output message you've already inserted.

An application program can also issue IMS/VS commands, and retrieve responses to the commands it issues. There are two calls you use to do this. A CMD call sends the command to IMS/VS, and returns the first segment of the response from IMS/VS, if there is one. A GCMD retrieves remaining responses to the command.

Note: In general, IMS/VS application programs should avoid issuing OS/VS STIMER macros. The reason for this is that IMS/VS uses STIMER for control purposes. If an application program issues an STIMER before the time interval set by the IMS/VS STIMER has expired, the application program's STIMER cancels the STIMER issued by IMS/VS. MVS then resets the time interval for IMS/VS, and the results are unpredictable. If, for some reason, your program must use STIMER, you need to set the STIMER parameter to 0. This means that you don't want IMS/VS to issue STIMERs. For an MPP, you set the STIMER parameter in the DFSMPR procedure; for a BMP, you set the STIMER parameter in the IMSBATCH procedure; and for a Fast Path program, you set the STIMER parameter in the IMSFP procedure. OS/VS2 MVS Supervisor Services and Macro Instructions describes the OS/VS macros.

I/O PCB Masks

An I/O PCB represents the logical terminal that sent the input message to the application program. An I/O PCB, like a DB PCB, is outside of the application program, so the program must define a mask of the I/O PCB in order to check the fields of the I/O PCB. IMS/VS describes the results of retrieval calls and ISRT calls that reference the I/O PCB in the I/O PCB. To find out about your DC calls, your program checks the I/O PCB by referencing the I/O PCB mask. An MPP checks the status code after a DC call in the same way that a program checks the status code after a DL/I call.

An I/O PCB contains 8 fields. Figure 71 shows these fields and gives the length of each.

1. Logical Terminal Name 8 bytes
2. Reserved for IMS/VS 2 bytes
3. Status Code 2 bytes
4. Current Date 4 bytes
5. Current Time 4 bytes
6. Input Message Sequence Number 4 bytes
7. Message Output Descriptor Name 8 bytes
8. User Identification 8 bytes

Figure 71. I/O PCB Mask

Notes:

1. **Logical Terminal Name:** When you receive an input message, IMS/VS places the name of the logical terminal that sent the message in this area. When you want to send a message back to this terminal, you refer to the I/O PCB when you issue the ISRT call, and IMS/VS takes the name of the logical terminal from the I/O PCB mask as the destination.

2. **Reserved for IMS/VS**

   This area is reserved for IMS/VS.

3. **Status Code**

   IMS/VS places the status code describing the result of the DC call in this field. IMS/VS updates the status code for each DC call. A blank status code means that your call was successful.

   Like the status codes that DL/I returns after a data base call, some of the status codes that you can receive after a DC call indicate exceptional conditions rather than errors. Your program should always test the status code after a DC call, and it should be prepared for the status codes that indicate such conditions. For example, a QC status code means that there are no more messages in the message queue for the MPP; when your program receives this status code, it should terminate immediately.

   For status codes that indicate errors, your program should have an error routine available to it that prints information about the most recent call before terminating the program. Most installations have a standard error routine that all application programs at the installation use. MPPs can use the sample status code error routine provided in "Appendix E. Sample Status Code Error Routine (DFSOAER)." "Checking Status Codes" describes this routine.

4. **Current Date**

   The current date, current time, and the input message sequence number are in the prefix of all input messages. Only

I/O PCBs have the fields that contain the current date, current time, and input message sequence number. Alternate PCBs don't have these fields because you never use an alternate PCB to receive an input message; you only use it to send output messages. The current date is the Julian date given in packed decimal, right-aligned, in the format, "YYDDD". This gives the date that IMS/VS received the entire message and enqueued it as input for the MPP. If your system uses MSC, this field contains the data that your IMS/VS system received the message from the originating terminal.

5. **Current Time**

   The current time is also given in packed decimal, in the format "HHMMSS.S". This gives the time of day at which IMS/VS received the entire message and enqueued it as input to the MPP.

6. **Input Message Sequence Number**

   This field contains the sequence number that IMS/VS assigned to the input message. The number is in binary. IMS/VS assigns sequence numbers by terminal; they are continuous since the most recent IMS/VS startup.

7. **Message Output Descriptor Name**

   You only use this field when you use MFS.

   If you use MFS, IMS/VS places in this area the name of the message output descriptor (MOD), when you issue a GU call. If your program encounters an error and needs to change the format of the screen in order to send an error message to the terminal, you can place the name of the MOD that you want IMS/VS to use in this area. Then you issue an ISRT call. If you don't use MFS, IMS/VS clears this area to blanks.

8. **User Identification**

   This field is used only with I/O PCBs; alternate PCBs do not use it.

   The contents of this field depend on the origin of the input message.

   - If the message was sent by a person at a terminal, and the system uses sign-ons, then IMS/VS places the user's identification in this field when the MPP issues a GU call for the first segment of the message.
   - If the message was sent by a person at a terminal, and the system does not use sign-ons, then IMS/VS places the name of the logical terminal in this field when the MPP issues a GU for the first segment of the message.
   - If the message was sent by a BMP, then IMS/VS places the name of the PSB of the BMP that sent the message in this field.

Alternate PCB Masks

When your program receives a message from a terminal, you usually send the reply to the input message back to the same terminal. But sometimes you want to send the reply to a different terminal, or to another program instead of the originating terminal. To do this, you use an alternate PCB.

An alternate PCB contains only three fields: the destination name, which is 8 bytes; a 2-byte field reserved for IMS/VS; and a 2-byte status code field. You place the logical terminal name or transaction code to which you want the message sent in the

destination name field. Figure 72 shows the format of an alternate PCB.

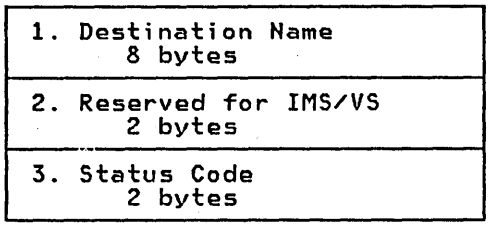

Figure 72. Alternate PCB Mask

Notes:

1. **Destination Name:** This field contains the name of the logical terminal or the transaction code that you want to send the output message to.

2. **Reserved for IMS/VS**

   This area is reserved for IMS/VS; the application program has no need to use it.

3. **Status Code**

   This field contains the 2-byte status code that describes the results of the call that used this PCB most recently.

An alternate PCB has several options:

- If you're only going to send output messages to one alternate destination, you can define the alternate PCB for that destination.

- If you're going to send output messages to more than one alternate destination, and you want to be able to change the destination of the alternate PCB, you define the alternate PCB as modifiable during PSB generation. Then, before you issue the ISRT call, you issue a CHNG call to set the destination of the alternate modifiable PCB for the program or terminal that you want to send the message to.

- There's a special kind of alternate PCB that you use in certain circumstances. This is an express alternate PCB. This is also defined during PSB generation, by specifying EXPRESS=YES.

  When you use an express alternate PCB, messages you send using that PCB are sent to their final destinations immediately. Messages sent with other PCBs are sent to temporary destinations until the program reaches a sync point. But messages sent with express PCBs are sent even if the program subsequently terminates abnormally, or if the program issues one of the rollback calls. Using an express alternate PCB in this kind of situation is a way to make sure that the program can notify the person at the terminal, even if the program terminates abnormally.

- You can also specify an alternate PCB as a response alternate PCB. "Sending Messages: ISRT, CHNG, and PURG" explains when and why you use this option.

MESSAGES

There are three kinds of messages that a person at a terminal can send to IMS/VS.

The first 8 characters of an IMS/VS message identify the message as one of three kinds:

- A **logical terminal name** in the first 8 bytes means that this is a message switch. A user at one logical terminal wants to send a message to another terminal. To do this, the person at the terminal enters the name of the receiving logical terminal followed by the message. The IMS/VS control region routes the message to the specified logical terminal. This kind of message does not involve an MPP.
- A **transaction code** in the first 8 bytes means that the message is for an application program. A transaction code is the way that IMS/VS identifies MPPs; when a person at a terminal needs a particular program to process requests, the person enters the transaction code for that application program.
- A "/" (slash) in the first byte means that the message is a command for IMS/VS. Issuing commands from an MPP is not part of a typical MPP. People design MPPs to issue commands when they want an MPP to perform some of the tasks that an operator at a terminal usually performs. This is called automated operator programming, and is described in the IMS/VS System Programming Reference Manual.

In addition to communicating with terminals, an MPP can send messages to and receive messages from other programs. To send a message to another program, you specify the transaction code for that program in the first field of the alternate PCB. IMS/VS then routes the message built using that alternate PCB to the application program associated with the transaction code you specified. This is called a program-to-program message switch.

The messages that your program receives and sends are made up of segments. This is the why you have two calls that you can use to retrieve messages. You use a GU call to retrieve the first segment of a new message, then you use GN calls to retrieve the remaining segments of the message. Figure 73 shows three messages. Message A contains 1 segment; message B contains 2 segments; and message C contains 3 segments.

To retrieve message A, you would only have to issue a GU call; but to retrieve messages B and C, you would issue one GU call to retrieve the first segment, then GN calls to retrieve the remaining segments. This assumes that you know how many segments each message contains. If you didn't know this, you would have to issue GN calls until IMS/VS returned a QD status code.

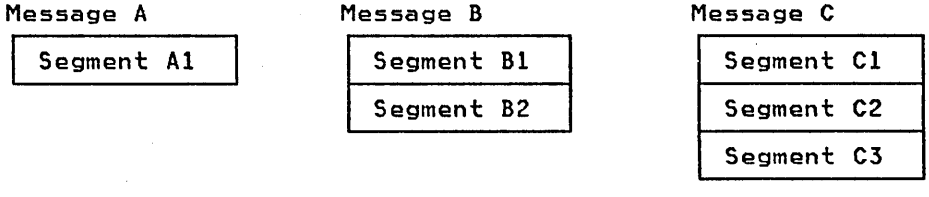

Figure 73. Message Segments

WHAT HAPPENS WHEN YOU PROCESS A MESSAGE

What a program does when it receives a message depends on the kind of message it receives. A transaction code associates a request for information from a terminal with the application program that can process and respond to that request. IMS/VS schedules an MPP when there are messages to be processed that contain the transaction code associated with that MPP.

For example, suppose you have an MPP that processes the transaction code "INQINV" for inventory inquiry. The MPP receives a request from a person at a terminal for information on the current inventory of certain parts. When the person at the terminal enters the transaction code for that application program, IMS/VS schedules the application program that can process the request.

The MPP works like this. When you enter "INQINV" and a part number, or several part numbers, the MPP returns to you the quantity on hand of each part, and the quantity on order.

When you enter INQINV at the terminal, IMS/VS puts the message on the message queue for the MPP that processes INQINV. Then, once IMS/VS has scheduled the MPP, the MPP issues GU and GN calls to retrieve the message, processes the request, and sends the output message to the queue for your logical terminal. (The logical terminal name is in the I/O PCB.) When the MPP sends the output message back, IMS/VS sends it the the queue for that logical terminal, and the message goes to the physical terminal. Figure 74 shows the flow of a message between the terminal and the MPP.

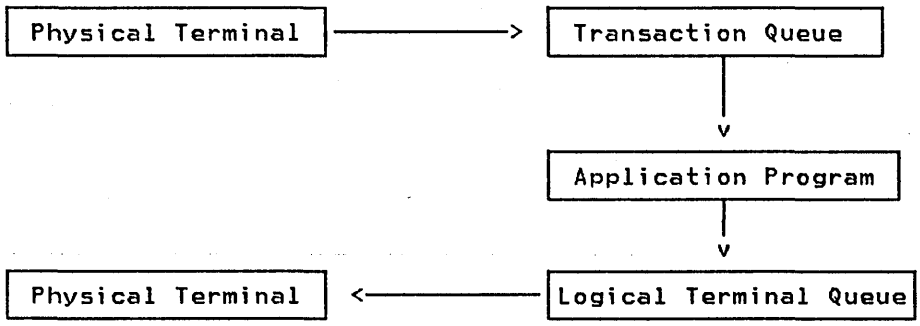

Figure 74. Transaction Message Flow

Figure 75 shows the calls you would use, the status codes, and what the input and output would look like. This example shows messages that contain three segments each to show you how you use GU and GN to retrieve messages, and how you insert multiple-segment messages. If input and output messages in this example were single segment, the program would have to issue only a GU to retrieve the entire message, and only one ISRT to send the message.

The message formats shown in this example are examples only. Not all messages are in this format. When the program receives the input message in the I/O area, the first 2 bytes (LL) of each segment contain the length of that segment. For clarity, Figure 75 shows this length in decimal; in the input message, however, it's in binary. The second 2 bytes (ZZ) are reserved for IMS/VS. The text of the message follows the reserved two bytes. The first message segment contains the transaction code in the 8 bytes following the ZZ field. These are the first 8 bytes of the text portion of the message.

The format of the output messages is just the same. You don't need to include the name of the logical terminal because that's in the first 8 bytes of the I/O PCB. PART, QTY, and ON ORDER in Figure 75 are headings. They are not data that the program has to compute. They are values that you can define as constants that you want to appear on the terminal screen. To include headings in MFS output messages, you define them as literals.

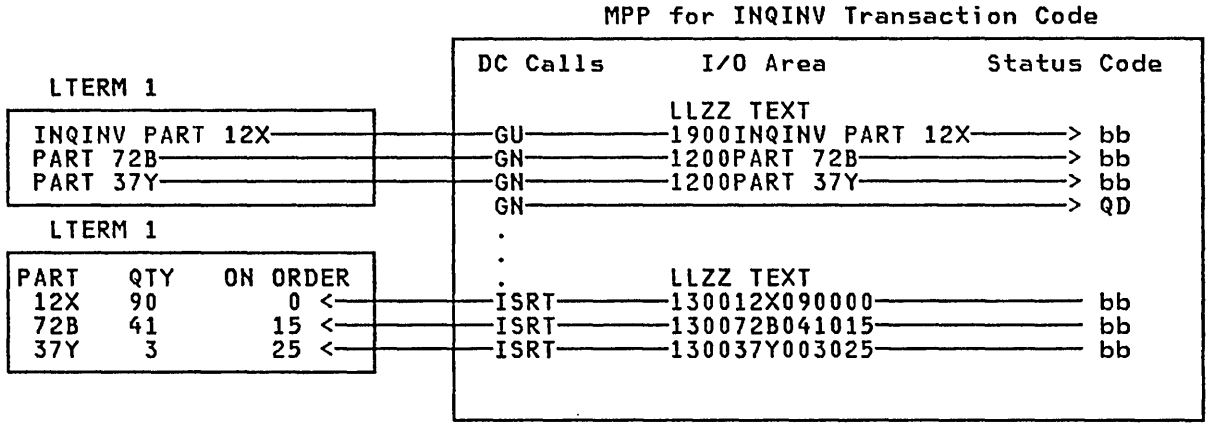

Figure 75. Inventory Inquiry MPP Example

To retrieve the messages from LTERM1, the application program issues a GU for the first segment of a message, then issues GN calls until IMS/VS returns a QD status code. This means that the program has retrieved all of the segments of that message.

WHAT INPUT MESSAGES LOOK LIKE

The input message that an MPP receives from a terminal or another program always has three fields: the length field, the "ZZ field," and the text field. Figure 76 shows the format of an input message. The contents of each of these fields is:

LL The length field is a 2-byte field that contains the total length of the message segment, including LL and ZZ. IMS/VS supplies this number in this field when you retrieve the input message.

ZZ The ZZ field is a 2-byte field that is reserved for IMS/VS. Your program has no need to modify this field.

Text This field contains the message text from the terminal to the application program. The first segment of a message may also contain the transaction code associated with the program in the beginning of the text portion of the message. Input messages don't have to include the transaction code, but, for consistency, it's a good idea. bytes of the message, as you can see in Figure 75.

LL	ZZ	Text
2	2	variable

Figure 76. Input Message Format

What the text field in the input message contains, and how the contents are formatted when your program receives the message, depends on the editing routine your program uses.

WHAT OUTPUT MESSAGES LOOK LIKE

The format of the output message that you build to send back to a terminal or another program is similar to the input message format. The difference is in what the fields contain.

Output messages contain four fields: a length field, a Z1 field, a Z2 field, and the text field. Figure 77 shows how these fields are arranged. Their contents are:

LL The length field is the same as the length field in the input message—it's 2 bytes and it contains the total length of the message in binary. The length includes the LL, Z1, and Z2 fields. For output message segments, you supply this length when you are ready to send the message segment.

Z1 The Z1 field is a 1-byte field that must contain binary zeros. It is reserved for IMS/VS.

Z2 The Z2 field is a 1-byte field that can contain special device-dependent instructions—such as instructions to ring the alarm bell, disconnect a switched line, or paging instructions—or device-dependent information—such as information about structured field data or bypassing MFS. For information on using this field, see "Message Formatting and Editing" in Chapter 3, "Design Considerations for IMS/VS Networks," in the IMS/VS System Administration Guide.

If you don't use any of these instructions, the Z2 field must contain a zero. If you're using MFS, this field contains the number of the option that is being used for this message. MFS options are explained in "Using Message Format Services."

Text The text portion of the message segment contains the data that you want to send to the logical terminal or to an application program. (Message texts are typically EBCDIC characters.) The length of the text depends on the data that you want to send.

LL	Z1	Z2	Text
2	1	1	variable

Figure 77. Output Message Format

HOW YOU EDIT YOUR MESSAGES

When an MPP passes messages to and from a terminal, IMS/VS edits the messages before the program receives the message from the terminal, and before the terminal receives the message from the application program. IMS/VS gives you a lot of choices about how you want your messages to appear both on the terminal screen and in the program's I/O area; you may need to know about the editing routines that have been specified for your program, and how they affect your programming.

USING MESSAGE FORMAT SERVICES

Message Format Services, or MFS, is a part of IMS/VS that uses control blocks that you define to format messages between a terminal and an MPP. The MFS control blocks indicate to IMS/VS how you want your input and output messages arranged:

- For input messages, MFS control blocks define how you want the message that the terminal sends to your MPP to be arranged in the MPP's I/O area.
- For output messages, MFS control blocks define how you want the message that your MPP sends to the terminal to be arranged

on the screen or at the printer. You can also define words or other data that will appear on the screen (as headings, for example) but won't appear in the program's I/O area. This kind of data is called a literal.

Terminals and MFS

Whether or not your program uses MFS depends on the type of terminals and secondary logical units (SLUs) your network uses. The decisions about using MFS are high-level design decisions that are separate from the tasks of application design and application programming; many installations that use MFS have a specialist who designs MFS screens and message formats for all applications that use MFS.

MFS makes it possible for an MPP to communicate with different types of terminals without having to change the way it reads and builds messages. When the MPP receives a message from a terminal, how the message appears in the MPP's I/O area doesn't depend on what kind of terminal sent it; it just depends on the MFS options that you've specified for the program. So if the next message that the MPP received was from a different type of terminal, you wouldn't have to do anything to the MPP; MFS shields the MPP from the physical device that's sending the message in the same way that a DB PCB shields the program from what the data in the data base actually looks like and how it is stored.

An MFS Example

The way that MFS does this is to use control blocks that format the input and output for the program and the terminal. There are four control blocks that do this:

- The device input format, or DIF, tells MFS the format that the data from the screen will be in.
- The message input descriptor, or MID, indicates to MFS how you want the data from the screen to appear in the MPP's I/O area.
- The message output descriptor, or MOD, defines the layout of the data that MFS receives from the MPP.
- The device output format, or DOF, defines how you want the data to be formatted at the screen or the printer.

One way to look at this is to look at a message from the time a person enters it at a terminal to the time the MPP processes the message and sends the reply back to the terminal. You can use MFS with both display terminals and printer terminals; for clarity in this example, however, assume you are using a display terminal.

Suppose you have an MPP that answers this request:

List the employees who have the skill "ENGINEER" with a skill level of "3." List only those employees who have been with the firm for at least 4 years.

To enter the request from a display terminal, you issue the format command (/FORMAT) and the MOD name. This formats the screen in the way defined by the MOD you supply. When you enter the MOD name, the screen contains only literals and no output data from an application program. At this stage, there is no MPP involved. Suppose the name of the MOD that formats the screen for this request is LE, for "locate employee." You would then enter this:

```
/FORMAT LE
```

Once you've done this, IMS/VS locates the MFS MOD control block with the name LE and arranges your screen in the format defined by the DOF. Your screen then looks like this:

```
SKILL
LEVEL
YEARS
        LOCEMP
```

The DOF defines a terminal format that asks you to qualify your request for an employee by giving the skill, level, and number of years of service of the employee you want. LOCEMP is the transaction code that is associated with the MPP that can process this request. When you enter the MOD name, the tran code is included in the first screen format that is displayed for you. This means that you don't have to know the name of the program that processes your request; you just need the name of the MOD that formats the screen.

Once you have the screen format, you can enter your request. There are four stages in sending a message to the program and receiving the reply.

1. First you enter the information at the terminal. In this example, you do this by entering the values of the qualifications that IMS/VS has given you on the screen: the skill is engineer, the level of skill is 3, and the employee has been with the firm for at least 4 years.

   When you enter your request, your screen would contain this data:

   ```
   SKILL   ENG
   LEVEL   3
   YEARS   4
             LOCEMP
   ```

2. When IMS/VS receives this data, MFS uses the DIF and the MID control blocks to translate the data from the way you entered it on the terminal screen to the way that the application program is expecting to receive it. The DIF control block tells MFS how the data is going to come in from the terminal. The MID control block tells MFS how the application program is expecting to receive the data. When the application program issues a message call, IMS/VS places the "translated" message in the program's I/O area.

   When the MPP receives the message in its I/O area, the message looks like this:

   ```
   LOCEMP  ENG0304
   ```

   "LOCEMP" is the transaction code. The name of the logical terminal does not appear in the message itself; IMS/VS places it in the first field of the I/O PCB.

3. The MPP processes the message, including any required data base access, and builds the output message in its I/O area.

   Suppose there is more than one employee who meets these qualifications. The MPP can use one message segment for each employee. After retrieving the information from the data base, the program builds the output message segment for the first employee. The program's I/O area contains:

   ```
   LLZZJONES,CE 3294
   ```

   When the program sends the second segment, the I/O area would contain:

   ```
   LLZZBAKER,KT 4105
   ```

4. When the application program sends the message back to the terminal, MFS translates the message again, this time from the application program format to the format in which the terminal expects the data.

The MOD tells MFS the format that the message will be in when it comes from the application program's I/O area. The DOF tells MFS how the message is supposed to look on the terminal screen. MFS translates the message and IMS/VS displays the translated message on the terminal screen. The screen would look like this:

```
SKILL       ENG
NAME        NO
JONES,CE    3294
BAKER,KT    4105
```

MFS Input Message Formats

When you define a message to MFS, you do so in fields—just as you define fields within a data base segment. When you define the fields that make up a message segment, you give MFS information such as:

- The length of the field
- The fill character to use when the length of the input data is less than the length that has been defined for the field
- Whether the data in the field should be left-justified or right-justified
- If the field is truncated, whether it should be truncated on the left or right

The order and length of these fields within the message segment depend on the MFS option that your program is using. You specify the MFS option in the MID. The decision about which option to use for an application program is a design decision that is based on how complex the input data is, and how much it will vary; the language the application program is written in; and how complex the application program is. There are also performance factors involved in this decision; the section called "Performance Effects" in Chapter 3 of the MFS User's Guide describes these considerations.

The Z2 field in MFS messages contains the MFS formatting option that is being used to format the messages to and from your program. If something is wrong in the way that IMS/VS returns the messages to your I/O area, and you suspect that the problem might be with the MFS option being used, you can check this field to see if IMS/VS is using the correct option. An X'00' in this field means that MFS did not format the message at all.

One way to understand how each of the MFS options formats your input and output messages is to look at examples of each option. Suppose you have defined the four message segments shown in Figure 78. Each of the segments contains a 2-byte length field and a 2-byte ZZ field. The first segment contains the transaction code that the person at the terminal entered to invoke the application program. The number of bytes defined for each field is given below the name of the field.

Segment 1

LL	ZZ			
0027	XXXX	TRANCODE	PATIENT#	NAME
2	2	8	5	10

Segment 2

0054	xxxx	ADDRESS
2	2	50

Segment 3

0016	xxxx	CHARGES	PAYMENTS
2	2	6	6

Segment 4

0024	xxxx	TREATMNT	DOCTOR
2	2	10	10

Figure 78. Message Segment Formats

For these examples, assume that:

- The transaction code has been defined in the MID as a literal. A literal is a field in the output message from the application program or the input message from the terminal, that contains constant data.
- All of the fields are to be left-justified.
- The fill character has been defined to be blank. When the length of the data in a field is less than the length that has been defined for that field, MFS pads the field with fill characters. Fill characters can be:
  - Blanks
  - An EBCDIC character
  - An EBCDIC graphic character
  - A null, specified as X'3F'

  When you specify that the fill character is to be a null, MFS compresses the field to the length of the data, if that length is less than the field length.

  The fields of the message segments are arranged on the terminal screen in the format shown in Figure 79. For example, assume the person enters the name of a patient, and the charges and payments associated with that patient.

```
PATIENT#:              NAME:  MC ROSS
ADDRESS:
CHARGES: 106.50        PAYMENTS:  90.00
TREATMENT:
DOCTOR:
```

Figure 79. Terminal Screen for MFS Example

OPTION 1 FORMAT: The way in which option 1 formats messages depends on whether or not you have defined a null as being the fill character for any of the fields in the segment.

If none of the fields in the message has been defined as having a fill character of null:

- The program receives all of the segments in the message.
- Each segment is the length that was specified for it in the MID.
- Each segment contains all of its fields.
- Each field contains data, data and fill characters, or all fill characters.

Figure 80 shows what the message segments that the application program received would look like.

Segment 1

LL	Z				
0027	XX	01	TRANCODE	blanks	MCROSSbbbb
2	1	1	8	5	10

Segment 2

0054	XX	01	blanks
2	1	1	50

Segment 3

0016	XX	01	010650	009000
2	1	1	6	6

Segment 4

0024	XX	01	blanks	blanks
2	1	1	10	10

Figure 80. Option 1 Message Format

The message format for option 1 output messages is the same as the input message format. The program builds output messages in an I/O

area in the format shown above. The program can truncate or omit fields in one of two ways:

- Inserting a short segment
- Placing a null character in the field

If one or more of the fields are defined as having a null fill character, the message is a little different. In this case, the message has these characteristics:

- If a field has been defined as having a fill character of null and there is no data from the terminal, the field is eliminated from the message segment.
- If this is true for all of the fields in a segment and none of the fields contains any literals, the segment is eliminated from the message.
- If this is true for only some of the fields in a segment, the field(s) containing nulls is eliminated from the segment and the relative positions of the fields within the segments are changed.
- When the length of the data that's received from the originating terminal is less than the length that's been defined for the field, the field is truncated to the length of the data.

Option 1 is a good choice for programs that will be receiving and transmitting most of the fields in the message segments.

OPTION 2 FORMAT: Option 2 formats messages in the same way that option 1 does, unless the segment contains no input data from the terminal after IMS/VS has removed the literals. If this is true, and if there are no more segments in the message that contain input data from the terminal, IMS/VS ends the message. The last segment that the program receives is the last segment that contains input data from the terminal.

Sometimes a segment that doesn't have any input data from the terminal is followed by segments that do contain input data from the terminal. When this happens, MFS gives the program the length field and the Z fields for the segment, followed by a 1-byte field containing X'3F'. This indicates to the program that this is a null segment.

If the message segments shown in Figure 78 were formatted by option 2, they would be in the format shown in Figure 81.

Segment 1

LL	Z	Z			
0027	XX	02	TRANCODE	blanks	MCROSS
2	1	1	8	5	10

Segment 2

0005	XX	02	3F
2	1	1	1

Segment 3

0016	XX	02	010650	0090000
2	1	1		6

Figure 81. Option 2 Message Format

Segment 2 in Figure 81 contains only an X'3F' because that segment is null, but Segment 3 contains data. Segment 4 is not part of the message because it is null.

Option 2 is a good choice when the program processes multisegment messages where most of the fields are transmitted but some of the segments will be omitted.

OPTION 3 FORMAT: When you use option 3, the program receives only those fields that have been received from the terminal. The program receives only segments that contain fields received from the originating terminal. Segments and fields can be of variable length if you've defined option 3 as having a null fill character.

A segment in an option 3 message is identified by its relative segment number—in other words, what position in the message it occupies. The fields within a segment are identified by their offset count within the segment. For example, the NAME field in Segment 1 has an offset value of 17. The offset is the sum of the fields that come before that field in the segment.

Option 3 messages do not contain literals defined in the MID. This means that the transaction code is removed from the message, except during a conversation. If the transaction that the program is processing is a conversational transaction, the transaction code is not removed from the message. The transaction code still appears in the SPA as well.

Each segment that the program receives contains the relative number of this segment in the message. In addition, each field within the segment is preceded by two fields:

- A 2-byte length field. The length given here includes the length field itself, the 2-byte relative field offset, and the data in the field.
- A 2-byte relative field offset, giving the field's position in the segment as defined in the MID

These two fields are followed by the field data. MFS includes these fields for each field that is returned to the application program.

Figure 82 shows the message segments that the program would receive if option 3 were being used to format the program's

messages. The notes following the figure explains the letters A, B, C, D, and E above Segment 1 and Segment 2.

Segment 1

LL	Z	Z	A	B	C	D
0020	XX	03	0001	0014	0017	MCROSS
2	1	1	2	2	2	10

Segment 3

LL	Z	Z	A	B	C	D	B	C	D
0000	XX	03	0003	0010	0004	010650	0010	0010	009000
2	1	1	2	2	2	6	2	2	6

Figure 82. Option 3 Message Format

Notes:

1. The fields marked A are the fields containing the relative segment number. This number gives the segment's position within the message.
2. The fields marked B are the fields containing the field length. This length is the sum of the lengths of the B field (2 bytes), the C field (3 bytes) and the D field (the length of the data).
3. The fields marked C are the fields containing the relative field offset. This gives each field's position within the segment.
4. The fields marked D are the fields containing the data from the terminal. In this example, the fill character was defined as blank, so the data field is always its defined length. IMS/VS doesn't truncate it. If you define the fill character as null, the lengths of the data fields can differ from the lengths defined for them in the segment. With a null fill character, if the length of the data from the terminal is less than the length defined for the field, IMS/VS truncates the field to the length of the data. Using a null fill with option 3 cuts down on the space required for the message even further.

Option 3 is a good choice when the program will be receiving and transmitting only a few of the fields within a segment.

For details on using MFS and MFS options refer to the IMS/VS MFS User's Guide.

MFS Output Message Formats

For output messages, you define to MFS what it is to receive from your program. If you use option 1 or option 2, the format is the same for output messages as it is for input messages. You present all fields and segments to MFS. You may present null segments. All fields in output messages are fixed length and fixed position. The only difference is that output messages do not contain option numbers.

Option 3 output messages don't contain option numbers in them. Other than that, they are similar to input messages. The program submits the fields as required in their segments with the position information.

USING BASIC EDIT

If you don't use MFS, IMS/VS does some editing for you automatically. How much IMS/VS does to each message segment is different for the first message segment and subsequent message segments.

Editing Input Messages

When IMS/VS receives the first segment of an input message for your application program, IMS/VS removes:

- Leading and trailing control characters
- Leading and trailing blanks
- Backspaces (from a printer terminal)

If the message segment contains a password, IMS/VS edits the segment by:

- Removing the password and inserting a blank where the password was.
- Removing the password if the first character of the text is a blank. IMS/VS doesn't insert the blank.
- Left-justifying the text of the segment.

For subsequent input message segments, IMS/VS doesn't remove leading blanks from the text of the message; other than that, IMS/VS does all of the things listed above.

Editing Output Messages

When you don't use MFS, you have to provide the necessary horizontal and vertical control characters that are necessary to format your output messages in the messages themselves.

To print your output at a printer terminal, you include these control characters where you need them within the text of the message:

X'05' Skip to the tab stop, but stay on the same line.

X'15' Start a new line at the left margin.

X'25' Skip to a new line, but stay at the same place horizontally.

If you want to skip multiple lines, you can start a new line (X'15'), then skip as many lines as necessary (X'25').

RETRIEVING MESSAGES

An MPP uses two calls to retrieve input messages from the message queue: GU and GN.

When you issue a successful GU or GN, IMS/VS returns the message segment to the I/O area that you specify in the call. The only parameters on GU and GN calls are the call function, the I/O PCB, and the I/O area; there are no SSAs in message calls. Message segments are not all the same length; because of this, your I/O area must be long enough to hold the longest segment that your program can receive. The first 2 bytes of the segment contain the length of the segment.

RETRIEVING THE FIRST SEGMENT: GU

Because of message priming, your program should always issue a GU call to the message queue as the first call in the program. When IMS/VS schedules an MPP, IMS/VS transfers the first segment of the first message to the message processing region. When the MPP issues the GU for the first message, IMS/VS already has the message waiting. If the program doesn't issue a GU message call as the first call of the program, IMS/VS has to transfer the message again, and the efficiency provided by message priming is lost.

If an MPP responds to more than one transaction code, the MPP has to examine the text of the input message to determine what processing the message requires. After a successful GU call, IMS/VS places the following information in the I/O PCB mask:

- The name of the logical terminal that sent the message.
- The status code for this call. A blank status code means that the GU call was successful; a QC status code means that there are no more messages for the application program to process. You should terminate your MPP as soon as you receive a QC status code.
- The input prefix, giving the current date, time, and sequence number for the message.
- The MOD name.
- The userid of the person at the terminal, or if userids are not used in the system, the logical terminal name. If the message is from a BMP, IMS/VS places the name of the BMP's PSB in this field.

RETRIEVING SUBSEQUENT SEGMENTS: GN

If you are processing messages that contain more than one segment, you use a different call to retrieve the second and subsequent segments of the message. After you've retrieved the first segment of the message, you issue GN calls to retrieve the rest of the message. IMS/VS returns one segment to your I/O area each time you issue a GN, until you have retrieved all of the segments of that message. IMS/VS returns a status code of QD to indicate to you that you have retrieved all of the segments of that message. For an example of this, refer to Figure 75.

SENDING MESSAGES: ISRT, CHNG, AND PURG

When your program is ready to send a message, you send the message by issuing an ISRT call. You send one segment of the message at a time. Before you issue the ISRT call for each segment of the message, you have to build the message segment in your program's I/O area.

You can send a message to the terminal that sent the input message to you, or you can send a message to another terminal or another application program. When you want to send an output message to an alternate destination, you use an alternate PCB.

REPLYING TO THE SENDER

To send a reply to the terminal that sent the message, all you do is build the message and issue the ISRT call for each message segment, using the I/O PCB.

When you use the ISRT call to answer the originating terminal, you name the I/O PCB and the I/O area that contains the message segment in the call. You can also specify a MOD name in the I/O PCB if you want to change the screen format. One situation in which you might want to do this is if your program detects an

error and needs to notify the person at the terminal. When you send the message, you can specify a MOD name that formats the screen to receive the error message.

In the example explained in "An MFS Example," the program would move the first employee segment (JONES,CE 3294) to the I/O area and issue the ISRT call, then move the second segment to the I/O area and issue the ISRT call for it.

SENDING MESSAGES TO OTHER TERMINALS

To reply to a different terminal, you also use the ISRT call, but you use an alternate PCB instead of the I/O PCB.

Just as the I/O PCB represents the terminal that sent the message, an alternate PCB represents the terminal to which you want to send the message.

To One Alternate Terminal

If you're going to send messages to only one alternate terminal, you can define the alternate PCB for that terminal during PSB generation. When you define an alternate PCB for a particular destination, you cannot change that destination during program execution; each time you issue an ISRT call that references that PCB, the message goes to the logical terminal whose name was specified for the alternate PCB. To send a message to that terminal, you place one message segment at a time in the I/O area, and issue an ISRT call referring to the alternate PCB, instead of the I/O PCB.

To Several Alternate Terminals

If you want to send messages to several different terminals, you can define the alternate PCB as modifiable during PSB generation. This means that you can use this alternate PCB to send messages to different terminals instead of to just one. In other words, the alternate PCB doesn't represent just one alternate terminal; you can change the destination while your program is running.

Before you can set or change the destination of an alternate PCB, you have to indicate to IMS/VS that the message you've been building so far with that PCB is finished. You do this by issuing a PURG call.

PURG allows you to send multiple output messages while processing one input message. When you don't use PURG, IMS/VS groups message segments into a message and sends them when the program issues a GU for a new message, terminates, or reaches a sync point. A PURG call tells IMS/VS that the message built against this alternate PCB (by issuing one ISRT call per message segment) is complete. IMS/VS collects the message segments that you've inserted to one PCB as one message and sends it to the destination represented by the alternate PCB you've referenced.

A PURG call that doesn't contain the address of an I/O area indicates to IMS/VS that this message is complete. If you include an I/O area in the call, PURG acts as an ISRT call as well. IMS/VS treats the data in the I/O area as the first segment of a new message. When you include an I/O area on a PURG call, you can also include a MOD name to change the format of the screen for this message. You can specify a MOD name on any PURG call in a series, but you can specify the MOD name only once.

To set the destination of a modifiable alternate PCB during program execution, you use a change call, or CHNG. When you issue the CHNG call you supply the name of the logical terminal that you want to send the message to. The alternate PCB you use then remains set with that destination until you do one of the following:

- You issue another CHNG call to reset the destination.
- You issue another GU to the message queue to start processing a new message. In this case, the name still appears in the alternate PCB, even though it is no longer valid.
- You terminate your program. When you do this, IMS/VS resets the destination to blanks.

The first 8 bytes of the alternate PCB contain the name of the logical terminal that you want to send the message to.

When you issue a CHNG call, you give IMS/VS the address of the alternate PCB you're using, and the destination name you want set for that alternate PCB.

When you use the PURG call, you give IMS/VS only the address of the alternate PCB. IMS/VS sends the message you've built using that PCB.

To indicate an error situation you can send a message by issuing an ISRT call followed by a PURG call against an express PCB. These calls send the message to its final destination immediately.

For example, suppose the program goes through these steps:

- The program issues a GU call (and GN calls, if necessary) to retrieve an input message.
- While processing the message, the program encounters an abnormal situation.
- The program issues a PURG call to indicate to IMS/VS the start of a new message.
- The program issues a CHNG call to set the destination of an express PCB to the name of the originating logical terminal. The program can get this name from the first 8 bytes of the I/O PCB.
- The program issues ISRT calls as necessary to send message segments. The ISRT call(s) reference the express PCB.
- The program issues a PURG call referencing the express PCB. IMS/VS then sends the message to its final destination.
- The program can then terminate abnormally, or it can issue a ROLL or ROLB call to back out its data base updates and cancel the output messages its created since the last sync point.

If your output messages contained three segments, and you used the PURG call to indicate the end of a message (and not to send the next message segment), you could use this call sequence:

```
PURG ALTPCB1
CHNG ALTPCB1, LTERMA
ISRT ALTPCB1, SEG1
ISRT ALTPCB1, SEG2
ISRT ALTPCB1, SEG3
PURG ALTPCB1
CHNG ALTPCB1, LTERMB
ISRT ALTPCB1, SEG4
ISRT ALTPCB1, SEG5
ISRT ALTPCB1, SEG6
```

SENDING MESSAGES TO OTHER APPLICATION PROGRAMS

When one MPP sends a message to another online program (another MPP, a message-driven Fast Path program, or a transaction-oriented BMP), that's called a program-to-program message switch. To do this you use an alternate PCB, and you have some of the same options that you have when you use an alternate

PCB to send messages to alternate terminals. If you'll only send messages to one application program, then you can define the alternate PCB with the transaction code for that application program during PSB generation. If you want to send messages to more than one application program, you can define the alternate PCB as modifiable.

If you use an alternate modifiable PCB, IMS/VS does some security checking when you issue the CHNG call to set the destination of the alternate modifiable PCB. The terminal that enters the transaction code that causes the message switch must be authorized to enter the transaction code that the CHNG call places in the alternate modifiable PCB. IMS/VS doesn't do any security checking when you issue the ISRT call.

The things you have to consider when you do a program-to-program message switch aren't very different from the things you consider when you communicate with a logical terminal. The things you have to remember are:

- The program you're sending the message to (for clarity, assume you're sending a message to an MPP called Program B) must have an I/O area large enough to hold the largest segment that you're sending.
- You must use an alternate PCB, not the I/O PCB, to send the message.
- If the alternate PCB is modifiable, you issue a CHNG call before issuing the ISRT call to place Program B's transaction code in the first field of the alternate PCB. If the alternate PCB was set to this transaction code in the PSBGEN, then you just issue the ISRT call.
- IMS/VS must know the transaction code. This means that it was defined at system definition.
- Figure 83 shows the format for an output message to an application program.

LL	Z1	Z2	Text
2	1	1	variable

Figure 83. Message Format for Program-to-Program Message Switch

As you can see, the format is the same as it is for output messages to terminals. Z1 and Z2 are fields that must contain binary zeros. These fields are reserved for IMS/VS. The text field contains the message segment that you want to send to the application program.

You should include Program B's transaction code as part of the message text because IMS/VS does not automatically include the transaction code in switched messages. Including the transaction code in the message text keeps all messages in the same format, whether they're sent from terminals or other programs.

COMMUNICATING WITH OTHER IMS/VS SYSTEMS

In addition to communicating with programs and terminals in your IMS/VS system, your program can communicate with terminals and programs in other IMS/VS systems through Multiple Systems Coupling, or MSC. MSC makes this possible by establishing links between two or more separate IMS/VS systems. The terminals and transaction codes within each IMS/VS system are defined as belonging to that system. Terminals and transaction codes within your system are called "local," and terminals and transaction

codes defined in other IMS/VS systems connected by MSC links are called "remote."

For the most part, communicating with a remote terminal or program doesn't affect how you code your program; MSC handles the message routing between systems. For example, if you receive an input message from a remote terminal, and you want to reply to that terminal, you issue an ISRT call against the I/O PCB—just as you would to reply to a terminal in your system. There are situations, however, in which MSC does affect your programming:

- When your program needs to know if an input message is from a remote terminal or a local terminal. For example, if two terminals in separate IMS/VS systems had the same logical terminal name, your program's processing might be affected by knowing which system sent the message.
- When you want to send a message to an alternate destination in another IMS/VS system.

Directed routing makes it possible for your program to find out whether an input message is from your system or a remote system, and to set the destination of an output message for an alternate destination in another IMS/VS system. With directed routing, you can send a message to an alternate destination in another IMS/VS system, even if that destination is not defined in your system as remote.

RECEIVING MESSAGES FROM OTHER IMS/VS SYSTEMS

When an application program retrieves an input message, the program can determine whether the input message is from a terminal or program in its IMS/VS system, or from a terminal or program in another IMS/VS system. There may be situations in which the application program's processing is changed if the input message is from a remote terminal, rather than from a local terminal.

For example, suppose that your IMS/VS system is system A, and you are linked to another IMS/VS system called system B. MSC links are "one-way" links. The link from system A to system B is called LINK1, and the link from system B to system A is called LINK2. The application program named MPP1 runs in system A; the logical terminal name of the master terminals in both systems is MASTER. Figure 84 shows systems A and B.

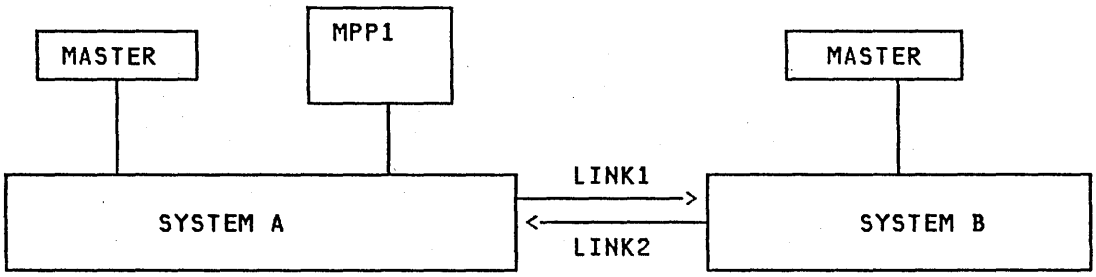

Figure 84. MSC Example

If the MASTER terminal in system B sends the message, "System is shutting down" to MPP1 in system A, MPP1 needs to know that the message is from MASTER in system B and not MASTER in system A.

If you have specified ROUTING=YES on the TRANSACT macro during IMS/VS system definition, IMS/VS does two things to indicate to

the program that the message is from a terminal in another IMS/VS system.

First, instead of placing the logical terminal name in the first field of the I/O PCB, IMS/VS places the name of the MSC logical link in this field. In the example, this is "LINK2." This is the logical link name that was specified on the MSNAME macro at system definition.

Second, IMS/VS turns on a bit in the field of the I/O PCB that is reserved for IMS/VS. This is the "01" bit in this 2-byte field. Figure 85 shows the location of this bit within the reserved field.

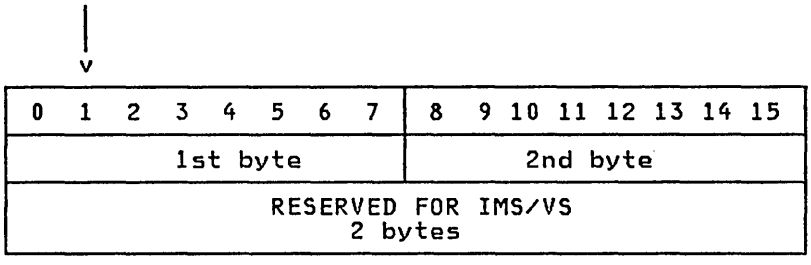

Figure 85. Directed Routing Bit in I/O PCB

The reason that MPP1 needs to test this bit is that if the message is from MASTER in system A, MPP1 should terminate immediately. If, on the other hand, the message is from MASTER in system B, MPP1 could perform some local processing and send transactions for system B to a message queue so that those transactions could be processed later on, when system B is up.

SENDING MESSAGES TO ALTERNATE DESTINATIONS IN OTHER IMS/VS SYSTEMS

To send an output message to an alternate terminal in another IMS/VS system, your system must have an MSC link with the system to which you want to send the message. To do this, you issue a CHNG call against an alternate PCB and supply the name of the MSC link (in the example used so far this would be LINK1) that connects the two IMS/VS systems. For example, if you were sending a message to TERMINAL 1 in system B after you had received a message from some other terminal, you would first issue this CHNG call:

```
CHNG  altpcb, LINK1
```

After issuing the CHNG call, you would issue an ISRT call (or calls) to send the message—just as you would to send a message to a local terminal.

Figure 86 shows the format of this type of output message.

field	LL	ZZ	DESTNAME	b	TEXT
length	2	2	1-8	1	variable

Figure 86. Directed Routing Output Message Format

The LL and ZZ fields are 2 bytes each; LL contains the total length of the message. This is the sum of all of the fields in the message, including the LL field. ZZ is reserved for IMS/VS. The destination name, DESTNAME, is the name of the logical terminal to which you're sending the message. This field is from 1 to 8 bytes long and it must be followed by a blank. The TEXT field contains the text of the message; its length depends on the message you're sending.

If the destination in the other system is a terminal, IMS/VS removes the DESTNAME from the message. If the destination in the other system is a program IMS/VS does not remove the DESTNAME.

If there is a security violation in your message, MSC detects it in the receiving system (in this case, system B), and reports it to the person at the originating terminal (system A).

CONVERSATIONS

The difference between a conversational program and a regular MPP is that a conversational program doesn't process the entire transaction at once. You use conversational processing when there are several parts to one transaction.

A nonconversational program receives a message from a terminal, processes the request, and sends a message back to the terminal.

A conversational program, on the other hand, receives a message from a terminal, does some processing, and replies to the terminal, but the program saves the data from the transaction in a scratchpad area, or SPA. Then, when the person at the terminal enters more data, the program has the data it saved from the last message in the SPA, so it can continue processing the request without the person at the terminal having to enter the data again. A conversational program divides processing into a connected series of terminal-to-program-to-terminal cycles. One way to understand how the process works is to look at an example.

A CONVERSATIONAL EXAMPLE

For this example, suppose that you want to find out whether or not someone can receive a loan for a car. There are two parts to this inquiry. First, you need to give the name and address of the person requesting the loan, and the number of years for which the person wants the loan. After you give this information, IMS/VS asks you for the information on the car: model, year, and cost. You enter this information, IMS/VS invokes the program that processes this information, and the program responds as to whether or not the loan can be granted.

This process, assuming you use MFS, involves six parts:

1. First, you enter the format command (/FORMAT) and the MOD name. This says to IMS/VS, "Format the screen in the way defined by this MOD."

Suppose the MOD name is CL. Your screen would look like this:

```
/FORMAT CL
```

IMS/VS then takes that MOD from the MFS library and formats your screen in the way defined by the MOD. When the MOD for the car loan application formats your screen, it looks like this:

```
CARLOAN
NAME:
ADDRESS:
YEARS:
```

The word "CARLOAN" is the transaction code for this application. Each transaction code is associated with an application program, so when IMS/VS receives the transaction code "CARLOAN," that's what tells IMS/VS what application program to schedule for this request.

2. Next, you enter the information that is requested. Suppose the person's name is John Edwards, that his address is 463 Pinewood, and that he wants the loan for 5 years. When you enter this information, your screen would look like this:

```
CARLOAN
NAME: JOHN EDWARDS
ADDRESS: 463 PINEWOOD
YEARS: 5
```

3. When you enter this information, IMS/VS reads the transaction code, CARLOAN, and invokes the program that handles that transaction code. MFS formats the information from the screen for the MPP's I/O area by using the DIF and the MID.

   When the MPP issues a GU for a message, IMS/VS clears the SPA to binary zeros and passes it to the application program. IMS/VS gives the program the first input message segment when the program issues the GN after the GU.

4. Next, the MPP processes the input data from the terminal, and does two things. It moves the data that it will need to save to the SPA, and it builds the output message for the terminal in the I/O area. The information that the MPP saves in the SPA is the information the MPP will need when the second part of the request comes in from the terminal. You don't save information in the SPA that you can get from the data base. In this example, you would have to save only the name of the person applying for the loan. You save that so that if the person is granted the loan, the program will know what information to update in the data base.

   The program then issues an ISRT call to return the SPA to IMS/VS, and another ISRT call to send the output message to the terminal.

   The response that the MPP sends to the terminal gives IMS/VS the name of the MOD to format the screen for the next cycle of the conversation. In that cycle, you need to supply the model, year, and cost of the car that John Edwards wants to buy. Your screen would then look like this:

```
MODEL:
YEAR:
COST:
```

5. When you enter this information, IMS/VS again uses the DIF and MID associated with the tran code, and sends the information back to the MPP. The MPP hasn't been running all this time; when IMS/VS receives the terminal input with the tran code CARLOAN, IMS/VS invokes the MPP that processes that transaction again for this cycle of the conversation.

6. When IMS/VS invokes the MPP, IMS/VS returns the updated SPA to the MPP when the MPP issues a GU, then returns the message to the MPP when the MPP issues a GN. The MPP does the required processing (in this case, determining whether or not the loan can be granted and updating the data base if necessary), and is then ready to end the conversation. To do this, the MPP blanks out the transaction code in the SPA and inserts it back to IMS/VS, then sends a message to the terminal saying whether or not the loan has been granted.

This example is a simple one; its purpose isn't to show you everything that conversational processing can do. It is just meant to give you an understanding of what conversational processing involves.

CONVERSATIONAL STRUCTURE

How you structure your conversational program depends on the interactions between your program and the person at the terminal. To understand how to structure a conversational program, see "Appendix D. Sample Conversational MPP."

These are the things you need to know about and take into consideration before you can structure your program:

- **What should the program do in an error situation?**

  When a a program in a conversation terminates abnormally, IMS/VS backs out only the last cycle of the conversation. A cycle in a conversation is one terminal/program interaction. Because the conversation can terminate abnormally during any cycle, you should be aware of some things you can do that can make recovering the conversation easier:

  - The ROLB call can be used in conversational programs to back out data base updates that the program has made since the last sync point. ROLL can also be used in conversational programs, but ROLL terminates the conversation. "Using ROLB and ROLL in Conversations" explains how these calls work with conversational processing.

  - If possible, updating the data base should be part of the last cycle of the conversation so that you don't have different levels of data base updates resulting from the conversation.

  - If your program encounters an error situation and it has to terminate, it can use an express alternate PCB to send a message to the originating terminal, and, if you wish, to the master terminal operator. To do this, the program issues a CHNG call against the express alternate PCB and supplies the name of the logical terminal from the I/O PCB, then issues an ISRT call that references that PCB and the I/O area that contains the message. The program can then issue another CHNG call to set the destination of the express alternate PCB for the master terminal, and issue another ISRT call that references that PCB, and the I/O area that contains the output message.

    "Recovery Considerations in Conversations" contains more information about recovery-related design decisions in conversational processing.

- **Does your application program process each cycle of the conversation?**

  A conversation can be processed by several application programs, or it can be processed by only one program.

  If your program processes each stage of the conversation (in other words, your program processes each input message from

the terminal), the program has to know what stage of the conversation it's processing when it receives each input message. When the person at the terminal enters the transaction code that starts the conversation, IMS/VS clears the SPA to binary zeros and passes the SPA to the program when the program issues a GU call. On subsequent passes, however, the program has to be able to tell which stage of the conversation it's on so that it can branch to the section of the program that handles that processing.

One technique that the program can use to determine which cycle of the conversation it's processing is to keep a counter in the SPA. The program increments this counter at each stage of the conversation. Then, each time the program begins a new cycle of the conversation (by issuing a GU call to retrieve the SPA), the program can check the counter in the SPA to determine which cycle it's processing. The program can then branch to the appropriate section.

- **If your program passes control of the conversation to another conversation program, how should it do this?**

  Sometimes it is more efficient to use several application programs to process a conversation instead of one. This does not affect the person at the terminal; it depends on the processing that's required.

  In the car loan example, for instance, one MPP could handle the first part of the conversation (processing the name, address, and number of years), then another MPP could process the second part of the conversation (processing the data about the car and responding as to the status of the loan).

  A program can:

  - Reply to the originating terminal but specify that the next input message should go to another conversational program. This is called a deferred program switch.

  - Pass the SPA (and, optionally, a message) to another conversational program without responding to the terminal. In this case, it's the next program's responsiblity to respond to the originating terminal. This is called an immediate program switch.

A conversational program has seven main steps:

1. Retrieve the SPA and the message using GU and GN calls.

2. If your MPP is starting this conversation, test the variable area of the SPA for zeros to determine whether or not this is the beginning of the conversation. If the SPA doesn't contain zeros, it means that you started the conversation earlier and that you're now at a later stage in the conversation. If this is true, you would branch to the part of your program that processes this stage of the conversation to continue the conversation.

3. If another MPP has passed control to your MPP to continue the conversation, then you know that the SPA will contain the data you need to process the message, so you don't have to test it for zeros. You start processing the message right away.

4. Process the message, including any necessary data base access.

5. Send the output message to the terminal by using another ISRT call against the I/O PCB.

6. Store the data that your program (or the program that you pass control to) will need to continue processing in the SPA using an ISRT call to the I/O PCB. This step may precede step 5.

IMS/VS determines which segment is the SPA by examining the XXXX field of the segment shown in Figure 87.

7. To end the conversation, move blanks to the area of the SPA that contains the transaction code, then insert the SPA back to IMS/VS by issuing an ISRT call and referencing the I/O PCB.

If your MPP passes the conversation to another conversational program, then the steps after the program processes the message are a little different. "Passing the Conversation to Another Conversational Program" explains this.

Also, there is a special situation your program should be designed to handle. This occurs if the first GU call to the I/O PCB doesn't return a message to the application program. This can happen if the person at the terminal cancels the conversation by entering the /EXIT command before the program issues a GU call. (This happens if the message from this terminal was the only message in the message queue for the program.)

What the SPA Contains

The SPA that IMS/VS gives your program when you issue a GU contains the four parts listed below. Figure 87 shows the SPA format.

- A 2-byte length field ("LL") that gives the total length of the SPA. This length includes 2 bytes for the LL field.
- A 4-byte field reserved for IMS/VS ("XXXX") that your program must not modify.
- The 8-byte transaction code for this conversation.
- A work area that you will use to save the information that you'll need to continue the conversation. The length of this area depends on the length of the data you wish to save. This length is defined at system definition.

LL	XXXX	Trancode	User Work Area
2	4	8	variable

Figure 87. SPA Format

When your program retrieves the SPA with a GU to start the conversation, IMS/VS removes the transaction code from the message and in your first message segment you receive only the data from the message that the person at the terminal entered.

There are some restrictions about the way that an application program processes the SPA. They are:

- The program must not modify the first 6 bytes of the SPA (LL and XXXX). IMS/VS uses these fields to identify the SPA.
- If the program modifies the SPA, the program must return the SPA to IMS/VS (or, for a program switch, to the other program).
- The program must not return the SPA to IMS/VS more than once during one cycle of the conversation.
- The program must not insert the SPA to an alternate PCB that represents a nonconversational transaction code or a logical terminal. The program may use a response alternate PCB if it

represents that same physical terminal as the originating logical terminal.

Note: If you're using MFS, there are some situations in which IMS/VS doesn't remove the transaction code. Chapter 2, "Message Formatting Functions," in the IMS/VS Message Format Services User's Guide explains these situations.

What Messages Look Like in a Conversation

Conversational input messages are at least two segments, because the first segment contains the SPA; the input message starts in the second message segment.

The input message segment in a conversation contains only the data from the terminal. IMS/VS removes the transaction code from the input message and places it in the SPA. When the program issues the first GU, IMS/VS returns the SPA. To retrieve the first message segment, the program must issue a GN.

The format for the output messages that you send to the terminal is no different from the format for output messages in nonconversational programs.

Saving Information in the SPA

After you have processed the message and are ready to reply to the terminal, you can save the data you or the other program will need in the SPA. The part of the SPA in which you save data is the work area portion. You save this by using an ISRT call. This is a special use of the ISRT call in that you're not sending the SPA to a terminal, you're saving it for future use.

If your program processes each stage of the conversation, you just issue an ISRT call to the I/O PCB and give the name of the I/O area that contains the SPA. For example:

```
ISRT   I/O PCB, SPA, I/O AREA
```

This returns the updated SPA to IMS/VS so that IMS/VS can pass it to your program at the next cycle of the conversation.

If you don't modify the SPA, you don't have have to return it to IMS/VS.

REPLYING TO THE TERMINAL

For a conversation to continue, the originating terminal must receive a response to each of its input messages. The person at the terminal cannot enter any more data to be processed (except IMS/VS commands) until the response has been received at the terminal.

To continue the conversation, the program must respond to the originating terminal by issuing the required ISRT calls to send the output message to the terminal. To send a message to the originating terminal, the ISRT calls must reference either the I/O PCB or a response alternate PCB. You use a response alternate PCB in a conversation when the terminal you're responding to has two components—for example, a printer and a punch—and you want to send the output message to a component that's separate from the component that sent the input message. If the program references an alternate response PCB, the PCB must be defined for the same physical terminal as the logical terminal that sent the input message.

The program can send only one output message to the terminal for each input message. Output messages can contain multiple segments, but the program can't use the PURG call to send multiple output messages. If a conversational program issues a PURG call,

IMS/VS returns an AZ status code to the application program and doesn't process the call.

PASSING THE CONVERSATION TO ANOTHER CONVERSATIONAL PROGRAM

There are two ways in which a conversational program can pass the conversation to another conversational program:

- A deferred switch:

  The program can respond to the terminal but cause the next input from the terminal to go to another conversational program by:

  - Issuing an ISRT call against the I/O PCB to respond to the terminal
  - Placing the transaction code for the new conversational program in the SPA
  - Issuing an ISRT call referencing the I/O PCB and the SPA to return the SPA to IMS/VS

  IMS/VS then routes the next input message from the terminal to the program associated with the transaction code that was specified in the SPA. Other conversational programs can continue to make program switches by changing the transaction code in the SPA.

- An immediate switch:

  The program can pass the conversation directly to another conversational program by:

  - Issuing an ISRT call against the alternate PCB that has its destination set to the other conversational program. The first ISRT call must send the SPA to the other program, but the program passing control can issue subsequent ISRT calls to send a message to the new program.
  - If the program does this, it cannot return the SPA to IMS/VS or respond to the originating terminal.

  In an immediate switch, it's the new program's responsibility to respond to the terminal (or pass the conversation to a third program).

In addition, the person at the terminal can issue the command /SET CONV to change the next conversational transaction to process before continuing a conversation from the terminal.

There are some restrictions concerning the size of SPAs when you are passing a conversation from one program to another. Briefly, they are:

- If the program that processes the first cycle of the conversation uses a fixed-length SPA, all of the other programs that process the conversation must also use fixed-length SPAs. None of the programs in the conversation can pass the conversation to a program that uses a variable-length SPA.
- If the program that processes the first cycle of the conversation uses a variable-length SPA, the conversation can be processed by programs that use fixed- or variable-length SPAs.

"Things You Need to Know about the SPA" explains these restrictions in more detail.

Conversational Processing and MSC

If your installation has two or more IMS/VS systems, and they are linked to each other through MSC, a program in one system can process a conversation that originated in another system.

- All of the SPAs used in a conversation between two or more IMS/VS systems must be fixed-length SPAs of the same size.
- If a conversational program in system A issues an ISRT call that references a response alternate PCB in system B, system B does the necessary verification. This is because the destination is implicit in the input system. The verification that system B does includes determining whether or not the logical terminal that's represented by the response alternate PCB is assigned to the same physical terminal as the logical terminal that sent the input message. If it isn't, system B (the originating system) terminates the conversation abnormally without issuing a status code to the application program.
- Suppose program A processes a conversation that originates from a terminal in system B; Program A passes the conversation to another conversational program by changing the transaction code in the SPA. If the transaction code that program A supplies is invalid, system B (the originating system) terminates the conversation abnormally without returning a status code to the application program.

Ending the Conversation

To end the conversation, a program blanks out the transaction code in the SPA and returns it to IMS/VS by issuing an ISRT call and referencing the I/O PCB and the SPA. This terminates the conversation as soon as the terminal has received the response.

The program can also end the conversation by placing a nonconversational transaction code in the transaction field of the SPA and returning the SPA to IMS/VS. This causes the conversation to remain active until the person at the terminal has entered the next message. The transaction code will be inserted from the SPA into the first segment of the input message. IMS/VS then routes this message from the terminal to the MPP or BMP that processes the transaction code that was specified in the SPA.

In addition to being ended by the program, a conversation can be ended by the person at the originating terminal, the master terminal operator, and IMS/VS.

- The person at the originating terminal can end the conversation by issuing one of several commands:
  - /EXIT. The person at the terminal can enter the /EXIT command by itself, or the /EXIT command followed by the conversational identification number assigned by the IMS/VS sytem.
  - /HOLD. The /HOLD command stops the conversation temporarily to allow the person at the terminal to enter other transactions while IMS/VS holds the conversation. When IMS/VS responds to the /HOLD command, IMS/VS supplies an identifier that the person at the terminal can later use to reactivate the conversation. The /RELEASE command followed by this identifier reactivates the conversation.
- The master terminal operator can end the conversation by entering a /START LINE command (without specifying a PTERM) for the terminal in the conversation.
- IMS/VS ends a conversation if, after the program successfully issues a GU call or an ISRT call to return the SPA, the

program doesn't send a response to the terminal. In this situation, IMS/VS sends the message "DFS2171I NO RESPONSE, CONVERSATION TERMINATED" to the terminal. IMS/VS then terminates the conversation and performs sync point processing for the application program.

ISSUING COMMANDS

There are two calls a program can use to issue and receive responses from commands. CMD sends the command to IMS/VS; and GCMD retrieves the message segments that IMS/VS has sent in response to the command. The most common use for these calls is in a program that performs some of the tasks that are otherwise performed by someone at a terminal. Designing a program that does this is called automated operator programming. This section explains how you use the calls that make this possible. For information on the uses and techniques of automated operator programming, see Chapter 7, "Automated Operator Programming," in the IMS/VS System Programming Reference Manual. To issue a command, the program uses the command call (CMD). When you issue this call, IMS/VS passes the command you supply to the IMS/VS control region to be executed. Before you issue the CMD call, you have to place the command you want executed in the I/O area that you point to in the call. IMS/VS places your program in a wait state until IMS/VS has executed the command.

When IMS/VS receives the command, IMS/VS returns a response to your program. This response from IMS/VS means that IMS/VS has received and executed the command—unless the command the program issued is a delayed response command. IMS/VS returns the first segment of its response to the application program's I/O area. After receiving the command, IMS/VS takes your program out of wait state.

If there are additional response segments to the command, IMS/VS returns a CC status code to the program. and the program issues the get command call, or GCMD, to retrieve the remaining response segments. The GCMD call retrieves one segment at a time.

RESERVING AND RELEASING SEGMENTS

Since online programs don't have exclusive use of the data base as batch programs do, there may be times when you want to reserve a segment, to keep other programs from accessing that segment while you are using it. To some extent, IMS/VS does this for you through program isolation. Using the Q command code lets you reserve segments in a different way, then release them by using the dequeue call (DEQ).

Program isolation and the Q command code both reserve segments for your program's use. However, they work differently and are independent of each other. They are two separate things. To understand how and when to use the Q command code and DEQ call, you should also understand program isolation enqueues.

PROGRAM ISOLATION ENQUEUES

Program isolation reserves the data base record that you are processing for your program. When you move to a new data base record in the same data structure (as defined by the DB PCB), program isolation releases the data base record you were just processing and reserves the new record you're processing. The idea is that only one program at a time can access a data base record.

To understand why this is necessary, look at the example shown in Figure 88. It shows a root segment, A1, and two dependents, B1 and C1. Suppose your program, Program A, is working with data base record 1. While you are processing this record, you delete segment B1. If program isolation did not put a hold on data base record 1

for you, Program B could access segment B1 at the same time you were trying to delete it. Program B would read the segment before you deleted it, so Program B's results would be inaccurate.

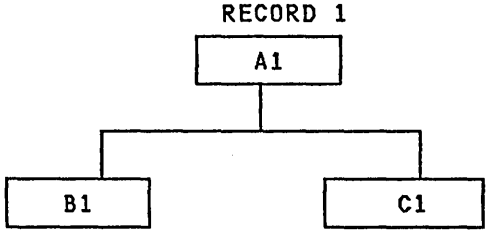

Figure 88. Program Isolation Example

If you were only reading segment B1, B1 would be available to other programs as soon as your program moved to another data base record. If, on the other hand, your program updated segment B1, IMS/VS would not release it until your program reached a sync point.

THE Q COMMAND CODE

You use the Q command code just as you do any of the other command codes described in "Using Command Codes." It is described in this chapter instead of with the other command codes is that you never need to use it in a batch program; it is an online tool.

For example, suppose a customer wants to place an order for items 1, 2, and 3 in Figure 89, but only if certain quantities of all three items are available, for example, 50 item 1s, 75 item 2s, and 100 item 3s. To place the order, you need to look at all three segments at once in order to determine if there are enough of each item on hand. Also, you don't want any other program to place an order while you're looking at these segments.

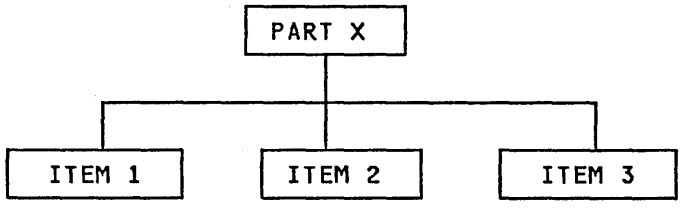

Figure 89. Q Command Code Example

To place the order, you use the Q command code when you retrieve items 1, 2, and 3. Since there's a chance that you will update these segments, it's a good idea to issue get hold calls for them:

```
GHU Part X
   Item 1 *QA
GHU Part X
   Item 2 *QB
GHU Part X
   Item 3 *QC
```

Once you have retrieved these segments, your program can examine each segment to find out if there are enough of each item on hand to place the order.

Suppose that there are 100 each of all three items on hand. You can then place the order and update the data base accordingly. To update the segment to reflect the order you're placing, you retrieve each segment again with a get hold call, and follow it immediately with a REPL call:

```
GHU  Item 1
REPL Item 1 with the value 50
GHU  Item 2
REPL Item 2 with the value 25
GHU  Item 3
REPL Item 3 with the value 0
```

If you update segments that you've reserved with the Q command code, you can't release them by using the DEQ call; IMS/VS holds them until your program reaches a sync point. (The Q command code does not hold segments from one cycle of a conversation to another.)

If, on the other hand, your program only reads segments, then you can release the segments with the DEQ call. IMS/VS will release these segments when your program reaches a sync point, but it's a good idea to issue the DEQ call for them so that they will be available to other programs as soon as possible. For each segment that you want to release, you place the letter for that segment's class (the letter you assigned to the segment when you issued the retrieval call with the Q for it) in an I/O area. You then issue a DEQ call for each segment, supplying the names of the I/O areas that contain letters for each call.

If you use the Q command code on a root segment, other programs will not be able to access any of the segments in that data base record. If you use the Q command code on a dependent segment, other programs can only read the segment using one of the retrieval calls without the hold.

When you use the Q command code in the SSA, you assign a class to the segment you're reserving. You designate the class by one of the letters A through J. Then, when you want to dequeue that segment, you include the letter that you assigned to the segment so that IMS/VS knows which segment you're dequeuing.

If your program updates the segment you've enqueued with the Q command code, IMS/VS won't release the segment until the next synchronization point (in the same way that IMS/VS holds all of the segments you update until a sync point).

IMS/VS returns a status code of GL if the first byte of the I/O area is not one of the letters between and including A and J.

BACKING OUT DATA BASE UPDATES: ROLB AND ROLL

When an MPP determines that some of its processing is invalid, there are two system service calls that make it possible for the program to remove the effects of its inaccurate processing. When you issue either of these calls, the following actions take place:

- IMS/VS backs out the data base updates that the program has made since the program's most recent sync point.
- IMS/VS cancels the output messages that the program has created since the program's most recent sync point.

The main differences between the two calls are that ROLB returns control to the application program after backing out updates and canceling output messages, and ROLL terminates the program with a user abend code of 0778; also, ROLB can return the last message segment to the program, but ROLL can't. Also, ROLB is valid only in single-mode programs, but ROLL is valid in both single-mode and multiple-mode programs. There are some other differences; Figure 90 summarizes these.

Actions Taken:	ROLB	ROLL
Back out data base updates since the last sync point	X	X
Cancel output messages created since the last sync point	1 X	1 X
Delete input messages from the queue since the last sync point		X
Return the last input message segment to the MPP to reprocess	2 X	
0778 abnormal termination, no dump		X
No abend; MPP continues processing	X	

Figure 90. Comparison of ROLB and ROLL

Notes:

1. ROLB will cancel output messages sent with an express PCB unless the program issued a PURG. For example, if the program issues the call sequence below, MSG1 would be sent to its destination, but MSG2 would be canceled:

   ```
   ISRT   EXPRESS PCB, MSG1
   PURG   EXPRESS PCB, MSG2
   ROLB   I/O PCB
   ```

2. If you supply the address of an I/O area as one of the call parameters.

USING ROLB

The advantage of using ROLB is that IMS/VS returns control to the program after executing ROLB, so the program can continue processing. If the program supplies the address of an I/O area as one of the ROLB parameters, the ROLB acts as a message retrieval call and returns the message segment that was being processed to the application program. This is true only if the program has issued a GU call to the message queue since the last sync point; if it hasn't, it wasn't processing a message when it issued the ROLB call.

If the program issues a GN to the message queue after issuing the ROLB, IMS/VS returns the next segment of the message that was being processed when ROLB was issued. If there are no more segments for that message, IMS/VS returns a QD status code.

If the program issues a GU to the message queue after the ROLB call, IMS/VS returns the first segment of the next message to the application program. If there aren't any more messages on the message queue for the program to process, IMS/VS returns a QC status code to the program.

If you include the I/O area parameter, but you haven't issued a successful GU call to the message queue since the last sync point, IMS/VS returns a QE status code to you.

If you don't include the address of an I/O area in the ROLB call, IMS/VS does the same things for you—except giving you the message

segment that was being processed. If the program has issued a successful GU in the sync interval, and then issues a GN, IMS/VS returns a QD status code. If the program issues a GU after the ROLB, IMS/VS returns the first segment of the next message, or a QC status code if there are no more messages for the program.

If you haven't issued a successful GU since the last sync point, and you don't include an I/O area parameter on the ROLB call, IMS/VS just backs out the data base updates and cancels the output messages that you've created since the last sync point.

The parameters for ROLB are the call function, the I/O PCB, and, optionally, the address of the I/O area.

USING ROLL

A ROLL call backs out the data base updates that the program has made since the last sync point, and it cancels the output messages that the program has created since the last sync point. It also deletes any messages for the program that have been sent to the message queue since the last sync point. IMS/VS then terminates the program with a user abend code 0778. This type of abnormal termination terminates the program without a storage dump.

You can use ROLB only in single-mode programs, so if your MPP is multiple mode you can use only ROLL.

USING ROLB AND ROLL IN CONVERSATIONS

When you issue a ROLB in a conversational program, IMS/VS backs out any messages that the application program inserted (except those sent with express PCBs). This means that if the program issues a ROLB, and then reaches a sync point without sending the required response to the originating terminal, IMS/VS will terminate the conversation and send the message, "DFS2171I NO RESPONSE, CONVERSATION TERMINATED" to the originating terminal.

If you issue ROLL during a conversation, IMS/VS backs out the updates and cancels output messages, but it also terminates the conversation.

CONSIDERATIONS FOR MESSAGE-DRIVEN FAST PATH PROGRAMS

A message-driven Fast Path program is similar to an MPP in that it retrieves messages and processes them and it can read and update MSDBs, DEDBs, and IMS/VS data bases. Also, a Fast Path program can issue only basic CHKP calls without the OS/VS checkpoint option.

Fast Path programs can send messages to:

- The logical terminal that sent the input message, by issuing an ISRT call that references the I/O PCB
- A different component of the physical terminal that sent the input message, by issuing an ISRT call that references a response alternate PCB
- A different physical terminal than the one that sent the input message, by issuing an ISRT call referencing an alternate PCB

A message-driven Fast Path program is different from an MPP in that:

- A message-driven Fast Path program must be specified as single mode. This means that each GU to the message queue is a sync point.
- It runs in wait-for-input mode. This means that the program is prescheduled and that it remains in main storage waiting for messages. The program must be prescheduled before anyone

enters an input transaction for it from a terminal. A message-driven Fast Path program doesn't terminate when there are no more messages for it to process.

- It processes only single-segment messages. A GU to the message queue retrieves the whole message. Fast Path programs can issue GN calls to the message queue; if they do, they do so to provide compatibility between IMS/VS programs and Fast Path programs.
- Fast Path programs doesn't receive input from remote IMS/VS systems through MSC. The same DB/DC system can have both Fast Path and MSC installed, but a program must not use both.
- Fast Path programs cannot be conversational programs. It is possible, however, to simulate conversational processing by using a dynamic MSDB as a SPA.
- A Fast Path program must terminate when it receives a QC status code.

RETRIEVING AND SENDING MESSAGES IN FAST PATH

The message calls that a message-driven Fast Path program can issue are GU, GN, ISRT, and CHNG.

A GU call to the message queue retrieves the next message queued for the application program. Fast Path messages are processed in response mode. This means that the originating terminal can't enter anymore input messages until the program has sent a response for the most recent input message from that terminal. If a message-driven program issues two GU calls without issuing an ISRT call between them, Fast Path generates a null message to the terminal as a response.

The GU may have a MOD name as one of the parameters. The MOD name is ignored for the GU call, but if you want to use the MOD name in an ISRT call, it's a good idea to include it in the GU call because you can then use the same parameter lists for GU and ISRT calls. If you do this, you only have to change the call function.

A message-driven Fast Path program issues GN calls for compatibility. Since all Fast Path messages are single segment, the program only has to issue a GU to retrieve the whole message. IMS/VS returns a blank status code when a message-driven Fast Path program issues a GN.

The program can use the CHNG call before an ISRT call to set the destination of a response alternate PCB that's been defined as modifiable. The destination remains set until the application program either issues a ROLB call or a GU call to the message queue. IMS/VS then resets the destination to blanks.

The CHNG call must not be used to set the destination to a response alternate PCB to a remote terminal or transaction code. If it is, IMS/VS returns an AI status code to the program.

The program uses the ISRT call to build and send output messages. Fast Path programs can send output messages using the I/O PCB or a response alternate PCB. When a message-driven Fast Path program wants to send a message to the originating terminal, the program places the message in an I/O area and issues an ISRT call that references the I/O PCB and the I/O area. Output messages, like input messages, are single-segment. If the program issues an ISRT call that references a response alternate PCB, the program must have issued a CHNG call before the ISRT call to set the destination for the response alternate PCB. Messages can't be greater than the maximum length that was defined at system definition.

You can issue only one ISRT call between two GU message calls. This applies to ISRT calls that reference any type of PCB. If an

ISRT call references a nonmodifiable response alternate PCB that represents a remote terminal or transaction, IMS/VS rejects the call and returns a QH status code.

If you use MFS, you can specify a MOD name in the ISRT call in the same way that you can in an ISRT call in an MPP.

USING ROLB IN FAST PATH

Suppose an application program tries to send an output message that is longer than the maximum length that's been defined for messages. IMS/VS returns an AG status code to the program, and the program could then discard its updates by issuing a ROLB. The program could then send an error message to the terminal.

ROLB is also helpful with FLD/VERIFY processing. If FLD/VERIFY fails when it's initially processed, the program can either:

- Issue a ROLB followed by an optional error message and a message GU to discard the current transaction.
- Issue a ROLB, reprocess the current transaction, and follow a different path through the program. If you specify an I/O area with a ROLB, IMS/VS returns a new copy of the message to this area.

Note: If a FLD/VERIFY fails when it's initially processed, Fast Path doesn't retry it at the sync point. If you don't issue a ROLB after a FLD/VERIFY that fails, the updates to the data base before the failed call are permanent.

USING CHKP IN FAST PATH

Fast Path programs can use only basic CHKP. When a Fast Path program reaches a sync point, IMS/VS writes out all data base buffers that the program has modified to the data base, and releases enqueued resources. Using a CHKP call in a message-driven Fast Path program is like issuing a GU call to the message queue.

Fast Path programs cannot use the OS/VS checkpoint option on the basic CHKP call. This is also true for IMS/VS programs that include any Fast Path PCBs.

CHAPTER 9. CODING A MESSAGE PROCESSING PROGRAM

This chapter tells you how to code the data communications portion of an MPP. "Chapter 7. Coding the DL/I Portion of a Program" tells how to code the data base calls and SSAs that you can use in an MPP; this chapter tells you how to code DC calls, message I/O areas, I/O PCB masks, and alternate PCB masks. There are four sections to this chapter:

- **Before You Code**

  This section gives you an overview of coding an MPP and describes the information that you need before you can code the program.

- **Coding the Program Logic**

  This section tells you how to code the data communications part of the program logic: the entry statement, the DC calls, and the system service calls available to MPPs.

- **Coding the Data Area**

  This section tells you how to code the data areas that an MPP uses: message I/O areas, I/O PCB masks, alternate PCB masks, and SPAs.

- **Coding a Message-Driven Fast Path Program**

  This section gives additional information for coding message-driven Fast Path programs.

BEFORE YOU CODE

This section introduces two things that you need to understand before you code your MPP: first, it explains the parts of the MPP and how they fit together in COBOL and PL/I; and second, it tells you what you need to know about your program and your data before you start to code your program.

PARTS OF AN MPP

At a minimum, the program logic in an MPP contains DC calls to retrieve and send messages and the processing logic; although most MPPs do some data base accessing as well, they don't have to. The data area of an MPP contains the I/O areas for the input and output messages that the program handles; the I/O PCB mask; the alternate PCB mask; and, if the program is a conversational program, the SPA.

One way to understand how these elements of the program fit together once they are coded is to look at a skeleton program that has these elements.

Figure 91 and Figure 92 show skeleton MPPs in COBOL and PL/I. These programs don't have all the processing logic that a typical MPP has. The purpose of providing these programs is to give you a basic understanding of MPP structure in COBOL and PL/I. Both programs do the same things; they each have three steps:

1. The program retrieves an input message segment from a terminal by issuing a GU call to the I/O PCB. This retrieves the first segment of the message. Unless you knew that this message contained only one segment, your program would then issue GN calls to the I/O PCB to retrieve the remaining segments of the message. IMS/VS places the input message

segment in the I/O area that you specify in the call. In the examples below, this is MSG-SEG-IO-AREA.

2. The program retrieves a segment from the data base by issuing a GU call to the DB PCB. This call specifies an SSA, SSA-NAME, to qualify the request. IMS/VS places the data base segment in the I/O area specified in the call. In this case, the I/O area is called DB-SEG-IO-AREA.

3. Next the program sends an output message to an alternate destination by issuing an ISRT call to the alternate PCB. Before issuing the ISRT call, the program has to build the output message segment in an I/O area, then the program specifies the I/O area in the ISRT call. The I/O area for this call is ALT-MSG-SEG-OUT.

COBOL MPP Structure

The program in Figure 91 is a skeleton MPP in COBOL that shows the main elements of an MPP. The numbers to the left of each part of the program refer to the notes that follow the program.

```
        ENVIRONMENT DIVISION.
             .
             .
             .
        DATA DIVISION.
        WORKING-STORAGE SECTION.
1            77 GU-CALL PICTURE XXXX VALUE 'GU  '.
             77 ISRT-CALL PICTURE XXXX VALUE 'ISRT'.
             77 CT PICTURE S9(5) COMPUTATIONAL VALUE +4.
             .
             01 SSA-NAME.
             .
2            01 MSG-SEG-IO-AREA.
             01 DB-SEG-IO-AREA.
             01 ALT-MSG-SEG-OUT.
        LINKAGE SECTION.
3            01 IO-PCB.
             01 ALT-PCB.
             01 DB-PCB.
        ------------------------------------------------------------
        PROCEDURE DIVISION.
4            ENTRY 'DLITCBL' USING IO-PCB, ALT-PCB, DB-PCB.
             .
5            CALL 'CBLTDLI' USING GU-CALL, IO-PCB,
                 MSG-SEG-IO-AREA.
             .
6            CALL 'CBLTDLI' USING GU-CALL, DB-PCB,
                 DB-SEG-IO-AREA, SSA-NAME.
             .
7            CALL 'CBLTDLI' USING ISRT-CALL, ALT-PCB,
                 ALT-MSG-SEG-OUT.
             .
8            GOBACK.
        ------------------------------------------------------------
9            COBOL LANGUAGE INTERFACE
```

Figure 91. COBOL MPP Skeleton

Notes:

1. To define each of the call functions that your program uses,

use a 77 or 01 level working storage statement. You assign the value to the call function in a picture clause defined as 4 alphameric characters.

2. Use a 01 level working storage statement for each I/O area that you'll use for message segments.

3. In the linkage section of the program, use a 01 level entry for each PCB that your program uses. You can list the PCBs in the order that you list them in the entry statement below, but this is not a requirement.

4. The entry statement must be the first executable COBOL statement in the procedure division. You list the PCBs that your program uses in the order they're defined in the program's PSB: first the I/O PCB, then any alternate PCBs, and finally the data base PCBs that your program uses.

5. The program issues a GU call to the I/O PCB to retrieve the first segment of an input message.

6. The program issues a GU call to the DB PCB to retrieve the segment that would be described in the SSA-NAME area.

7. The program sends an output message segment to an alternate destination by using an alternate PCB.

8. When there are no more messages for your MPP to process, you return control to IMS/VS by issuing the GOBACK statement.

9. You must link-edit your program to the language interface module, DFSLI000, after you've compiled your program.

PL/I MPP Structure

The program shown in Figure 92 is a skeleton MPP written in PL/I. The numbers to the left of the program refer to the notes that follow the program.

```
      /* ------------------------------------------------------- */
      /*                      ENTRY POINT                        */
      /* ------------------------------------------------------- */
1         UPDMAST:  PROCEDURE (IO_PTR, ALT_PTR, DB_PTR)
          OPTIONS (MAIN);

2     DCL   FUNC_GU     CHAR(4)   INIT('GU  ');
      DCL   FUNC_ISRT   CHAR(4)   INIT('ISRT');
      .
      DCL   SSA_NAME...;
      .
3     DCL   MSG_SEG_IO_AREA    CHAR(n);
      DCL   DB_SEG_IO_AREA     CHAR(n);
      DCL   ALT_MSG_SEG_OUT    CHAR(n);
      .
4     DCL   1 IO_PCB BASED (IO_PTR),...;
      DCL   1 ALT_PCB BASED (ALT_PTR),...;
      DCL   1 DB_PCB BASED (DB_PTR),...;
      .
5     DCL   THREE FIXED BINARY(31) INIT(3);
      DCL   FOUR FIXED BINARY(31)  INIT(4);
      DCL   PLITDLI ENTRY EXTERNAL;
      .
6     CALL PLITDLI (THREE, FUNC_GU, IO_PTR, MSG_SEG_IO_AREA);
      .
7     CALL PLITDLI (FOUR, FUNC_GU, DB_PTR, DB_SEG_IO_AREA,
           SSA_NAME);
      .
8     CALL PLITDLI (THREE, FUNC_ISRT, ALT_PTR, ALT_MSG_SEG_OUT);
      .
9     END UPDMAST;
-------------------------------------------------------------------
10    PL/I LANGUAGE INTERFACE
```

Figure 92. PL/I MPP Skeleton

Notes:

1. This is the standard entry point to a PL/I Optimizing Compiler MPP. This statement includes a **pointer** for each PCB that the MPP uses. The order in which you refer to the PCBs must coincide with the order in which they're listed in the PSB: first the I/O PCB, then any alternate PCBs that your program uses, and finally the data base PCBs that your program uses.

2. The program defines each call function that it uses in its data area. In PL/I you define the function codes as character strings and assign the appropriate values to them.

3. The most efficient way to define your I/O areas in PL/I is to define them as fixed-length character strings or through pointer variables.

4. To define your PCBs, use major structure declarations.

5. PL/I calls have a parameter that is not required in COBOL programs or assembler language programs. This is the parmcount, and it is always the first parameter. You define the values that your program will need for the parmcount in each of its calls. The parmcount gives the number of parameters that follow parmcount itself.

6. The program issues a GU call to the I/O PCB to retrieve the first message segment.

7. The program issues a GU call to the DB PCB to retrieve a data base segment. The function codes for these two calls are identical; the way that IMS/VS identifies them is by the PCB that each call refers to.

8. The program then sends an output message to an alternate destination by issuing an ISRT call to an alternate PCB.

9. When there are no more messages for the program to process, the program returns control to IMS/VS by issuing the END statement or the RETURN statement.

10. You must link-edit your program to the IMS/VS language interface module, DFSLI000, after you've compiled your program.

Assembler Language MPP Structure

The structure of an assembler language MPP is the same as it is for the DL/I assembler language structure shown in Figure 70. An assembler language MPP receives a PCB parameter list address in register 1 when it executes its entry statement. The first address in this list is a pointer to the I/O PCB; the addresses of any alternate PCBs that the program uses come after the I/O PCB address, and the addresses of the data base PCBs that the program uses follow. Bit 0 of the last address parameter is set to 1.

YOUR INPUT

In addition to the information you need about the data base processing your program does, there is some information you need about the message processing your program does. Before you start to code, make sure you're not missing any of this information. Also, be sure you are aware of the standards at your installation that affect your program.

Information You Need about Your MPP's Design

- The names of the logical terminals that your program will communicate with
- The transaction codes, if any, for the application programs that your application program will send messages to
- The DC call structure for your program
- The destination for each output message that you send
- The names of any alternate destinations that your program will send messages to

Information You Need about Input Messages

- The size and layout of the input messages your program will receive, if possible
- The format in which your program will receive the input messages
- The editing routine your program uses
- The range of valid data in input messages
- The type of data that input messages will contain
- The maximum and minimum length of input message segments
- The number of segments in a message

Information You Need about Output Messages

- The format in which IMS/VS expects to receive them from your application program

- The destination for the output message
- The maximum and minimum length of output message segments

Information You Need for a Conversational Program

- If you are going to pass control to another program, the tran code to use for that program
- The data that you should save in the SPA
- The maximum length of that data

CODING THE PROGRAM LOGIC

The program logic in an MPP contains an entry statement; the DC calls that you use to receive and send messages; the system service calls, if any, that the program issues; and the DL/I calls that the program issues to process the data base. You code the DL/I calls just as they are described in "Chapter 7. Coding the DL/I Portion of a Program."

CODING DC CALLS

The DC calls you use to retrieve message segments are GU and GN. You use an ISRT call to send messages, and you can use CHNG to set alternate destinations for messages. You can also use PURG in your program to send output messages to several destinations while you're processing one input message. For programs that issue commands, you use the CMD and GCMD to issue commands and retrieve command responses from IMS/VS.

The only parameter that all of the message calls have in common is the function code. You code the function code for a DC call just as you code it for DL/I calls. The function codes for the DC calls are:

GUbb Get unique

GNbb Get next

ISRT Insert

CHNG Change

PURG Purge

CMDb Command

GCMD Get command

CODING DC SYSTEM SERVICE CALLS

There are three system service calls available to an MPP that you might use in an MPP that you would not use in a batch program. These are ROLB, ROLL, and DEQ. You use ROLB and ROLL when you find you are processing an invalid transaction and you want to eliminate the data base updates and output messages you have produced since the last sync point. You use a DEQ call to release a segment that you have retrieved and reserved using the Q command code.

The parameters for these calls differ. The only parameter they have in common is the function code.

CHECKING STATUS CODES

An MPP has to check the status code after each call it issues. Most installations provide a standard status code routine for all the application programs at the installation. MPPs can use the sample status code error routine provided in "Appendix E. Sample Status Code Error Routine (DFSOAER)." To use this routine, your program issues a call similar to a DL/I call, giving the name of the PCB to use for this call and specifying the options you want for the call.

CODING THE DATA AREA

The data area in an MPP contains the I/O areas your program uses for input and output messages; the I/O PCB mask; the alternate PCB masks, if any, that your program uses; and, for conversational programs, the SPA.

CODING I/O AREAS

The only difference between coding message I/O areas in an MPP and coding the data base I/O areas is the format of the message area.

Input and output messages have two prefix fields: the 2-byte length field, and the 2-byte Z field. The format of the text of the message depends on your program's data and the message formats that have been chosen for your program.

CODING I/O PCB MASKS

To code the I/O PCB mask, you must be familiar with the eight fields of the I/O PCB mask shown in Figure 71. The order in which you define the PCB masks in your program doesn't matter; you just have to refer to them in the correct order in the entry statement.

CODING ALTERNATE PCB MASKS

Modifiable, express, and response alternate PCB masks are coded in the same way. An alternate PCB mask has only three fields:

- The 8-byte logical terminal name
- The 2-byte field reserved for IMS/VS
- The 2-byte status code field

CODING SPAS

A SPA contains 4 fields:

- The 2-byte length field.
- The 4-byte field that's reserved for IMS/VS
- The 8-byte tran code
- The work area where you store the conversation data. The length of this field is defined at system definition.

CODING A MESSAGE-DRIVEN FAST PATH PROGRAM

Because Fast Path messages can only be single-segment messages, the only call you use to retrieve messages is GU. Fast Path programs can't use the PURG call. Other than that, coding the DC calls that you use in message-driven programs is no different from coding them in an MPP.

Some of the system service calls that are available to an MPP can't be used in a Fast Path program. The system service calls that you can use in a Fast Path program are:

- Basic CHKP
- ROLB

You code these calls just as you code them in an IMS/VS program.

Fast Path has one additional system service call, called the SYNC call. This call causes a synchronization point to occur. The parameters of the call are the call function, SYNC, and the I/O PCB.

Fast Path programs cannot be conversational programs, so you would never code a SPA in a Fast Path program.

CHAPTER 10. STRUCTURING AND CODING A BATCH MESSAGE PROGRAM

A batch message program (BMP) is a cross between an MPP and a batch program. It has some characteristics of message processing and some of batch processing.

This chapter explains how you structure and code a BMP. The aspects of batch message processing that this chapter explains are:

- **Processing Online Data Bases**

  When you run a batch program, that program is the only program using the data base at that time. This is because DL/I has no way to ensure data integrity and security if several programs were to access the same segment and try to update it. This section tells how processing online data bases is different from processing DL/I data bases.

- **Designing Transaction-Oriented BMPs**

  This section covers considerations in structuring a transaction-oriented BMP.

- **Designing Batch-Oriented BMPs**

  This section covers considerations in structuring a batch-oriented BMP.

A BMP is similar to an MPP in that a BMP:

- Accesses online data bases, so it can use the Q command code, the DEQ call, and the ROLB and ROLL calls.
- Can receive input from and send output to the IMS/VS message queues to do this, a BMP uses the same DC calls that you use in an MPP.
- Runs online.

A BMP is similar to a batch program in that a BMP:

- Can access OS/VS files
- Should issue checkpoint calls throughout the program
- Is started up with JCL

There are two types of BMPs: batch oriented and transaction oriented. The major differences between transaction-oriented BMPs and MPPs are:

- Transaction-oriented BMPs are started by JCL, while MPPs are scheduled by the control region.
- Transaction-oriented BMPs don't have to access the message queues, while MPPs must access the message queues for both input and output.

A batch-oriented BMP is a batch program that is executed under the control and supervision of the IMS/VS online system.

PROCESSING ONLINE DATA BASES

When you run a BMP, there are other BMPs and MPPs that are accessing the data base at the same time. The data bases that BMPs and MPPs access are online data bases; these data bases are supervised by the IMS/VS control region, not DL/I. The IMS/VS

control region has ways to make sure that, if a program updates a segment, other programs will not be able to access the updated segment until the program is sure that the new segment is valid. In other words, the updates that an MPP or BMP makes to the data base do not become effective immediately; IMS/VS holds them until the MPP or BMP indicates to IMS/VS that the results of its processing thus far are valid, even if the program terminates abnormally. The program indicates this when it reaches a sync point. Where sync points occur in your BMP depends on whether your BMP is transaction or batch oriented, and, if transaction oriented, whether it is multiple or single mode.

TOOLS AVAILABLE TO BMPS

To read and update online data bases you use the same calls that you use to read and update DL/I data bases. There is no difference in the way you use them or code them. There are, however, some additional tools available to you when you process online data bases. "Chapter 8. Structuring a Message Processing Program"explains how you use these tools.

- **Q Command Code**

  The Q command code reserves the segment or segments you specify in the SSA for your program's use. Other programs can read these segments, but can't update them. As part of program isolation, IMS/VS puts a hold on the data base record that your program is currently processing. Although IMS/VS releases the data base record when you move your current position to a new data base record within the same DB PCB data structure, IMS/VS doesn't release the segment(s) you have reserved with the Q command code when it releases the data base record that it is holding. If you haven't updated the segment, IMS/VS releases it when you issue a DEQ call or when you reach a sync point. If you have updated it, IMS/VS won't release the segment until your program reaches a sync point.

- **DEQ**

  After you have reserved a segment or group of segments by using the Q command code, you release the same segment(s) by issuing the DEQ call. When you reserve the segments, you must specify a 1-character code, within the range of A to J, for each segment or group of segments you reserve. You can reserve one segment in a call, or you can reserve several, by specifying one letter for a group of segments. When you want to release a segment, you issue the DEQ call and specify the code of the segment or segments that you want to release. "The Q Command Code" explains how you use the Q command code and DEQ. If you have updated the segment, IMS/VS does not release it until you reach a sync point.

- **ROLB and ROLL**

  A program can issue a ROLB call or a ROLL call when it realizes that some or all of its processing has been incorrect. When you issue ROLB or ROLL, IMS/VS backs out the data base updates that your program has made since the last sync point. When you issue a ROLB or ROLL in a transaction-oriented BMP, IMS/VS also cancels any output messages that your program has created since the last sync point.

SYNC POINTS

When a transaction-oriented BMP issues an ISRT call to send an output message, IMS/VS holds the output message at a temporary destination until the program reaches a sync point. IMS/VS also holds the data base segments that a BMP has updated until the next sync point. Transaction-oriented BMPs are specified as either single mode or multiple mode in the TRANSACT statement of the

APPLCTN macro for the BMP. Where sync points occur in a transaction-oriented BMP depends on what has been specified on this statement for your program. "Checkpoints in MPPs and Transaction-Oriented BMPs" describes where sync points occur in transaction-oriented BMPs. In a batch-oriented BMP, the only sync points are the checkpoint calls in your program.

BMPs can issue either kind of checkpoint call (symbolic or basic). "Taking Checkpoints" describes the advantages of the symbolic call. There is a status code that IMS/VS can return to a BMP following a checkpoint call that it doesn't return to other programs. This is the status code XD. If your program receives this status code, it should terminate immediately. It means that IMS/VS is undergoing a checkpoint freeze, and that, if your program issues any more DL/I calls, IMS/VS will abnormally terminate it.

DESIGNING TRANSACTION-ORIENTED BMPS

A typical use of a transaction-oriented BMP is to use it to simulate direct update online. This means that an MPP, instead of updating the data base as it processes its transactions, sends the updates to a message queue for the BMP to process later on. You can then run the BMP as you need to, depending on the quantity and type of updates. This improves the response time for the MPP, and it keeps the data in the data base fairly current.

PROCESSING MESSAGES

To process messages, a transaction-oriented BMP uses the same message calls that an MPP uses. These are GU and GN to retrieve input messages from the queue. Both transaction-oriented BMPs and batch-oriented BMPs can send output to the message queue by issuing ISRT calls. BMPs can also use CHNG and PURG to set alternate destinations using an alternate PCB for output messages. Using these calls in a BMP is no different from using them in an MPP. "Chapter 8. Structuring a Message Processing Program"explains how to use these calls.

If a BMP processes transactions that have been defined as "wait-for-input," IMS/VS allows the BMP to remain in main storage after it has processed the available input messages. IMS/VS returns the QC status code to the program if the limit count is reached, if the master terminal operator enters a command to stop any further processing, or if IMS/VS is terminated with a checkpoint shutdown. Wait-for-input is specified on the WFI parameter on the TRANSACT macro.

SYNC POINTS AND CHECKPOINTS IN TRANSACTION-ORIENTED BMPS

When you issue either kind of checkpoint call in a transaction-oriented BMP, IMS/VS returns the first segment of the next message, if there is one, to the I/O area that you name as one of the parameters of the call. In other words, a checkpoint call acts like a GU to the message queue. The I/O area that you name in the checkpoint call must be large enough to hold this message segment. When you check the status codes after a checkpoint call, you should check for status codes that apply to a GU message call, as well as for those that apply to a checkpoint call.

Single-Mode BMPs

If your program is single mode, each GU message call that your program issues is s sync point. You use checkpoints in single-mode BMPs to make restart easier. If your program is short enough to restart from the beginning, you don't have to use checkpoints at all. If you do want to be able to restart it from some place other

than the beginning of the program, however, use checkpoint calls in the program.

If you use checkpoint calls in a single-mode transaction-oriented BMP, don't mix checkpoint calls and GU message calls; use all checkpoint calls and GN calls to access the message queue. The reason for this is that, although both calls establish sync points, you can't restart a program from a sync point; you can restart a program only from a checkpoint. When the program terminates abnormally, IMS/VS backs it out to the most recent sync point, then restarts it from the most recent checkpoint. If you issue checkpoints and GU message calls in the program, you have no way to be sure that the most recent sync point will be the most recent checkpoint.

Multiple-Mode BMPs

In a multiple-mode BMP, the only sync points in the program (other than program termination) are checkpoint calls. Since GU message calls are not sync points, you can use both GU message calls and checkpoint calls in the program. The considerations for where and how often you issue checkpoints in a multiple mode program are the same as those for a batch-oriented BMP that are explained below.

DESIGNING BATCH-ORIENTED BMPS

A batch-oriented BMP is similar to a batch program. They are different in that a BMP can use the additional tools described above, and it should issue checkpoints even more frequently than a batch program should. Also, a batch-oriented BMP can send output to the message queue. It does this by issuing one ISRT call for each message segment.

In a batch program, checkpoints are important for restart and recovery time. But in a batch-oriented BMP there are some additional reasons for issuing checkpoints. They are:

- **Enqueue Lockout**

  When a BMP reads a segment, IMS/VS keeps other programs from accessing that segment. IMS/VS does this by holding the data base record that the segment belongs to until the BMP moves to another data base record within the data structure defined by the same DB PCB. If the program updates the segment, IMS/VS doesn't release the segment until the BMP reaches a sync point. In a batch-oriented BMP, checkpoint calls are the only sync points (except for program termination). If a BMP accesses a large number of data base segments between checkpoint calls, the program can tie up large portions of the data base and cause long waits for MPPs and other BMPs trying to access the same segments. When a BMP issues a checkpoint call, IMS/VS releases all segment occurrences that the program has enqueued in the interval since the last checkpoint call, and makes them available to other online programs.

- **Enqueue Space**

  Another reason that a BMP needs to issue checkpoints frequently is that IMS/VS can run out of space for the segments enqueued by the BMP. If a BMP has enqueued too many data base segments, the amount of storage required for the enqueued records can exceed the amount of storage available for them. If this happens, IMS/VS will terminate a program abnormally with a code of 0775. Before you could reexecute the program, you would have to increase the checkpoint frequency for the program.

- **Dynamic Log Space**

  In addition to freeing enqueued data base segments, a checkpoint call also frees dynamic log records. As a BMP updates segments, IMS/VS records the before images of the segments on the dynamic log. This space is not freed until the BMP reaches a sync point. If the program doesn't issue checkpoints frequently enough, the dynamic log can reach a wraparound point. You can then either issue checkpoints more frequently or increase the amount of storage available for the dynamic log.

CHAPTER 11. TESTING AN APPLICATION PROGRAM

This chapter tells you what is involved in testing an application program as a unit and gives you some guidelines on how to do it. This stage of testing is called program unit test. The purpose of program unit test is to test each application program as a single unit to see that the program correctly handles its input data, processing, and output data.

Once all the application programs that are part of the same system have been tested, the entire system is tested. This is called application system test. The application system test checks the way in which data is passed within the system. Its purpose is to make sure that the system works as a whole, and to verify that the system will work under stress. This is usually done under the supervision of the data communications administrator and the system programmer. This chapter covers only unit test.

The amount and type of testing you do depend on the individual program. There are no hard and fast rules, but there are some guidelines that can be of help. This chapter contains the following sections:

- **What You Need to Test Your Program**

  This section tells you what you need before you can test your program.

- **Testing DL/I Call Sequences**

  Before you test the program as a whole, you can use the DL/I test program, DFSDDLT0, to test the sequence of DL/I calls that your program uses. This section tells you how to do this.

- **Using BTS II to Test Your Program**

  This section gives you an overview of how to use Batch Terminal Simulator II, or BTS II. You can use BTS II to test conversational programs, MPPs, BMPs, and batch programs. It cannot test Fast Path programs. BTS II is valuable to application programs that communicate with terminals, because it simulates the terminal's interactions with your program. This makes it possible for you to test online programs offline.

- **What to Do When Your Program Terminates Abnormally**

  If your program doesn't run, or if it gives inaccurate results, you have to find the problem before you can fix it. This section gives some suggestions concerning some of the actions that you, as the application programmer, can take in determining what caused the problem. The suggestions in this section are limited to fixing problems in your program.

- **Calls You Use for Monitoring and Debugging**

  This section explains some system service calls that can be helpful in debugging. They are: STAT, LOG, and GSCD.

WHAT YOU NEED TO TEST A PROGRAM

When you are ready to test your program, be sure you're aware of the test procedures at your installation before you start. Some of the things you need to test your program are:

- Test JCL

- A test data base. When you're testing a program, you don't execute it against a production data base because the program, if faulty, might damage valid data.
- Test input data. The input data that you use doesn't have to be current, but it should be valid data. You can't be sure that your output data is valid unless you use valid input data.

The purpose of testing the program is to make sure that the program can correctly handle all of the situations that it can encounter.

To thoroughly test the program, try to test as many of the paths that the program can take as possible. For example:

- Try to test each path in the program. To do this, use input data that forces the program to execute each of its branches.
- Be sure that your program tests its error routines. Again, use input data that will force the program to test as many error conditions as possible.
- Test the editing routines your program uses. Give the program as many different data combinations as possible to make sure it correctly edits its input data.

TESTING DL/I CALL SEQUENCES

DFSDDLT0 is an IMS/VS application program that executes the DL/I calls you specify against a test data base. An advantage of using DFSDDLT0 is that you can test the DL/I calls in your program without executing the program as a whole. Testing the DL/I call sequence before you test the program makes debugging easier, because, by the time you test the program, you know that the DL/I calls are accurate. When you test the program, if it doesn't execute correctly, you know that the DL/I calls aren't part of the problem if you've already tested them using DFSDDLT0.

For each DL/I call that you want to test, you give DFSDDLT0 the call and any SSAs that you're using with the call. DFSDDLT0 then executes and gives you the results of the call. After each call, DFSDDLT0 shows you the contents of the DB PCB mask and the I/O area. This means that for each call, DFSDDLT0 checks the access path you've defined for the segment, and the effect of the call. DFSDDLT0 is helpful in debugging, because it can display DL/I control blocks.

To use DFSDDLT0, you need to know what the results of each call should be when the call is executed against the test data base you're using. To indicate to DFSDDLT0 the call you want executed, you use four types of control statements.

- **Status statements** establish print options for DFSDDLT0's output and select the DB PCB to use for the calls you specify.
- **Comments statements** let you choose whether or not you want to supply comments.
- **Call statements** indicate to DFSDDLT0 the call you want to execute, any SSAs you want used with the call, and how many times you want the call executed.
- **Compare statements** tell DFSDDLT0 that you want it to compare its results after executing the call with the results you supply.

In addition to testing call sequences to see if they work, you can also use DFSDDLT0 to check the performance of call sequences. There are two ways you can do this.

If you use DL/I call trace, you can use DFSDDLT0 to compare the results of two call sequences that access the same segment. You can determine which sequence of the two is more efficient by analyzing the results of the trace.

You can also use the DFSDDLT0 timings to check the performance of different call sequences. DFSDDLT0 records two timings: task time and real time.

- **Task time** records the amount of time the DL/I call takes to execute. This time is printed in microseconds. Task time measures only the DL/I task. It does not include the OS/VS supervisor state time or the I/O processing time.
- **Real time** records the time of day at call completion from the store clock.

You turn both timings on and off in the status statement. "Appendix F. Using the DL/I Test Program (DFSDDLT0)" explains the formats and meanings of the DFSDDLT0 control statements.

USING BTS II TO TEST YOUR PROGRAM

BTS II is a valuable tool for testing all types of IMS/VS application programs, except Fast Path programs. BTS can test call sequences, and the documentation it produces is helpful in debugging.

When you want to test an MPP or BMP that communicates with a terminal, using BTS II is a good idea, because it allows you to test the program without running the program online. BTS II is a program that simulates what IMS/VS data communications does so that you can run an online program offline. BTS II simulates all DC calls; you can specify that it print the contents of the I/O PCB, any alternate PCBs, I/O areas, and SPAs.

BTS II gives you information about each transaction as it goes through the IMS/VS system. You can use it to test your program logic, the program's IMS/VS interfaces, and its data base calls as well as the program's interactions with the terminal. For information on how to use BTS II, refer to the BTS II Program Description/Operations Manual.

WHAT TO DO WHEN YOUR PROGRAM TERMINATES ABNORMALLY

If your program terminates abnormally, there are some actions that you can take that can make finding and fixing the problem a little easier. First, you can record as much information as possible about the circumstances under which the program terminated abnormally; and second, you can check for certain initialization and execution errors.

WHEN YOU FIND YOU HAVE A PROBLEM

Many installations have guidelines concerning what you should do if your program terminates abnormally. The suggestions given here are some common ones.

- Document the error situation to help in investigating and correcting it. Some of the information that can be helpful is:
  - The program's PSB name
  - The transaction code that the program was processing
  - The text of the input message being processed
  - The name of the originating logical terminal
  - The call identifier

- The call function
- The contents of the PCB that was referenced in the call that was executing
- The contents of the I/O area when the problem occurred
- If a data base call was executing, the SSAs, if any, that the call used
- The date and time of day

- When your program encounters an error, it can pass all of the required error information to a standard error routine. Online programs might want to send a message to the originating logical terminal to inform the person at the terminal that there's been an error. The program can get the logical terminal name from the I/O PCB, place it in an express PCB, and issue one or more ISRT calls to send the message.
- An online program might also want to send a message to the master terminal operator giving information about the program's termination. To do this, the program places the logical terminal name of the master terminal in an express PCB and issues one or more ISRT calls.
- You might also want to send a message to a printer so that you'll have a hardcopy record of the error.
- You can send a message to the system log by issuing a LOG call. "Writing Information to the System Log: LOG" describes this call.
- Some installations run a BMP at the end of the day to list all the error that have occurred during the day. If your installation does this, you can send a message using an express PCB that has its destination set for that BMP.

FINDING THE PROBLEM

If your program doesn't run correctly when you are testing it or when it's executing, you need to isolate the problem. The problem might be anything from a programming error (for example, an error in the way you coded one of your calls), to a problem in the IMS/VS system. This section gives some guidelines about the steps that you, as the application programmer, can take when your program fails to run, terminates abnormally, or gives incorrect results.

Initialization Errors

Before your program receives control, IMS/VS must have correctly loaded and initialized the PSB and DBDs used by your application program. Often, when the problem is in this area, you need a system programmer or DBA (or the equivalent specialist at your installation), to fix the problem. One thing you can do is to find out if there have been any recent changes to the DBDs, PSB, and the control blocks that they generate.

Execution Errors

If you don't have any initialization errors, check the following in your program:

1. The output from the compiler—make sure that all error messages have been resolved.
2. The output from the linkage editor:
   - Are all external references resolved?

- Have all necessary modules been included?
- Was the language interface module correctly auto-linked?
- Is the correct entry point specified?

3. Your JCL:
   - Is the information that described the files that contain the data bases correct? If not, check with your DBA.
   - Have you included the DL/I parameter statement in the correct format?
   - Have you included the region size parameter in the EXEC statement? Does it specify a region or partition large enough for the storage required for IMS/VS and your program?
4. Your program:
   - Have you declared the fields in the PCB masks correctly?
   - If your program is an assembler language program, have you saved and restored registers correctly? Does register 1 point to a fullword parameter list before issuing any DL/I calls?
   - For COBOL and PL/I, are the literals you're using for arguments in DL/I calls producing the results you expect? For example, in Pl/I, is the parameter count being generated as a halfword instead of a fullword, or is the function code producing the required 4-byte field?
   - Use the PCB as much as possible to determine what in your program is producing incorrect results.

CALLS YOU USE FOR MONITORING AND DEBUGGING

There are three IMS/VS system service calls that can be helpful when you are debugging your program:

- The statistics call, or STAT, retrieves system statistics.
- The log call, or LOG, makes it possible for the application program to write a record on the IMS/VS system log.
- The GSCD call retrieves the addresses of the system contents directory and the partition specification table.

RETRIEVING IMS/VS SYSTEM STATISTICS: STAT

The statistics system service call, or STAT, can be helpful in debugging because it retrieves IMS/VS system statistics. It is also helpful in monitoring and tuning for performance. STAT can retrieve ISAM/OSAM data base buffer pool statistics and VSAM data base buffer subpool statistics.

When you issue a STAT call, you have three choices about the format that you want the statistics to be in. IMS/VS will give them to you unformatted, in a summary report, or formatted. "STAT Call Formats" explains how you specify which format you want.

If you want to retrieve ISAM/OSAM statistics, you only have to issue one STAT call. If you want to retrieve VSAM statistics, each STAT call you issue retrieves the statistics for a subpool. The first call retrieves the statistics for the smallest subpool. When IMS/VS has returned the statistics for all of the subpools, IMS/VS gives you a GA status code. If you issue the call again, IMS/VS returns the total statistics for all of the VSAM subpools. If you issue a STAT call for VSAM or ISAM/OSAM statistics and

you're not using any of the type of buffer pools that you specified in the call, IMS/VS returns a GE status code to the program.

WRITING INFORMATION TO THE SYSTEM LOG: LOG

An application program can write a record to the system log by issuing the LOG call. Fast Path programs cannot use the LOG call. When you issue the call, you give the address of the I/O area that contains the record you want IMS/VS to record on the system log. This record must be in the format shown in Figure 93.

LL	ZZ	C	Text
2	2	1	variable

Figure 93. Log Record Format

LL A 2-byte field that contains the total length of the record.

ZZ This 2-byte field must contain binary zeros.

C This 1-byte field contains a user log code. This code must be greater than or equal to X'A0'.

Text This is the text you want IMS/VS to write to the system log.

There are two restrictions on the length of the log record:

- The total length (LL + ZZ + C + Text) cannot be greater than the logical record length (LRECL) for the system log data set, minus 4 bytes.
- The total length can't be greater than the I/O area length specified in the IOASIZE keyword of the PSBGEN statement in the PSB.

If the call is successful, IMS/VS returns a blank status code. These are error status codes for LOG:

AD A Fast Path program tried to issue a LOG call.

AT The record length in the LL field is too long.

GL The log code is not a valid user log code.

RETRIEVING SYSTEM ADDRESSES: GSCD

The GSCD call ("get system contents directory") retrieves the addresses of the IMS/VS system contents directory (SCD) and the partition specification table (PST). If your program references these control blocks, you should use the dsects provided by the macros in the macro library for the IMS/VS system. The macro for the SCD dsect is ISCD SCDBASE=0. The macro for the PST dsect is IDLI PSTBASE=0. You must reference a PCB as one of the parameters in the call; this PCB can be any valid PCB.

The GSCD call returns the addresses in an 8-byte I/O area that you reference in the call. IMS/VS returns the SCD address in the first 4 bytes of the area, and the PST address in the second 4 bytes of the area.

Note: GSCD functions normally in a message processing or batch processing region using OS/VS2 or OS/VS1 with fetch protect. Use GSCD only in batch programs. The reason that you should not use it

in BMPs and MPPs it that the operating system doesn't allow a program in one region to access data in another region. Since the addresses are from the control region, and MPPs and BMPs run in separate regions, these addresses cannot be used in a message processing or batch message processing region. If they are, an OC4 system abend will occur.

CHAPTER 12. DOCUMENTING AN APPLICATION PROGRAM

This chapter does not give any rules about the form you use to document your program, or about the type and amount of information you include, but it does give some guidelines about the type of program documentation that is valuable. Many installations establish standards in this area; make sure you are aware of the standards at your installation.

When you document an application program, you record information about the program for two reasons:

- **Documentation for Other Programmers**

  One kind of information you record is information about structuring and coding the program. You record this information for other programmers who may have to maintain your program in the future.

- **Documentation for Users**

  The other kind of information you should record is information for the people who use your application program. You need to provide these people with the information they need to use your program.

DOCUMENTATION FOR OTHER PROGRAMMERS

Documenting a program is not something you should wait until the end of the project to do; your documentation will be much more complete, and more useful to others, if you record information about the program as you structure and code it. Include any information you think might be useful to someone else who has to work with your program.

The reason you record this information is so that people who maintain your program know something about why you chose certain call structures, SSAs, and command codes. For example, if the DBA was considering reorganizing the data base in some way, information about why your program accesses the data the way it does would be helpful.

A good place to record information about your program is in a data dictionary. It is then available to all of those who might need it, and it is easy to maintain.

Some of the information you should include for other programmers is:

- Flowcharts and pseudocode for the program.
- Comments about the program from code inspections.
- A written description of the program flow.
- Information about why you chose the call sequence you did, such as:
  - Did you test the call sequence using DFSDDLT0?
  - In cases where more than one combination of calls would have had the same results, why did you choose the sequence you did?
  - What was the other sequence? Did you test it using DFSDDLT0?

- Any problems you encountered in structuring or coding the program.
- Any problems you had when you tested the program.
- Warnings about what should not be changed in the program.

This information all relates to structuring and coding the program. In addition, you should include the information described in "Documentation for Users" with the documentation for programmers.

Again, the amount of information you include and the form in which you document it depend on you and your installation. These are provided as suggestions.

DOCUMENTATION FOR USERS

All of the information listed above relates to the design of the program. In addition to this, you should record information about how you use the program. The amount of information that users need, and how much of it you should supply, depend on who the users of the program are and what type of program it is.

At a minimum, you should include this information for users of your program:

- What the user needs to use the program
  - For online programs, is there a password?
  - For batch programs, what is the required JCL?
- The input users need to supply for the program
  - For example, for an MPP, what is the MOD name that the user enters to initially format the screen?
  - For a batch program, is the input in the form of cards or tape?
- The content and form of the program's output
  - If it's a report, show the format or include a sample listing.
  - For an online application program, show what the screen will look like.
- For online programs, if the user has to make decisions, tell what is involved in each decision. Give choices and defaults.

If the people who will be using your program are unfamiliar with terminals, they will need some kind of user's guide as well. For example, this guide should give explicit instructions on how to use the terminal and what they can expect from the program. Although you may not be responsible for providing this kind of information, you should provide any information that is unique to your application, regarding terminal use, to the person responsible for this information.

PART 3. FOR YOUR REFERENCE

This part of the book contains reference information about the DL/I and DC tools that have been explained in Part 2 of this book. This part does not contain explanations of the tools; it gives you rules for coding each of the tools, and examples of how they are coded in COBOL, PL/I, and assembler language. This part contains reference information about:

- IMS/VS Entry and Return Conventions
- DL/I Calls
- DB PCB Masks
- I/O Areas
- SSAs
- DC Calls
- System Service Calls
- Fast Path Reference
- GSAM Reference
- Status Codes

IMS/VS ENTRY AND RETURN CONVENTIONS

The formats for entry statments in COBOL, PL/I, and assembler language are as follows. Your entry statement must refer to the I/O PCB first; then any alternate PCBs it uses in the order they're defined in the PSB; and lastly the DB PCBs it uses, in the order they've been defined in the PSB.

COBOL

This statement must be the first statement in the procedure division:

```
ENTRY 'DLITCBL' USING pcb-name-1 [,...,pcb-name-n].
.
.
.
GOBACK.
```

PL/I

This statement must be the first statement in the program. When IMS/VS passes control to your program, it passes the addresses, in the form of pointers, of each of the PCBs your program uses. When you code the entry statement, make sure you code the parameters of this statement as pointers to the PCBs, and not the PCB names.

```
DLITPLI: PROCEDURE (pcb1_ptr [,...,pcbn_ptr]) OPTIONS (MAIN);
RETURN;
```

ASSEMBLER LANGUAGE

You can use any name for the entry point to an assembler language DL/I program. When IMS/VS passes control to the application program, register 1 contains the address of a variable-length fullword parameter list. Each word in the list contains the address of a PCB. IMS/VS sets the high-order byte of the last fullword in the list to X'80' to indicate the end of the list. After saving DL/I registers and the entry point address, you should save the PCB addresses.

```
 PROGRAM CSECT Entry point
              USING   *,BASE          Addressability
              SAVE    14,12           Save DL/I registers
              LR      Base,15         Entry point address
              LR      R2,R1           Save PCB list
*
*  Reg 1 contains pointer to PCB address list
*  Reg 13 contains pointer to DL/I save area
*  Reg 14 contains DL/I return address
*  Reg 15 contains program entry point
*  BASE contains the address of the program
*
.
.
.
              RETURN  (14,12)         Reload DL/I registers
*                                     and return
.
.
.
              END
```

DL/I CALLS

This section shows you DL/I call formats in COBOL, PL/I, and assembler language, and defines the call parameters.

DL/I CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, db pcb, i/o area
     [,ssa1,...,ssa15]
```

PL/I

```
CALL PLITDLI (parmcount, function,
     db pcb, i/o area [,ssa1,...,ssa15]);
```

ASSEMBLER LANGUAGE

```
column                                                    column
10                                                          72
CALL ASMTDLI,(function,(db pcb reg),i/o area                 X
     [,ssa1,...,ssa15]),VL
```

DL/I CALL PARAMETERS

All of the DL/I calls use the same parameters. Three of the parameters shown above are required in all DL/I calls: function, db pcb, and i/o area. In addition, an ISRT call requires at least one unqualified SSA. SSAs are optional for the other calls.

parmcount
This is required only in PL/I. Assembler language programs must use either parmcount or VL. This parameter is the address of a 4-byte field that contains the number of parameters that follow parmcount in the list. This value does not include parmcount itself. Parmcount is optional for COBOL.

function
This parameter gives the address of a 4-byte field that contains the DL/I function code for the type of call you want. The function code must be 4 bytes long. If the mnemonic for the call is less than 4 bytes, for example, GN, it is padded on the right with blanks to fill in the extra 2 bytes, GNbb. The function codes for each of the DL/I calls are:

GUbb Get unique

GHUb Get hold unique

GNbb Get next

GHNb Get hold next within parent

GNPb Get next within parent

GHNP Get hold next within parent

DLET Delete

REPL Replace

ISRT Insert

db pcb or db pcb reg
This parameter gives the name of the PCB that DL/I will reference for this call. In COBOL, give the name of the DB PCB that you've defined in the linkage section of the program. In PL/I programs, you can pass either the PCB name or the PCB pointer. In assembler language programs, pass the register that contains the address of the PCB for this call.

i/o area
This parameter identifies the area in your program with which your program will communicate with DL/I. When you issue one of the get calls successfully, DL/I returns the requested segment to this area. When you want to replace an existing segment in the data base with a new segment, you place the new segment in the I/O area before issuing the REPL call. After you issue a DLET call successfully, DL/I returns the segment it has just deleted to this area. When you want to add a new segment to the data base, you place the new segment in this area before issuing the ISRT call. This area must be large enough to hold the longest segment that DL/I returns to this area. For example, if none of the segments your program retrieves or updates is longer than 48 bytes, your I/O area has to be 48 bytes. If your program issues any path calls, the I/O area must be long enough to hold the longest concatenated segment following a path call. The segment data that this area will contain is always left-justified; the name of the I/O area points to the first byte of this area.

ssa1,...,ssa15
These parameters give DL/I the addresses of the SSAs, if any, to be used in this call. The ISRT call is the only call that requires any SSAs; this parameter is optional for the rest of the DL/I calls. The names you supply in the DL/I call point to data areas in your program in which you have defined the SSAs for the call.

DB PCB MASKS

A DB PCB mask must contain the fields shown in Figure 94. The fields in your DB PCB mask must be defined in the same order, with the same length, as the fields shown here. When you code the DB PCB mask, you also give it a name, but the name is not part of the mask. You use the name (or the pointer, for PL/I) when you reference each of the PCBs your program processes in the program entry statement. A GSAM DB PCB mask is slightly different from a DL/I DB PCB mask. "GSAM DB PCB Masks" gives the format for GSAM DB PCB masks.

1. Data Base Name 8 bytes
2. Segment Level Number 2 bytes
3. Status Code 2 bytes
4. Processing Options 4 bytes
5. Reserved for DL/I 4 bytes
6. Segment Name 8 bytes
7. Length of Key Feedback Area 4 bytes
8. Number of Sensitive Segments 4 bytes
9. Key Feedback Area variable length

Figure 94. DB PCB Mask

COBOL DB PCB MASK

```
01 PCBNAME.
   02 DBD-NAME                PICTURE X(8).
   02 SEG-LEVEL               PICTURE XX.
   02 STATUS-CODE             PICTURE XX.
   02 PROC-OPTIONS            PICTURE XXXX.
   02 RESERVE-DL/I            PICTURE S9(5)COMPUTATIONAL.
   02 SEG-NAME-FB             PICTURE X(8).
   02 LENGTH-FB-KEY           PICTURE S9(5)COMPUTATIONAL.
   02 NUMB-SENS-SEGS          PICTURE S9(5)COMPUTATIONAL.
   02 KEY-FB-AREA             PICTURE X(17).
```

Define the PCB mask as a 01 level linkage section entry, with the fields of the PCB defined in the linkage section so that your program may reference them.

PL/I DB PCB MASK

```
DECLARE     PCB_POINTER POINTER;
DECLARE1    PCBNAME BASED (PCB_POINTER),
            2 DBD_NAME CHAR(8),
            2 SEG_LEVEL CHAR(2),
            2 STATUS_CODE CHAR(2),
            2 PROC_OPTIONS CHAR(4),
            2 RESERVE_DLI FIXED BIN(31,0),
            2 SEG_NAME FB CHAR(8),
            2 LENGTH_FB_KEY FIXED BIN(31,0),
            2 NUMB_SENS_SEGS FIXED BIN(31,0),
            2 KEY_FB_AREA CHAR(17);
```

In PL/I the PCB mask should be defined as a level 1 declarative. Although in the entry statement you must pass the pointer to the PCB, and not the PCB name, in DL/I calls you can pass either the pointer or the PCB name.

ASSEMBLER LANGUAGE DB PCB MASK

```
PCBNAME     DSECT
DBPCBDBD    DS      CL8         DBDNAME
DBPCBLEV    DS      CL2         LEVEL FEEDBACK
DBPCBSTC    DS      CL2         STATUS CODES
DBPCBPRO    DS      CL4         PROC OPTIONS
DBPCBRSV    DS      F           RESERVED
DBPCBSFD    DS      CL8         SEGMENT NAME FEEDBACK
DBPCBLKA    DS      F           CURRENT LENGTH OF
*                               KEY FEEDBACK AREA
DBPCBNSS    DS      F           NO OF SENSITIVE SEGMENTS
DBPCBKFA    DS      CL17        KEY FEEDBACK AREA
```

In assembler language you must define a fullword for each DB PCB. Your program can then access the status codes after a DL/I call using the PCB base addresses. When you issue a DL/I call, pass the register that contains the address of the PCB for the call, not the PCB name for the call.

I/O AREA

The I/O area for data base calls must be long enough to hold the longest segment your program retrieves from or adds to the data base. If your program issues any get or ISRT calls that use the D command code, the I/O area must be large enough to hold the largest path of segments that the program retrieves or inserts. You can use separate I/O areas for different segment types, or you can use only one.

COBOL I/O AREA

The I/O area in a COBOL program should be defined as a 01 level working storage entry. You can further define the area with 02 entries.

```
IDENTIFICATION DIVISION.
.
.
.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 INPUT-AREA.
   02 KEY PICTURE X(n).
   02 FIELD PICTURE X(n).
```

PL/I I/O AREA

In PL/I, the name for the I/O area used in the DL/I call can be the name of a fixed-length character string, a major structure, a connected array, or an adjustable character string. It cannot be the name of a minor structure or a character string with the attribute VARYING. If you want to define it as a minor structure, you can use a pointer to the minor structure as the parameter.

Your program should define the I/O area as a fixed-length character string and pass the name of that string, or define it in one of the other ways mentioned above and then pass the pointer variable that points to that definition. If you want to use substructures or elements of an array, use the DEFINED or BASED attribute.

```
DECLARE 1 INPUT_AREA,
        2 KEY CHAR(6),
        2 FIELD CHAR(84);
```

ASSEMBLER LANGUAGE I/O AREA

```
IOAREA   DS   0CL90
KEY      DS   CL6
FIELD    DS   CL84
```

SEGMENT SEARCH ARGUMENTS

This section gives information on restrictions when using SSAs and coding formats and examples for defining SSAs in COBOL, PL/I, and assembler language.

SSA CODING RULES

- You specify the label of the place in your program where you have defined the SSA as the DL/I call parameter. The SSA itself is defined in the data area of your program.
- The segment name field:
  - Must be 8 bytes long. If the name of the segment you're specifying is less than 8 bytes long, it should be left-justified and padded on the right with blanks.
  - Must contain a segment name that has been defined in the DBD that your application program uses. In other words, make sure you use the exact segment name, or your SSA will be invalid.
- If the SSA contains only the segment name, the ninth byte must be a blank.
- If the SSA contains one or more command codes:
  - The ninth byte must be an asterisk (*).
  - The last command code must be followed by a blank, unless the SSA contains a qualification statement. If the SSA contains a qualification statement, the command code must be followed by the left parenthesis of the qualification statement.
- If the SSA contains a qualification statement:
  - The qualification statement must begin with a left parenthesis and end with a right parenthesis.
  - There must not be any blanks between the segment name, or command code(s) if used, and the left parenthesis.
  - The field that contains the field name must be 8 bytes long. If the field name is less than 8 bytes, it must be left-justified and padded on the right with blanks. The field name must have been defined for the specified segment type in the DBD the application program is using.
  - The relational operator follows the field name. It must be 2 bytes long, and may be represented alphabetically or symbolically. Figure 95 lists the relational operators.

Symbolic	Alphabetic	Meaning
=b or b=	EQ	Must be equal to
>= or =>	GE	Must be greater than or equal to
<= or =<	LE	Must be less than or equal to
>b or b>	GT	Must be greater than
<b or b<	LT	Must be less than
¬= or =¬	NE	Must not be equal to

Figure 95. Relational Operators

- The comparative value follows the relational operator. The length of this value must be equal to the length of the field that you specified in "field name." This length is defined in the DBD. The comparative value must include leading zeros for numeric values or trailing blanks for alphabetic values as necessary.

- If you are using multiple qualification statements within one SSA (Boolean qualification statements) the qualification statments must be separated by one of these symbols:

  * or & dependent AND

  + or | logical OR

  # independent AND

  This symbol must come between the right parenthesis of the first qualification statement and the left parenthesis of the second qualification statement with no blanks between them.

SSA CODING FORMATS

COBOL SSA DEFINITION EXAMPLES

Below is the definition for an unqualified SSA that does not use any command codes:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
.
.
.
01 UNQUAL-SSA.
   02 SEG-NAME     PICTURE X(08) VALUE '        '.
   02 FILLER       PICTURE X     VALUE ' '.
```

You can use an SSA coded like this for each DL/I call that needs an unqualified SSA by supplying the name of the segment type you want during program execution.

For COBOL, use a 01 level working storage entry to define each SSA that the program will use. You then use the name you have given the SSA, in this case, "UNQUAL-SSA," as the parameter in the DL/I call.

The SSA below is an example of a qualified SSA that does not use command codes. If you were using command codes in this SSA, you would code the asterisk (*) and the command code between the

8-byte segment name field and the left parenthesis that begins the qualification statement.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
.
.
.
01 QUAL-SSA-MAST.
   02 SEG-NAME-M        PICTURE X(08) VALUE 'ROOT    '.
   02 BEGIN-PAREN-M     PICTURE X     VALUE '('.
   02 KEY-NAME-M        PICTURE X(08) VALUE 'KEY     '.
   02 REL-OPER-M        PICTURE X(02) VALUE ' ='.
   02 KEY-VALUE-M       PICTURE X(n)  VALUE 'vv...v'.
   02 END-PAREN-M       PICTURE X     VALUE ')'.
```

The above SSA would look like this:

```
ROOTbbbb(KEYbbbbbb=vv...v)
```

These SSAs are both taken from the COBOL skeleton program in Figure 66. You can see how they are used in a DL/I call by referring to this program.

PL/I SSA DEFINITION EXAMPLES

An unqualified SSA that does not use command codes looks like this in PL/I:

```
DCL 1    UNQUAL_SSA          STATIC UNALIGNED,
         2   SEG_NAME_U      CHAR(8) INIT('NAME    '),
         2   BLANK           CHAR(1) INIT(' ');
```

You can use an SSA coded like this for each DL/I call that needs an unqualified SSA by supplying the name of the segment type you want during program execution.

In PL/I you define SSAs in structure declarations. The unaligned attribute is required for SSA data interchange with DL/I. The SSA character string must reside contiguously in storage. For example, assignment of variable to key values could cause DL/I to construct an invalid SSA if the key value has changed the aligned attribute. A separate SSA structure is required for each segment type that the program accesses, because the value of the key fields differs among segment types. Once you have initialized the fields other than the key values, you should not have to change the SSAs again. You can define SSAs in any of the ways explained for the I/O area.

The following is an example of a qualified SSA without command codes. If you wanted to use command codes with this SSA, you would code the asterisk (*) and command codes between the 8-byte segment name field and the left parenthesis that begins the qualification statement.

```
DCL 1    QUAL_SSA            STATIC UNALIGNED,
         2    SEG_NAME           CHAR(8) INIT('ROOT    '),
         2    SEG_QUAL           CHAR(1) INIT('('),
         2    SEG_KEY_NAME       CHAR(8) INIT('KEY     '),
         2    SEG_OPR            CHAR(2) INIT(' ='),
         2    SEG_KEY_VALUE      CHAR(n) INIT('vv...v'),
         2    SEG_END_CHAR       CHAR(1) INIT(')');
```

This SSA would look like this:

```
ROOTbbbb(KEYbbbbbb=vv...v)
```

These SSAs are both taken from the PL/I skeleton program shown in Figure 67. You can see how they are used in DL/I calls by referring to this program.

ASSEMBLER LANGUAGE SSA DEFINITION EXAMPLES

The example below shows how you would define a qualified SSA without command codes.

```
*          CONSTANT AREA
.
.
.
SSANAME     DS      0CL26
ROOT        DC      CL8'ROOT    '
            DC      CL1'('
            DC      CL8'KEY     '
            DC      CL2' ='
NAME        DC      CLn'vv...v'
            DC      CL1')'
```

This SSA would look like this:

```
ROOTbbbb(KEYbbbbbb=vv...v)
```

DC CALLS

DC CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, i/o pcb or alternate pcb,
    i/o area [,mod name] [,destination name].
```

PL/I

```
CALL PLITDLI (parmcount, function, i/o pcb or alternate pcb,
    i/o area [,mod name] [,destination name]);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function,i/o pcb or alternate pcb,                X
    i/o area[,mod name][,destination name]),VL
```

DC CALL PARAMETERS

function
The address of a 4-byte area that contains one of the values below.

GUbb Get unique

GNbb Get next

ISRT Insert

CHNG Change

PURG Purge

CMDb Command

GCMD Get command

i/o pcb or alternate pcb
The name of the PCB to use for this call.

i/o area
The address of the I/O area to use for this call. The I/O area must be large enough to hold the largest segment passed between the program and IMS/VS.

mod name
Mod name is the name of an 8-byte field that contains the name of the MOD you want used for this output message. The name must be left-justified and padded with blanks as necessary. Mod name is an optional parameter that is valid only if your program uses MFS. If you specify a valid MOD name, IMS/VS uses that MOD to format the screen for the input message you are sending. You specify the MOD name only on the first ISRT call for an output message.

destination name
The name of an 8-byte field containing the name of the logical terminal transaction code to which you want messages using this PCB sent.

SUMMARY OF DC CALLS

Figure 96 shows the parameters that are valid for each of the DC calls.

Function Code	Meaning	Use	Parameters
GUbb	Get Unique	Retrieves first segment of message	function, i/o pcb, i/o area
GNbb	Get Next	Retrieves subsequent message segments	function, i/o pcb, i/o area
ISRT	Insert	Builds output message in program's I/O area	function, i/o or alt pcb, i/o area [,mod name]
CHNG	Change	Sets destination on modifiable alt PCB	function, alt pcb, destination name
PURG	Purge	Enqueues messages from a PCB to destinations	function, i/o pcb [,i/o area, mod name]
CMDb	Command	Allows program to enter IMS/VS commands	function, i/o pcb, i/o area
GCMD	Get command	Retrieves second and subsequent responses to command	function, i/o pcb, i/o area

Figure 96. Summary of DC Calls

Note: Language-dependent parameters are not shown here. "Parmcount" is required for all PL/I calls. Either "parmcount" or "VL" is required for assembler language calls. Parmcount is optional in COBOL programs.

Figure 97 is a summary of which system service calls you can use in each type of IMS/VS application program, and the parameters for each call.

Function Code	Meaning and Use	Options	Parameters	Valid for
CHKP	Basic checkpoint. Recovery purposes.	Can request OS/VS chkp	function, i/o pcb, i/o area [,chkp func]	batch, BMP,MPP
CHKP	Symbolic checkpoint. Recovery purposes.	Can specify seven program areas to be saved.	function, i/o pcb, i/o area len, i/o area, 1st area len, 1st area [,...,7th area len, 7th area]	batch, BMP
DEQb	Dequeue. Release segments enqueued with Q command code.	Can specify segments to release.	function, i/o pcb, work area	BMP, MPP
GSCD (1)	Get address of System contents directory.	None	function, either pcb, i/o area	batch
LOGb	Log. Write a message to the system log.	None	function, i/o pcb, i/o area	batch, BMP,MPP
ROLB	Rollback. Eliminate data base updates.	Call retuns last message to I/O area.	function, i/o pcb [,i/o area]	BMP, MPP
ROLL (2)	Roll. Eliminate data base updates; abend.	None	function	BMP, MPP
STAT	Statistics. Retrieve IMS/VS system stats	Choose type and format.	function, db pcb, i/o area, stat func	batch, BMP,MPP
XRST	Restart. Works with symbolic chkp to restart application program after failure.	Can specify 7 areas to be saved.	function, i/o pcb, i/o area len, i/o area, 1st area len, 1st area [,...,7th area len, 7th area]	batch, BMP

Figure 97. Summary of System Service Calls

Notes:

1. GSCD functions normally in a message processing or batch processing region using OS/VS2 or OS/VS1 with fetch protect. Use GSCD only in batch programs. The reason that you should not use it in BMPs and MPPs it that the operating system doesn't allow a program in one region to access data in another region. Since the addresses are from the control region, and MPPs and BMPs run in separate regions, these addresses cannot be used in a message processing or batch message processing region. If they are, an 0C4 system abend will occur.

2. You can issue a ROLL call in a batch program, but it will not back out data base updates. It will terminate the program. One reason you might want to issue ROLL in a batch program is for compatibility.

SYMBOLIC CHKP AND XRST CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, i/o pcb,
     i/o area len, i/o area, 1st area len, 1st area [,...,
     7th area len, 7th area].
```

PL/I

```
CALL PLITDLI (parmcount, function, i/o pcb, i/o area len,
     i/o area, 1st area len, 1st area [,..., 7th area len,
     7th area]);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function, i/o pcb,                                    X
     i/o area len,i/o area,1st area len,1st area [,...,             X
     7th area len,7th area]),VL
```

SYMBOLIC CHKP AND XRST PARAMETERS

Four of the parameters shown above are required: function, i/o pcb, i/o area len, and i/o area. All of the parameters except function and i/o area are identical for symbolic CHKP and XRST; these differences in these parameters are noted in in the parameter descriptions.

The call parameters for symbolic CHKP and XRST are:

parmcount
This parameter is required only for PL/I. An assembler language call must include either parmcount or VL. Parmcount is the address of a 4-byte field in the data area in your program that contains the number of parameters that follow in the call. The value does not include parmcount itself.

function
This parameter gives the address of the area in your program that contains the 4-byte function code. For symbolic CHKP this area must contain "CHKP"; for XRST this area must contain "XRST". You can use any name you want to for this area, including the value of the function code itself, within the conventions of your programming language.

i/o pcb
Symbolic CHKP and XRST must reference the I/O PCB, not the DB PCB.

i/o area len
This is the address of a 4-byte area in your program that contains the length in binary of the largest I/O area used by your program. If you are using PL/I, define the area as a substructure, but specify the name of the major structure in the call.

i/o area for symbolic CHKP
This is the name of the I/O area in your program that contains the 8-byte ID for this checkpoint. If you are using PL/I, specify this parameter as a pointer to a major structure, an array, or a character string.

i/o area for XRST
This is the name of a 12-byte area in your program. Your program should set this area to blanks (X'40') before issuing the XRST call, then test it after issuing the call. If your program is being started normally, the area will not have changed. If your program is being restarted from a

checkpoint, IMS/VS places the ID that you supplied in the CHKP call and in the restart JCL in the first 8 bytes of the I/O area.

1st area len
This parameter gives the address of a 4-byte field in your program that contains the length in binary of the first area to checkpoint. If you are using PL/I, specify the name of the major structure in the call, but define the area itself as a substructure. All 7 area parameters (and the corresponding length parameters) are optional. The number of areas you specify on a XRST call must be greater than or equal to the number of areas you specify on the CHKP calls the program issues. IMS/VS restores only the areas you specified in the CHKP call when you restart the program.

1st area
This parameter specifies the first area in your program that you want IMS/VS to checkpoint.

,...,7th area len, 7th area
You can specify as many as six more areas (a total of seven) of your program that you want IMS/VS to checkpoint. Always specify the length parameter first, followed by the area parameter. The parameters for the second through seventh program areas are coded like the parameters for the first area.

BASIC CHKP CALL FORMAT

COBOL

```
CALL 'CBLTDLI' USING function, i/o pcb,
     i/o area [,chkp function].
```

PL/I

```
CALL PLITDLI (parmcount, function, i/o pcb,
     i/o area [,chkp function]);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function, i/o pcb,                              X
     i/o area,chkp function or chkp DCB]),VL
```

BASIC CHKP PARAMETERS

There are three parameters on the basic CHKP call that are required: function, i/o pcb, and i/o area. The rest (parmcount, chkp function, chkp DCB, and VL) are either optional or language dependent.

parmcount
Parmcount is the address of a 4-byte field in your program's data area that contains the number of parameters that follow. This count does not include parmcount itself. Parmcount is required only for PL/I. It is optional for COBOL and assembler language. If you are using assembler language, however, you must include either parmcount or VL.

function
Function is the address of the area in your program that contains the 4-byte function code, "CHKP". This function code can be coded just like the function codes for any of the DL/I calls.

i/o pcb
A basic CHKP call must reference the I/O PCB.

i/o area
This parameter gives the name of your program's I/O area that contains the 8-byte checkpoint ID. This parameter should be coded as a pointer to a major structure, an array, or a character string.

chkp function
If you code this parameter of the basic CHKP call, IMS/VS requests that an OS/VS checkpoint be taken of your region in addition to the program checkpoint that IMS/VS takes. This parameter gives the name of the 8-byte area in your program that contains the value "OSVSCHKP".

If you specify this option, you must also supply DD statements for the OS/VS data sets you want checkpointed. IMS/VS will then provide DCBs and issue an OS/VS checkpoint before issuing the CHKP call. The DD names for the DCBs that IMS/VS provides are CHKDD and CHKDD2. If you have only one OS/VS checkpoint data set, use CHKDD. If you don't supply any DD statements, IMS/VS issues a U0475 error message. If the data set cannot be opened, OPEN will issue an error message.

chkp DCB (for assembler language programs only)
This parameter may be used instead of chkp function in assembler language programs requesting an OS/VS checkpoint.

GSCD CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, pcb, i/o area.
```

PL/I

```
CALL PLITDLI (parmcount, function, pcb, i/o area);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function,pcb,i/o area),VL
```

GSCD PARAMETERS

GSCD has three required parameters: function, pcb, and i/o area. In addition, parmcount is required for PL/I, and either parmcount or VL is required for assembler language.

parmcount
Parmcount gives the address of a 4-byte field containing the number of parameters to follow. PL/I calls must include this parameter.

function
This parameter gives the address of a 4-byte field in your program that contains the value "GSCD". You can code this area just as you do the function codes for the DL/I calls explained in "Coding DL/I Function Codes" later in this chapter.

pcb
GSCD can reference either a DB PCB or an I/O PCB.

i/o area
The I/O area referenced by this parameter must be 8-bytes long. IMS/VS places the address of the system contents

directory in the first 4 bytes and the address of the program specification table in the second 4 bytes. PL/I programs should specify this parameter as a pointer to a major structure, an array, or a character string.

VL
For assembler language programs only.

LOG CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, i/o pcb, i/o area.
```

PL/I

```
CALL PLITDLI (parmcount, function, i/o pcb, i/o area);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function,Rp,Rr),VL
```

LOG PARAMETERS

Three of the parameters shown above are required: function, i/o pcb (Rp for assembler language), and i/o area (Rr for assembler language). Parmcount is required for PL/I; VL applies only to assembler language.

parmcount
Parmcount gives the address of a 4-byte field that contains the number of parameters to follow.

function
This gives the address of the 4-byte field in your program that contains the value "LOGb". (LOG must have a blank after the G.) You can code this area just as you do any of the DL/I call function codes explained in "Coding DL/I Function Codes" later in this chapter.

i/o pcb
LOG must reference the I/O PCB, not the DB PCB.

i/o area
This parameter gives the address of the area in your program that contains the record that you want to write to the system log. This record must be in the following format:

LL A 2-byte field containing the length of the record. The total length includes:

- 2 for the count field (although in PL/I this field is actually 4 bytes long)
- 2 for the ZZ field
- 1 for the C field
- n for the length of the record itself

If you are using PL/I, your program must define the length field as a binary fullword.

ZZ A 2-byte field of binary zeros.

C A 1-byte field containing a log code. This code must be equal to or greater than X'A0' or equal to or less than X'E0' in value.

RESTRICTIONS ON LOG I/O AREA

There are some restrictions on the total length of the I/O area for the LOG call. They are:

- The length of the I/O area (including all fields) cannot be larger than the logical record length (LRECL) for the system log data set, minus 4 bytes;
- The length of the I/O area (including all fields) cannot be larger than the I/O area specified in the IOASIZE keyword of the PSBGEN statement of the PSB.

STAT CALL FORMATS

The STAT call retrieves statistics on the IMS/VS system. The call formats in COBOL, PL/I, and assembler language are:

COBOL

```
CALL 'CBLTDLI' USING function, db pcb, i/o area,
     stat function.
```

PL/I

```
CALL PLITDLI (parmcount, function, db pcb, i/o area,
     stat function);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function,db pcb,i/o area,stat function),VL
```

STAT PARAMETERS

The addresses of function, the DB PCB, the I/O area, and the statistics function are all required parameters for the STAT call.

parmcount
Parmcount is the address of a 4-byte field required in PL/I that gives the number of parameters to follow.

function
Function is the address of a 4-byte field containing the function code "STAT".

db pcb
The STAT call must reference the DB PCB.

i/o area
The I/O area pointed to by the i/o area parameter must be large enough to hold the statistics you are requesting. The size of this area depends on the type and format of the statistics you are requesting. You specify the type and format in the stat function parameter described below. PL/I programs should specify this parameter as a pointer to a major structure, an array, or a character string.

stat function
This parameter is the address of a 9-byte field that contains the following:

- 4 bytes that define the type of statistics you want:

  DBAS for ISAM/OSAM data base buffer pool statistics

  VBAS for VSAM data base subpool statistics

- 1 byte that gives the format you want the statistics in:

  F For the full statistics to be formatted. If you specify F, your I/O area must be at least 360 bytes.

  U For the full statistics to be unformatted. If you specify U, your I/O area must be at least 72 bytes.

  S For a summary of the statistics to be formatted. If you specify S, your I/O area must be at least 120 bytes.

- 4 bytes of EBCDIC blanks

STATUS CODE ERROR ROUTINE CALL FORMAT

this sample error routine is part of the IMS/VS Primer function. It should not be used in MPPs, because it opens OS data sets. An error routine for an MPP should send messages to the master terminal operator.

COBOL

```
CALL 'DFSOAER' USING db pcb, call label, area 1,
     options, area 2,..., area 9.
```

PL/I

```
CALL DFSOAER (db pcb, call label,
     area 1, options, area 2,..., area 9);
```

DFSOAER should be declared as an ENTRY constant with OPTIONS (ASSEMBLER).

Note: The parmcount parameter that is required in DL/I calls is not allowed in the call to the status code error routine.

ASSEMBLER LANGUAGE

```
CALL DFSOAER, (db pcb, call label,
     area 1, options, area 2,..., area 9), VL
```

STATUS CODE ERROR ROUTINE CALL PARAMETERS

The call parameters are:

db pcb
The name of the DB PCB used in the preceding DL/I call. In PL/I, you must pass the DB PCB name in this call; for DL/I calls you can pass either the name or the pointer.

call label
The label given to the preceding DL/I call. The required format is Dxxxxxxx if the call was issued against a DB PCB. If the last call was issued against the I/O PCB, the format is Cxxxxxxx.

options
The address of a 4-byte field containing one of the options below. You indicate the option you want in the leftmost byte (byte 0) of this field. The second, third, and fourth bytes of this field are reserved.

0 Return to caller after print. This lets your program call the routine more than once for testing purposes. You must still have a final invocation.

1 Abnormal termination after print; recommended for production programs.

2 Final invocation to close print data set and return control to the program.

3 IMS/VS will issue message DFS3125A. You can use this option when you want to test the recovery in your program. After receiving the message, the OS/VS system console operator continue your program; terminate your program abnormally; cause your program to go into a loop to test its error routine; or cancel your program.

area 1,..., area 9
The program areas you want the routine to print. The routine will print the first 76 characters in each area. One area is required; the maximum number of areas you can specify is nine.

SUGGESTIONS

- For normal program execution, use option 1.
- Since you can use option 2 to have the status code error routine return to your program, using this routine with option 2 can help you in testing your program.
- For PL/I, declare DFSOAER as an entry with OPTIONS (ASSEMBLER). Pass the name of the PCB in the statement and not the PCB pointer.

DEQ CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, i/o pcb, i/o area.
```

PL/I

```
CALL PLITDLI (parmcount, function, i/o pcb, i/o area);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function,i/o pcb,i/o area),VL
```

DEQ CALL PARAMETERS

function
The address of a 4-byte area that contains the value "DEQb".

i/o pcb
The name of the I/O PCB. The DEQ call must reference the I/O PCB.

i/o area
The name of a 1-byte area that contains one of the letters A, B, C, D, E, F, G, H, I, or J. This letter represents the segment you are dequeuing in this call. You must provide this value.

ROLB CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, i/o pcb
    [, i/o area ].
```

PL/I

```
CALL PLITDLI (parmcount, function, i/o pcb,
    [, i/o area [);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function,i/o pcb[,i/o area]),VL
```

ROLB CALL PARAMETERS

function
The name of a 4-byte field that contains the value "ROLB".

i/o pcb
The name of the I/O PCB. ROLB must reference the I/O PCB.

[,i/o area]
If you supply this parameter on a ROLB in an MPP, a message-driven Fast Path program, or a transaction-oriented BMP, IMS/VS returns the message you last processed to the I/O area.

ROLL CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function.
```

PL/I

```
CALL PLITDLI (parmcount, function);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function),VL
```

ROLL CALL PARAMETERS

function
The name of a 4-byte area in your program that contains the value "ROLL". This is the only parameter, other than parmcount for PL/I, and VL for assembler language.

GSAM REFERENCE

This section gives GSAM call formats in COBOL, PL/I and assembler language, and it defines the GSAM call parameters. It also shows how to code GSAM data areas.

GSAM CALL FORMATS

COBOL

```
CALL 'CBLTDLI' USING function, gsam pcb, i/o area
     [, rsa].
```

PL/I

```
CALL PLITDLI (parmcount, function, gsam pcb, i/o area
     [, rsa]);
```

ASSEMBLER LANGUAGE

```
CALL ASMTDLI,(function,gsam pcb,i/o area[,rsa]),VL
```

GSAM CALL PARAMETERS

The parameters for each GSAM call vary. All the calls require the address of the function and GSAM PCB; most of the calls require more than that.

An OPEN call has two required parameters—function and gsam pcb—and one optional one—the address of the OPEN option.

CLSE has only two required parameters and no optional parameters. The required parameters are function and gsam pcb.

GN and ISRT both have three required parameters and one optional one. Function, gsam pcb, and i/o area are required. RSA is optional.

Function, gsam pcb, i/o area, and rsa are all required for a GU.

parmcount
Required only for PL/I. This parameter gives the address of a 4-byte field that contains the number of parameters that follow.

function
This field gives the address of 4-byte field that contains one of the following function codes:

OPEN Opens GSAM data base explicitly

CLSE Closes GSAM data base explicitly

GNbb Gets the next record in the GSAM data base

ISRT Adds the record you supply to the end of the data base

GUbb Retrieves the record with the RSA you provide

gsam pcb
This parameter gives the name of the GSAM PCB.

i/o area
For OPEN calls, this area is optional. Using this parameter lets you specify the kind of data set you are opening.

This parameter is required for GN, ISRT, and GU calls. When you use one of these calls, the area named by this parameter contains the record that you are retrieving for GN and GU, or the record that you want to add for ISRT. This area must be long enough to hold these records.

rsa
This parameter gives the address of the area in your program that contains the record search argument. RSA is optional for GN and ISRT; it is required for GU.

GSAM DATA AREAS

The PCB mask, the I/O area, and the RSA that you use in a GSAM call have special formats.

GSAM DB PCB MASKS

GSAM DB PCB masks are slightly different from PCB masks for DL/I data bases. The fields that are different are the length of the key feedback area and the key feedback area. Also, there is an additional field that gives the length of the record being retrieved or inserted, when using undefined-length records. Figure 98 shows the fields of a GSAM DB PCB mask, and their required lengths.

1. Data Base Name 8 bytes
2. Not Used by GSAM 2 bytes
3. Status Code 2 bytes
4. Processing Options 4 bytes
5. Reserved for DL/I 4 bytes
6. Not Used by GSAM 8 bytes
7. Length of Key Feedback Area and Undefined-Length Records Area 4 bytes
8. Not Used by GSAM 4 bytes
9. Key Feedback Area 8 bytes
10. Length of Undefined-Length Records 4 bytes

Figure 98. GSAM DB PCB Mask Format

GSAM I/O AREAS

For OPEN calls, the I/O area must contain one of these values:

- INP for an input data set
- OUT for an output data set
- OUTA for an output data set with ASA control characters
- OUTM for an output data set with machine control characters

For GN, ISRT, and GU calls, the format of the I/O area depends on whether the record is fixed length, undefined length (valid only for BSAM), or variable length. For each kind of record, you have the option of using control characters.

The formats for an I/O area for fixed-length or undefined-length records are:

- Without control characters, the I/O area contains only data. The data begins in byte 0.
- With control characters, the control characters are in byte 0 and the data begins in byte 1.

If you are using undefined-length records, the record length is passed between your program and GSAM in the PCB field that usually contains the length of the key feedback area. When you are issuing an ISRT call, you supply the length; when you're issuing a GN or GU, GSAM places the length of the returned record in this field. This length field is 4 bytes long.

The formats for variable-length records differ, because variable-length records include one field that other records don't have: a length field. The length field is 2 bytes. Variable-length I/O areas, like fixed and undefined-length I/O areas, may have control characters.

- Without control characters, bytes 0 and 1 contain the 2-byte length field and the data begins in byte 2.
- With control characters, bytes 0 and 1 still contain the length field, but byte 2 contains the control characters and the data starts in byte 3.

GSAM RSAS

The contents of an RSA depend on the access method you're using. For BSAM tape data sets and VSAM data sets, the RSA contains the relative byte address (RBA). For BSAM disk data sets, the RSA contains the disk address (TTR).

For VSAM, the RBA is the relative byte address of a specific record within the data set. The RBA is contained in the first byte of the RSA.

For BSAM, the RSA is a doubleword used to locate individual records within a block. The first word contains the references to the BSAM RBA, or to the BSAM direct access storage device TTR. The second word gives the volume sequence number in the first halfword and the displacement within the block in the second halfword.

Your program should define an RSA as two fullwords on a fullword or doubleword boundary. Below is a PL/I example of an RSA:

```
DCL     1 GSAM_RSA
          2 BLOCK_ID FIXED BIN (31),
          2 VOL_SEQ_NO FIXED BIN (15),
          2 RECORD_DISP FIXED BIN (15);
DCL     1 FIRST_RECORD_RSA,
          2 (BLOCK1, DISP0) FIXED BIN (31) INIT(1);
```

GSAM JCL RESTRICTIONS

GSAM requires that you define the record format (RECFM) keyword on the DATASET statement in the GSAM DBD. You can override this in the record format (RECFM) subparameter in the JCL. Figure 99 shows you how you do this.

Record Format	Defined in GSAM DBD as RECFM=	Can be overridden in JCL with RECFM
Fixed	F, FB	FB, FBA, FBM
Variable	V, VB	VB, VBA, VBM
Undefined	U	Does not apply

Figure 99. GSAM JCL Restrictions

The following are some JCL guidelines and additional restrictions for GSAM.

- To add new records to the end of the data base (using the ISRT call), use DISP=MOD.
- Temporary data sets cannot be repositioned.
- The DD statement for new files must specify the volume serial number; GU calls require that the correct volume be available.
- DISP=DELETE and DISP=UNCATLG should not be used.
- Passed data sets should not be used.
- Backward references to data sets in previous steps should not be used.
- Use the following dispositions for output data sets:

  To create an output data set use DISP=NEW.

  To create an output data set that is cataloged use DISP=OLD.

  When issuing a restart to an output data set, use DISP=OLD.
- For input data sets, use DISP=OLD.
- Generation data groups and concatenated data sets processed as a single data set can't be restarted successfully.
- When you are restarting the program, use the DBD override parameters that were effective at the time of the checkpoint.
- When using GSAM to reference a data set whose physical attributes differ from the attributes that have been defined in the DBD (for example, logical record length, or LRECL, and block size, or BLKSIZE), you must supply the differing attributes in the JCL DC parameters for **all** references to that data set.

FAST PATH REFERENCE

This section contains reference information on Fast Path data base calls, Fast Path system service calls, and Fast Path message calls. It also contains information on coding data areas that are required by Fast Path calls.

FAST PATH DATA BASE CALLS

Figure 100 summarizes the calls you use to process Fast Path data bases.

Function Code	Types of MSDBs			DEDB
	Nonterminal Related	Terminal Related Fixed	Terminal Related Dynamic	
FLD	X	X	X	
POS				X
GU, GHU	X	X	X	X
GN, GHN	X	X	X	X
DLET			X	X
REPL	X	X	X	X
ISRT			X	X

Figure 100. Summary of Fast Path Data Base Calls

The coding differences between DL/I programs issuing data base calls and Fast Path programs issuing data base calls are as follows:

- Only 1 SSA is allowed in any data base call.
- The SSA cannot have Boolean qualification statements.
- The SSA cannot use any command codes.

FLD CALL FORMAT

COBOL

```
CALL 'CBLTDLI' USING function, msdb pcb, i/o area.
```

PL/I

```
CALL PLITDLI (parmcount, function, msdb pcb, i/o area);
```

Assembler Language

```
CALL ASMTDLI,(function,msdb pcb,i/o area),VL
```

FLD CALL PARAMETERS

The FLD call has three required parameters: function, msdb pcb, and i/o area. Function and msdb pcb are coded just like the function and db pcb parameters for DL/I calls. The I/O area, however, contains something unique to Fast Path: the field search argument, or FSA.

function
The address of the area in your program that contains the value "FLDb".

msdb pcb
The name of the MSDB PCB for this call. This parameter should be coded like the DB PCB parameter on the DL/I calls.

i/o area
This parameter points to the I/O area in your program that contains the FSA for this call.

POS CALL FORMAT

There is one call that you can issue against a DEDB that is not a DL/I call. This is the POS call.

COBOL

```
CALL 'CBLTDLI' USING function, dedb pcb,
     i/o area [,rootssa].
```

PL/I

```
CALL PLITDLI (parmcount, function, dedb pcb, i/o area
[,rootssa]);
```

Assembler Language

```
CALL ASMTDLI,(function,dedb pcb,i/o area [,rootssa]),VL
```

POS CALL PARAMETERS

parmcount
The address of a 4-byte field that contains the number of parameters to follow. Required for PL/I.

function
The address of the area in your program that contains the function code "POSb". This data area can be coded like the function code for any of the DL/I calls.

db pcb
The name of the PCB for the DEDB that you're using for this call.

i/o area
This parameter points to the I/O area in your program that you want to contain the positioning information returned by a successful POS call.

ssa
This optional parameter gives the name of the ssa that you want used in this call. The format of SSAs in POS calls is no different from the format of SSAs in DL/I calls.

FSAS

The FSA that you reference in a FLD call contains 5 fields. These fields and their required lengths are as follows:

- Field name—8 bytes
- FSA status code—1 byte
- Op code—1 byte
- Operand—variable
- Connector—1 byte

The rules for coding these areas are:

Field name
This field must be 8 bytes long. If the field name you're using is less than 8 bytes, the name must be padded on the right with blanks.

FSA status code
This area is 1 byte. IMS/VS returns one of the following status codes to this area after a FLD call:

b Successful

A invalid operation

B Operand length invalid

C Invalid call—program tried to change key field

D Verify check was unsuccessful

E Packed decimal or hexadecimal field is invalid

F Program tried to change an unowned segment

G Arithmetic overflow

Op code
This 1-byte field contains one of the following for a change operation:

+ To add the operand to the field value

- To subtract the operand from the field value

= To set the field value to the value of the operand

For a verify operation this field must contain one of the following:

E Verify that the field value and the operand are equal.

G Verify that the field value is greater than the operand.

H Verify that the field value is greater than or equal to the operand.

L Verify that the field value is less than the operand.

M Verify that the field value is less than or equal to the operand.

N Verify that the field value is not equal to the operand.

Operand
This field contains the value against which you want to test the field value. The data in this field must be the same type of data as the data in the field. This is defined in the DBD. If the data is hexadecimal, the value in the operand will be twice as long as the field in the data base. If the data is packed decimal, the operand will not contain leading zeros, so the operand length might be shorter than the actual field. For other types of data the lengths must be equal.

Connector
This 1-byte field must contain a blank if this is the last or only FSA, or an asterisk (*) if another FSA follows this one.

POS I/O AREA

The I/O area on a POS call contains five fields.

length field—2 bytes
After a successful POS call, IMS/VS places the length of the data area for this call in this field.

area name field—8 bytes
This area contains the ddname from the AREA statement.

position information—8 bytes
IMS/VS places two pieces of data in this field after a successful POS call. The first 4 bytes will contain the cycle count, and the second 4 bytes will contain the VSAM RBA. These two fields uniquely identify a sequential dependent segment during the life of an area.

Unused CIs in sequential dependent part—8 bytes
This field will contain the number of unused control intervals in the sequential dependent part

Unused CIs in independent overflow part—8 bytes
This field will contain the number of unused control intervals in the independent overflow part.

FAST PATH MESSAGE CALLS

Figure 101 summarizes the DC calls that are available to Fast Path message-driven programs.

Function Code	Meaning	Use	Parameters
GUbb	Get unique	Retrieves next message	function, i/o pcb, i/o area [,mod name]
GNbb	Get next	Compatibility only	function, i/o pcb, i/o area [,mod name]
CHNG	Change	Changes destination on response alternate PCB	function, alt pcb, destination name
ISRT	Insert	Builds output message	function, i/o pcb or alt pcb, i/o area [,mod name]

Figure 101. Fast Path Message Calls

FAST PATH SYSTEM SERVICE CALLS

Figure 102 summarizes the system service calls that are available to Fast Path programs.

Function Code	Meaning	Use	Parameters
CHKP	Basic checkpoint	Takes checkpoint of program. OS/VS option invalid for FP	function, i/o pcb, i/o area
ROLB	Rollback	Eliminates updates in this sync interval.	function, i/o pcb, i/o area
SYNC	Synchronization	Requests sync point processing.	function, i/o pcb

Figure 102. Fast Path System Service Calls

SYNC CALL FORMAT

You can use a SYNC call only in a nonmessage-driven Fast Path program.

COBOL

```
CALL 'CBLTDLI' USING function, i/o pcb.
```

PL/I

```
CALL PLITDLI (parmcount, function, i/o pcb);
```

Assembler Language

```
CALL ASMTDLI,(function,i/o pcb),VL
```

SYNC CALL PARAMETERS

function
The address of the area in your program that contains the value "SYNC".

i/o pcb
The name of the I/O PCB.

IMS/VS STATUS CODES

This section contains reference information on all of the IMS/VS status codes. This information is given in two parts: first, the IMS/VS Status Codes Quick Reference gives a brief explanation for each status code, and it shows you the calls for which you can receive each status code. The quick reference also gives a number that represents the category of each status code. Second, this section contains more detailed explanations of each of the status codes, including possible causes for status codes, and what you can do to fix the problem. This information follows the quick reference.

IMS/VS STATUS CODES QUICK REFERENCE

IMS/VS status codes fall into five categories. They are:

1. Those indicating exceptional but valid conditions. The call is completed.
2. Those indicating warning or information-only status codes on successful calls (for example, GA and GK). If the call requested data, IMS/VS returns the data to the I/O area. The call is completed.
3. Those indicating warning status codes on successful calls when data is not returned to the I/O area. Call is completed.
4. Those indicating a programming error. This is the principal category. The call is not completed.
5. Those indicating system, I/O, or security errors encountered during the execution of I/O requests. The call is not completed.

STATUS CODES

STATUS CODE	DATA BASE CALLS								MSG CALLS							SYSTEM SERVICE CALLS									CATEGORY	DESCRIPTION
	GU GHU	GN GHN	GNP GHNP	DLET REPL	ISRT (LOAD)	ISRT (ADD)	FLD	POS	GU	GN	ISRT	CHNG	CMD	GCMD	PURG	CHKP	ROLL	DEQ	LOG	SNAP	SYNC	ROLB	STAT	XRST		
AA											X	X													4	CHNG CALL FOR RESPONSE ALTERNATE PCB CAN ONLY SPECIFY LOGICAL TERMINAL DESTINATION; TRANSACTION CODE DESTINATION SPECIFIED.
AB	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X		X	X						4	SEGMENT I/O AREA REQUIRED, NONE SPECIFIED IN CALL.
AC	X	X	X		X	X																	X		4	HIERARCHIC ERROR IN SSAs.
AD	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			4	FUNCTION PARAMETER INVALID.
AF	X	X																							4	GSAM DETECTED INVALID VARIABLE-LENGTH RECORD
AH					X	X																			4	REQUIRED SSA MISSING.
AI	X	X	X	X	X	X																			5	DATA MANAGEMENT OPEN ERROR.
AJ	X	X	X	X	X	X	X	X																	4	SSA QUALIFICATION FORMAT INVALID.
AK	X	X	X		X	X	X	X																	4	INVALID SSA FIELD NAME.
AL									X	X	X	X	X	X	X							X			4	CALL USING I/O-PCB IN BATCH PGM.
AM	X	X	X	X	X	X	X	X																	4	CALL FUNCTION NOT COMPATIBLE WITH PROCESSING OPTION OR SGMT SENSITIVITY.
AO	X	X	X	X	X	X		X																	5	I/O ERROR ISAM, OSAM, BSAM, OR VSAM.
AP									X	X	X	X	X	X	X	X									4	MORE THAN FOUR USER CALL PARAMETERS FOR A TPPCB ARE INVALID.
AT				X	X	X					X				X				X						4	USER I/O AREA TOO LONG.
AU	X	X	X	X	X	X																			4	SSAs TOO LONG.
AY											X														4	RESPONSE ALTERNATE PCB REFERENCED BY ISRT CALL HAS MORE THAN ONE PHYSICAL TERMINAL ASSIGNED FOR INPUT PURPOSES. NOTIFY MASTER TERMINAL.
AZ															X										4	THE CONVERSATIONAL PROGRAM HAS ISSUED A PURGE CALL TO PCB THAT CANNOT BE PURGED.
A1												X													4	CALL ATTEMPTED WITH 8 CHARACTER LOGICAL TERMINAL NAME NOT KNOWN TO SYSTEM.
A2												X													4	CALL ATTEMPTED WITH INVALID PCB (PCB NOT MODIFIABLE OR ISRT OPERATION ALREADY DONE).
A3											X				X										4	CALL ATTEMPTED TO A MODIFIABLE TP PCB WITH NO DESTINATION SET.
A4												X													4	SECURITY VIOLATION.
A5											X				X										4	FORMAT NAME SPECIFIED ON 2ND OR SUBSEQUENT MSG ISRT OR PURG.
A6											X														4	OUTPUT SEGMENT SIZE LIMIT EXCEEDED ON CALL.
A7											X														4	NUMBER OF OUTPUT SEGMENTS INSERTED EXCEEDED THE LIMIT BY ONE.
A8											X														4	ISRT TO RESPONSE ALTERNATE PCB FOLLOWED ISRT TO I/O PCB, OR VICE VERSA.
A9											X														4	RESPONSE ALTERNATE PCB REFERENCED BY CALL REQUIRES THAT THE SOURCE PHYSICAL TERMINAL RECEIVE THE OUTPUT RESPONSE.
CA													X												4	NO SUCH COMMAND. NO COMMAND RESPONSES PRODUCED.
CB													X												4	COMMAND, AS ENTERED, NOT ALLOWED FOR AOI. NO COMMAND RESPONSE PRODUCED.
CC													X												2	COMMAND EXECUTED. ONE OR MORE COMMAND RESPONSES PRODUCED.
CD													X												4	ENTERED COMMAND VIOLATES SECURITY. NO COMMAND RESPONSES PRODUCED.
CE									X																2	TRANSACTION RESCHEDULED AFTER CMD CALL. SYNC POINT HAD NOT BEEN REACHED.
CF									X																2	MESSAGE ON QUEUE BEFORE IMS/VS WAS LAST STARTED.
CG									X																2	TRANSACTION ORIGINATED FROM AOI USER EXIT.

Figure 103 (Part 1 of 3). IMS/VS Status Codes Quick Reference

STATUS CODES (Continued)

STATUS CODE	DATA BASE CALLS								MSG CALLS							SYSTEM SERVICE CALLS									CATEGORY	DESCRIPTION
	GU GHU	GN GHN	GNP GHNP	DLET REPL	ISRT (LOAD)	ISRT (ADD)	FLD	POS	GU	GN	ISRT	CHNG	CMD	GCMD	PURG	CHKP	ROLL	DEQ	LOG	SNAP	SYNC	ROLB	STAT	XRST		
CH													X												5	AOI DETECTED SYSTEM ERROR; CMD REQUEST NOT PROCESSED. REISSUE CMD CALL.
CI									X																2	TRANSACTION ON QUEUE BEFORE IMS/VS LAST STARTED. TRANS. RESCHEDULED SYNC PT NOT REACHED.
CJ									X																2	TRANSACTION FROM AOI USER EXIT. MSG RESCHEDULED. SYNC POINT NOT REACHED.
CK									X																2	TRANSACTION FROM AOI USER EXIT. MSG ON QUEUE BEFORE IMS/VS LAST STARTED.
CL									X																2	TRANSACTION FROM AOI USER EXIT. MSG ON QUEUE BEFORE IMS/VS LAST STARTED. MSG RESCHEDULED. SYNC PT HAD NOT BEEN REACHED.
DA				X			X																		4	SEGMENT KEY FIELD OR NON-REPLACEABLE. FIELD HAS BEEN CHANGED.
DJ				X																					4	NO PRECEDING SUCCESSFUL GHU OR GHN CALL OR AN SSA SUPPLIED AT A LEVEL NOT RETRIEVED.
DX				X																					4	VIOLATED DELETE RULE.
FA	X															X					X				2	MSDB ARITHMETIC OVERFLOW ERROR OCCURRED.
FC								X																	4	POS CALL FOR DIR. DEP. SEGS ONLY.
FD	X	X	X	X	X	X	X		X							X					X				2	DEADLOCK OCCURRED.
FE							X																		4	FSA ERROR, NOT FIELD NAME.
FF						X			X							X									3	NO SPACE IN MSDB.
FG							X																		4	COMBINATION OF FE AND FW CODES.
FH	X	X	X	X	X			X	X							X					X				3	DEDB INACCESSIBLE.
FI	X	X	X	X	X			X		X	X	X				X	X					X			4	I/O AREA NOT IN USER'S DEPENDENT REGION.
FN							X																		4	FSA ERROR, FIELD NAME.
FP				X		X	X																		4	INVALID HEX OR DEC DATA.
FR	X	X	X	X	X	X	X	X																	5	TOTAL BUFFER ALLOCATION EXCEEDED.
FS					X				X							X					X				3	DEDB AREAS ARE FULL.
FT	X	X	X	X	X			X																	4	TOO MANY SSA's ON CALL.
FV									X							X					X				3	MSDB VERIFY CONDITION FAILED.
FW	X	X	X	X	X	X	X	X			X	X													2	MORE RESOURCES NEEDED THAN NORMALLY ALLOWED.
GA		X	X																				X		2	CROSSING HIERARCHICAL BOUNDARY.
GB		X																							1	END OF DATA.
GC	X	X			X			X																	3	CROSSING UOW BOUNDARY.
GD						X																			1	CALL DID NOT HAVE SSAs FOR ALL LEVELS ABOVE INSERT OR HAS LOST SEGMENT POSITION.
GE	X	X	X			X			X							X							X		1	SEGMENT NOT FOUND.
GG	X	X	X																						5	SEGMENT CONTAINS INVALID POINTER.
GK		X	X																						2	CROSSING SEGMENT BOUNDARIES ON SAME LEVEL.
GL																		X	X						4	INVALID USER LOG CODE.
GP			X			X		X																	4	NO PARENTAGE ESTABLISHED.
II					X	X																			3	SEGMENT ALREADY EXISTS.
IX						X																			4	VIOLATED INSERT RULE.
LB					X																				1	SEGMENT BEING LOADED ALREADY EXISTS IN DATA BASE.
LC					X																				4	KEY FIELD OF SEGMENTS OUT OF SEQUENCE.
LD					X																				4	NO PARENT FOR THIS SEGMENT HAS BEEN LOADED.
LE					X																				4	SEQUENCE OF SIBLING SEGMENTS NOT THE SAME AS DBD SEQUENCE.

Figure 103 (Part 2 of 3). IMS/VS Status Codes Quick Reference

STATUS CODES (Continued)

STATUS CODE	DATA BASE CALLS								MSG CALLS							SYSTEM SERVICE CALLS									CATEGORY	DESCRIPTION
	GU GHU	GN GHN	GNP GHNP	DLET REPL	ISRT (LOAD)	ISRT (ADD)	FLD	POS	GU	GN	ISRT	CHNG	CMD	GCMD	PURG	CHKP	ROLL	DEQ	LOG	SNAP	SYNC	ROLB	STAT	XRST		
Nh				X	X	X																			5	INDEX MAINTENANCE RECEIVED AN UNEXPECTED RETURN CODE FROM BUFFER HANDLER.
NE				X																					3	DL/I CALL ISSUED BY INDEX MAINTENANCE CANNOT FIND SEGMENT.
NI				X	X	X																			1	INDEX MAINTENANCE FOUND DUPLICATE SEGMENT IN INDEX.
NO				X	X	X																			5	I/O ERROR ISAM, OSAM, BSAM OR VSAM.
QC									X							X									3	NO MORE INPUT MESSAGES EXIST.
QD										X				X											3	NO MORE SEGMENTS EXIST FOR THIS MESSAGE.
QE										X				X											4	GN REQUEST BEFORE GU. GMCD REQUEST BEFORE CMD.
QF									X		X				X	X									4	SEGMENT LESS THAN FIVE CHARACTERS (SEG LENGTH IS MSG TEXT LENGTH PLUS FOUR CONTROL CHARACTERS).
QH											X				X										4	TERMINAL SYMBOLIC ERROR-OUTPUT DESIGNATION UNKNOWN TO IMS/VS (LOGICAL TERMINALS OR TRAN CODE).
RX				X																					4	VIOLATED REPLACE RULE.
UC																									1	CHECKPOINT TAKEN.*
UR																									1	RESTART*
US																									1	STOP*
UX																									1	CHECKPOINT AND STOP*
VI				X	X																				4	SEGMENT LENGTH NOT WITHIN LIMITS OF DBDGEN.
XA											X														4	ATTEMPT TO CONT. PROC. CONV. BY PASSING SPA VIA PGM-TO-PGM SWITCH AFTER ANSWERING TERMINAL.
XB											X															PGM PASSED SPA TO OTHER PGM BUT TRYING TO RESPOND.
XC											X														4	PROGRAM INSERTED MESSAGE WITH Z1 FIELD BITS SET. THESE BITS RESERVED FOR SYSTEM USE.
XD																X					X				1	IMS IS TERMINATING. FURTHER DL/I CALLS MUST NOT BE ISSUED. NO MESSAGE RETURNED.
XE											X														4	TRIED TO ISRT SPA TO EXPRESS PCB.
XF											X														4	ALTERNATE PCB REFERENCED IN ISRT CALL FOR SPA HAD DESTINATION SET TO A LOGICAL TERMINAL, BUT WAS NOT DEFINED AS ALTRESP = YES.
XG											X														4	CURRENT CONVERSATION REQUIRES FIXED-LENGTH SPAs. ATTEMPT WAS MADE TO INSERT SPA TO TRANSACTION WITH A DIFFERENT OR NON-FIXED LENGTH SPA.
XX	X	X			X	X																			5	INTERNAL GSAM ERROR.
X1											X					X									5	I/O ERROR WRITING SPA.
X2											X				X										4	1ST INSERT TO TRAN COD PCB THAT IS CONVERSATIONAL IS NOT AN SPA.
X3											X														4	INVALID SPA.
X4											X														4	INSERT TO A TRAN CODE PCB THAT IS NOT CONVERSATIONAL AND THE SEGMENT IS AN SPA.
X5											X														4	INSERT OF MULTIPLE SPAs TO TRAN CODE PCB.
X6											X														4	INVALID TRAN CODE NAME INSERTED INTO SPA.
X7											X														4	LENGTH OF SPA IS INCORRECT (USER MODIFIED FIRST SIX BYTES).
X8											X														5	ERROR ATTEMPTING TO QUEUE AN SPA ON A TRAN CODE PCB.
X9											X														4	SPA LENGTH IS GREATER THAN THE I/O AREA SPECIFIED IN PSB.
bb	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	1	GOOD. NO STATUS CODE RETURNED, PROCEED.

* Utility Control Facility Status Codes b indicates a blank

Figure 103 (Part 3 of 3). IMS/VS Status Codes Quick Reference

AA

Explanation: IMS/VS ignored a CHNG or ISRT call because the response alternate PCB referenced in the call specified a transaction code as a destination. A response alternate PCB must have a logical terminal specified as its destination.

Programmer Response: Correct the CHNG or ISRT call.

AB

Explanation: An I/O area is required as one of the parameters on this call and the call did not specify one.

Programmer Response: Correct the call by including the address of an I/O area as one of the call parameters.

AC

Explanation: There is an error in one of the SSAs on a get or ISRT call for one of these reasons:

- DL/I could not find a segment in the DB PCB specified in the call that has the segment name given in the SSA.
- The segment name is in the DB PCB, but the SSA specifying that segment name is not in its correct hierarchic sequence.
- The call specifies two SSAs for the same hierarchic level.

IMS/VS also returns this status code when a STAT call has an invalid statistics function.

Programmer Response: Correct the segment name in the SSA, or the statistics function in the STAT call.

AD

Explanation: The call function parameter on the call is invalid. IMS/VS returns an AD status code if it does not recognize the function code you've supplied. If the function code is correct, some other possible causes are:

- Referencing a DB or alternate PCB on a CHKP call. CHKP calls must reference the I/O PCB.
- Issuing a message GU or GN that references an alternate PCB instead of the I/O PCB
- Using an invalid function string
- Referencing an I/O or alternate PCB for a data base call
- Referencing a DB PCB in a message call
- Issuing a ROLB that includes the address of an I/O area as one of the parameters in a batch-oriented BMP

Programmer Response: If you receive this status code on a data base, message, or CHKP call, correct the call so that it references the correct PCB. If you receive AD on a ROLB call in a batch-oriented BMP, remove the I/O area parameter from the call.

AF

Explanation: GSAM detected a variable-length record whose length is invalid on a GU, GHU, GN, or GHN.

Programmer Response: Correct the program.

AH

Explanation: The program issued an ISRT call (load or add) that did not include any SSAs. ISRT calls require SSAs. If the program was issuing a GU call to a GSAM data base, the GU did not specify an RSA. RSAs are required on GU calls to GSAM data bases.

Programmer Response: Correct the ISRT call by including an SSA, or correct the GU call by adding an RSA to the call.

AI

Explanation: A data management open error occurred. Some possible reasons are:

- There is an error in the DD statements.
- The data set OPEN request did not specify load mode, but the data set was empty. An empty data set requires the load option in the PCB.
- The buffer is too small to hold a record that was read at open time. See the storage estimates in the section "IMS/VS Data Base Buffer Pools," in the IMS/VS System Programming Reference Manual for specification of the minimum buffer pool size.
- There were no DD statements supplied for logically-related data bases.

- For an OSAM data set, the DSORG field of the OSAM DCB, DSCB, or JFCB does not specify PS or DA.
- For an old OSAM data set, the BUFL or BLKSIZE field in the DSCB is zero.
- The data set is being opened for load, and the processing option for one or more segments is other than L or LS.
- The allocation of the OSAM data set is invalid. The allocation is probably (1,,1) rather than (1,1) and this causes the DSORG to be PO.
- The processing option is L, the OSAM data set is old, and the DSCB LRECL and/or BLKSIZE does not match the DBD LRECL and/or BLKSIZE.
- Incorrect or missing information prevented IMS/VS from determining the block size or the logical record length.
- A catalog was not available for accessing a VSAM data base that was requested.
- OS could not perform an OPEN, but the I/O request is valid. Either the data definition information is incorrect, or information is missing.
- RACF was used to protect the ISAM or OSAM data set and the control region has no update authorization.

If IMS/VS returns message DFS0730I, you can determine the cause of the OPEN failure from this message. See the description of this message in the IMS/VS Messages and Codes Reference Manual for more information.

Programmer Response: These kinds of problems often require the help of a system programmer or system administrator. But before you go to one of these specialists, there are some things you can do:

- Check the DD statements. Make sure that the DD name is the same as the name specified on the DATASET statement of the DBD. The segment name area in the DB PCB has the DD name of the data set that couldn't be opened.
- Check the PSB and make sure that the appropriate processing options have been specified for each of the DB PCBs that your program uses.

AJ

Explanation: The format of one of your SSAs is invalid. Some possible reasons for this are:

- The SSA contains a command code that is invalid for that call.
- The relational operator in the qualification statement is invalid.
- A qualification statement is missing a right parenthesis or a Boolean connector.
- A DLET call has multiple or qualified SSAs.
- A REPL call has qualified SSAs.
- An ISRT call has the last SSA qualified.
- An ISRT call that inserts a logical child segment into an existing data base includes the D command code. ISRT calls for logical child segments cannot be path calls.
- The RSA parameter on a GSAM call is invalid.

Programmer Response: Correct the invalid portion of the SSA on the DLET, REPL, or ISRT call. If you receive this status code on a GSAM call, correct the RSA.

AK

Explanation: An SSA contains an invalid field name: the field name isn't defined in the DBD. The number in the segment level number field of the DB PCB is the level number of the SSA that contains the invalid name.

You can also receive this status code if the program is accessing a logical child through the logical parent. DL/I returns AK if the field specifed in the SSA has been defined for the logical child segment, and it includes (at least partially) the portion of the logical child that contains the concatenated key of the logical parent.

Programmer Response: Correct the SSA.

AL

Explanation: A batch program issued a message call or ROLB and referenced an I/O PCB.

Programmer Response: Correct the program. Batch programs cannot issue message or ROLB calls.

AM

Explanation: The call function is not compatible with the processing option in the PCB, segment sensitivity, or the transaction-code definition. The level

number in the PCB is the level number of the SSA that is invalid. Some of the reasons you might get this status code are:

- Issuing a retrieval call with the D command code in a program that doesn't have the P processing option specified in the DB PCB that was used for the call.
- Issuing an ISRT call with the D command code in an MPP or BMP that doesn't have the P processing option specified in the DB PCB that was referenced in the call. Batch programs do not need the P processing option to issue an ISRT call with the D command code—unless the program uses field level sensitivity.
- The processing option is L and the program issued a call other than an ISRT call. Load programs can issue only ISRT calls.
- Issuing a DLET, REPL, or ISRT call that references a DB PCB that doesn't have the necessary processing option for that call. The minimum processing options for these calls are D for DLET; R for REPL; and I for ISRT.
- Issuing a DLET, REPL, or ISRT call for a segment to which the program isn't sensitive.
- Issuing a DLET, REPL or ISRT while processing a transaction that has been defined as inquiry only.
- Issuing a CHKP call if a GSAM/VSAM data set is opened for output.
- Issuing a GSAM call with an invalid call function code.
- Issuing an ISRT or DLET call for the index target segment or a segment on which the index target is dependent in the physical data base while using an alternate processing sequence.
- Issuing a call to a GSAM dummy data set. Any call to a GSAM dummy data set is invalid.

Programmer Response: Correct the call, or make the necessary changes in the PCB.

AO

Explanation: There is a BSAM, GSAM, ISAM, VSAM, or OSAM physical I/O error. When issued from GSAM, this status code means that the error occurred when:

1. A data set was accessed
2. The CLOSE SYNAD routine was entered. The error occurred when the last block of records was written prior to closing of the data set.

IMS/VS does not return an AO status code for write errors with BISAM, VSAM, and OSAM.

Programmer Response: Determine whether the error occurred during input or output, and correct the problem.

AP

Explanation: A message or CHKP call has more than four parameters. This is invalid. In Fast Path programs, a message call included more than one SSA. Only one SSA is allowed.

Programmer Response: Correct the call and reprocess the transaction.

AT

Explanation: The length of the data in the program's I/O area is greater than the area reserved for it in the control region. The length of the area reserved is defined by the ACB utility program, DFSUACB0, and printed as part of its output.

Programmer Response: Correct the PSB or the program in error.

AU

Explanation: The total length of the SSAs in the data base call is greater than the area reserved for them in the control region. The length of the area reserved is defined by the ACB utility program, DFSUACB0, and printed as part of its output.

Programmer Response: Correct the PSB or the program in error.

AY

Explanation: IMS/VS ignored a message ISRT call because the logical terminal referenced by the response alternate PCB currently has more than one physical terminal assigned to it for input purposes.

Programmer Response: Ask the master terminal operator to determine (use /DISPLAY ASSIGNMENT LTERM x) which physical terminals (2 or more) refer to this logical terminal. Use the /ASSIGN command to correct the problem.

AZ

Explanation: IMS/VS ignored a PURG or ISRT call in a conversational program.

Some possible reasons are:

- Issuing a PURG call referencing the I/O PCB or an alternate response PCB. Conversational programs can issue PURG calls only when the PURG call references an alternate PCB that is not an alternate response PCB.
- Issuing a PURG call to send the SPA.
- Issuing an ISRT or a PURG call referencing an alternate PCB that is set for an invalid destination, or a destination that IMS/VS can't determine.
- Issuing an ISRT call referencing an alternate PCB whose destination is a conversational transaction code when the first segment inserted isn't the SPA; or when IMS/VS can't determine whether or not the SPA was the first segment inserted.

Programmer Response: Correct the PURG or ISRT call.

A1

Explanation: The logical terminal name supplied in the I/O area of a CHNG call is invalid. If IMS/VS returns A1 to a Fast Path program, it means that the program supplied a transaction code, instead of a logical terminal name, on a CHNG call.

Programmer Response: Correct the CHNG call.

A2

Explanation: The program issued a CHNG call against an invalid PCB. The PCB was invalid for one of these reasons:

- It was not an alternate PCB.
- It was an alternate PCB, but it wasn't modifiable.
- It was being used to process a message and had not completed processing it.

Programmer Response: Check the PCB that was used by the CHNG call and determine which PCB should have been used for the call.

A3

Explanation: The program issued an ISRT call or a PURG call using a modifiable alternate PCB that did not have its destination set.

Programmer Response: Issue a CHNG call to set the destination of the modifiable alternate PCB, then reissue the ISRT or PURG call.

A4

Explanation: A security violation occurred because the terminal entering the current transaction code was not authorized to enter that transaction code.

Programmer Response: Check the transaction code to make sure it was entered correctly. If it was, check with the person at your installation who handles security.

A5

Explanation: An ISRT or PURG call supplied an invalid parameter list. The call supplied the fourth parameter (the MOD name) but the ISRT or PURG being issued was not for the first segment of an output message.

Programmer Response: Correct the ISRT or PURG call.

A6

Explanation: IMS/VS ignored a message ISRT call because the length of the message segment being inserted exceeds the maximum length allowed.

Programmer Response: Correct the output message segment.

A7

Explanation: IMS/VS ignored a message ISRT call because the number of message segments inserted exceeds the limit specified by one. If the program tries to insert too many message segments before issuing a GU again, IMS/VS will terminate the program abnormally.

Programmer Response: Check the output messages and correct them.

A8

Explanation: IMS/VS ignored an ISRT call because:

- An ISRT call to a response alternate PCB must not follow an ISRT call to the I/O PCB.
- An ISRT call to the I/O PCB must not follow an ISRT call to a response alternate PCB.

Programmer Response: Correct the ISRT

call.

A9

Explanation: IMS/VS ignored the ISRT call because:

- The ISRT call referenced an alternate response PCB defined as SAMETRM=YES, but the PCB represented a logical terminal that isn't part of the originating physical terminal. An alternate response PCB defined as SAMETRM=YES must represent the same physical terminal as the physical terminal associated with the originating logical terminal.
- The originating terminal is in response mode and the response alternate PCB is not associated with that logical terminal.

IMS/VS does not return this status code if the program makes either of these errors while communicating with a terminal in a remote IMS/VS system through MSC.

Programmer Response: Determine whether the application program is in error, the output logical terminal has been incorrectly reassigned (using the /ASSIGN command), or if SAMETRM=YES should not have been specified for the response alternate PCB.

CA

Explanation: The program issued a CMD call with an invalid command verb, or the command verb does not apply to the IMS/VS system that the program's running in. IMS/VS does not return any command responses.

Programmer Response: Correct the command in the CMD call.

CB

Explanation: The command entered in the CMD call is not allowed from an AOI program. IMS/VS does not return any command responses.

Programmer Response: Correct the command. For a list of the commands that an AOI program can issue, see Chapter 7, "Automated Operator Programming," in the IMS/VS System Programming Reference Manual.

CC

Explanation: IMS/VS has executed the command and returned one or more command responses.

Programmer Response: Your program should issue GCMD calls as necessary to retrieve the responses.

CD

Explanation: The command that was entered on the CMD call violates security, or the application program isn't authorized to issue CMD calls. IMS/VS does not execute the command or return any command responses.

Programmer Response: Correct the command. If necessary, check with the person in charge of security at your installation to find out why your program is restricted from using that command.

CE

Explanation: IMS/VS rescheduled the message that this GU call retrieved since the last CMD call. The program had not reached a sync point when the message was rescheduled.

Programmer Response: This is an information-only status code.

CF

Explanation: The message retrieved by this GU was scheduled before IMS/VS was last started.

Programmer Response: This is an information-only status code.

CG

Explanation: The message retrieved by this GU originated from an AOI user exit.

Programmer Response: This is an information-only status code.

CH

Explanation: IMS/VS ignored the CMD call just issued because the AOI command interface detected a system error and was unable to process the command. IMS/VS processing continues.

Programmer Response: Reissue the command.

CI

Explanation: CI is a combination of CE and CF. The message retrieved by this GU was scheduled for transmission before IMS/VS was last started. The message was rescheduled, but the program hadn't reached a sync point.

Programmer Response: This is an information-only status code.

CJ

Explanation: CJ is a combination of CE and CG. The message retrieved by this GU was scheduled for transmission before IMS/VS was last started. The message originated from an AOI user exit.

Programmer Response: This is an information-only status code.

CK

Explanation: CK is a combination of CF and CG. The message retrieved with this GU originated from an AOI user exit. The message was scheduled for transmission before IMS/VS was last started.

Programmer Response: This is an information-only status code.

CL

Explanation: CL is a combination of CE, CF, and CG. The message retrieved with this GU originated from an AOI user exit. It was scheduled for transmission before IMS/VS was last started. It was rescheduled but the program had not reached a sync point.

Programmer Response: This is an information-only status code.

DA

Explanation: The program issued a REPL call that tried to modify the key field in the segment. You cannot change a segment's key field.

Programmer Response: Correct the REPL call.

DJ

Explanation: The program issued a DLET or REPL call without first issuing a successful get hold call; or an SSA in the DLET or REPL call was for a segment that was not retrieved in the get hold call.

Programmer Response: Correct the program by issuing a get hold call before the DLET or REPL call, or correct the get hold call or SSA.

DX

Explanation: The program issued a DLET call that violates the delete rule for that segment.

Programmer Response: Check the program to see whether or not the program should delete that segment; if it should, check with your DBA (or the equivalent specialist at your installation) to find out what delete rule has been specified for that segment.

FA

Explanation: IMS/VS returns this status code when the program reaches a sync point and an arithmetic overflow in an MSDB has occurred during that sync interval (since the last sync point, or, if the program had not reached a sync point, since the program began processing). You can receive this status code on a SYNC call, a CHKP call, or a GU call to the message queue, depending on your program. The overflow occurred after the program issued a FLD/CHANGE call or a REPL call for the MSDB. When this happens, IMS/VS issues an internal ROLB call to eliminate the changes that the program has made since the last sync point. All data base positioning is lost.

Programmer Response: Reprocess the transaction.

FC

Explanation: The program issued a call that is not valid for the segment type.

Programmer Response: Correct the call.

FD

Explanation: A nonmessage-driven program reached a deadlock when IMS/VS tried to get additional resources (either DEDB UOWs or overflow latches) for the program. IMS/VS eliminates all data base updates that the program has made since the last SYNC or CHKP call (or since the program started processing, if the program hasn't issued a SYNC or CHKP call). All data base positioning is lost.

Programmer Response: Start processing from the last sync point. If you reach a deadlock again (and you usually won't) terminate the program.

FE

Explanation: IMS/VS returns this status code anytime a program issues a FLD call that receives a nonblank status code in the FSA.

Programmer Response: See "Fast Path Data Areas" for an explanation of FSA status codes and correct the FLD call.

FF

Explanation: A program issued an ISRT (add) call against an MSDB that has no free space. If IMS/VS determines that there's no free space when the program issues the ISRT call, the program receives the FF status code for that call. IMS/VS may not determine this until the program reaches the next sync point. In this case, IMS/VS returns FF when the program issues a GU call to the message queue, a SYNC call, or a CHKP call, depending on which call caused the sync point.

Programmer Response: To avoid this situation, you can specify more space for the MSDB at the next system start (cold start or normal restart).

FG

Explanation: FG is a combination of FE and FW. A nonmessage-driven program issued a FLD call that received a nonblank status code in the FSA, and the program has used up its normal buffer allocation.

Programmer Response: Check the FSA status code and correct the FLD call, then issue SYNC or CHKP calls in the program more frequently. One way to handle this status code is to branch to an error routine that causes the program to issue SYNC or CHKP calls more frequently when it receives this status code.

FH

Explanation: A DEDB or a DEDB area was inacessible when the program issued a data base call or when the program reached a sync point. If IMS/VS returns this status code on a call that caused a sync point to occur (a SYNC call, a message GU, or a CHKP call), IMS/VS issues an internal ROLB call to eliminate the program's data base updates since the last sync point.

Programmer Response: If you receive this status code after a call that caused a sync point to occur (a GU call to the message queue, a SYNC call, or a CHKP call, depending on your program), reprocess from the last sync point to see if the condition exists when the program issues data base calls.

FI

Explanation: The program's I/O area is not at a storage address that the program can access.

Programmer Response: Correct the program.

FN

Explanation: The program issued a FLD call that contains a field name in the FSA that's not defined in the DBD. IMS/VS doesn't continue processing the FLD call or any of the FSAs in the FLD call. IMS/VS returns an FN status code in this situation even if an earlier FSA in the same FLD call earned an FE status code.

Programmer Response: Issue a ROLB call to remove the effects of the incorrect FLD call and correct the FLD call.

FP

Explanation: The I/O area referenced by a REPL, ISRT or FLD/CHANGE call to an MSDB contains an invalid packed decimal or hexadecimal field.

Programmer Response: Correct the data in the I/O area.

FR

Explanation: A nonmessage-driven program issued a data base call that forced the system to go beyond the buffer limit specified for the region. IMS/VS eliminates all data base changes made by the program since the last SYNC or CHKP call the program issued (or since the program started processing if the program hasn't issued any SYNC or CHKP calls). All data base positioning is lost.

Programmer Response: Either terminate the program and restart it with a larger buffer allocation, or provide an error-handling routine that will cause the program to issue SYNC or CHKP calls more frequently. Issuing SYNC or CHKP calls more frequently reduces the total buffer requirements.

FS

Explanation: A nonmessage-driven program issued an ISRT call for either a root or sequential dependent segment, but IMS/VS could not get enough space in either the root addressable or sequential dependent part of the DEDB area to insert the new segment. If IMS/VS returns this status code on an ISRT call for a root segment, the problem is with the root addressable portion of the area. If IMS/VS returns this status code when the program issues a SYNC or CHKP call, the problem is with the sequential dependent part of the area. In either case, IMS/VS eliminates all of the data base changes the program has made since the last sync point (or since the program started processing, if the program hasn't reached a sync point).

Programmer Response: Terminate the program.

FT

Explanation: The program issued a call to a Fast Path data base tht included more than one SSA. Only one SSA is allowed in any call to a Fast Path data base.

Programmer Response: Correct the call.

FV

Explanation: At least one of the verify operations in a FLD call issued in a nonmessage-driven program failed when the program reached a sync point. IMS/VS eliminates the data base updates the the program has made since it issued the last SYNC or CHKP call (or if the program hasn't issued a SYNC or CHKP call, since the program started processing). All data base positioning is lost.

Programmer Response: Reprocess the transaction or terminate the program.

FW

Explanation: A nonmessage-driven Fast Path program has used all buffers that are allocatd for normal usage. IMS/VS returns this status code to warn you that you may be running out of buffer space. An FR status code may be imminent.

Programmer Response: One solution to this problem is to supply an error-handling routine, triggered by the FW status code, that will cause your program to issue SYNC or CHKP calls more frequently. This will reduce the total buffer requirement.

GA

Explanation: In trying to satisfy an unqualified GN or GNP, IMS/VS crossed a hierarchic boundary into a higher level.

If IMS/VS returns GA after a STAT call, it means that the STAT call just issued retrieved the statistics for the last VSAM buffer subpool. These statistics are for the largest VSAM buffer subpool. If you issue the same STAT call again, IMS/VS returns the total statistics for all of the VSAM buffer subpools.

Programmer Response: The status code is an information-only status code. What you do next depends on your program.

GB

Explanation: In trying to satisfy a GN call, DL/I reached the end of the data base. In this situation, the SSA specified data beyond the last occurrence, and the search was not limited to the presence of a known or expected segment occurrence. For example, a GN call for a key greater than a particular value, rather than a GU specifying a key value veyond the highest value.

IMS/VS also returns this status code when it has closed a GSAM data set. The assumed position for a subsequent call for a GSAM or DL/I data base is the beginning of the data base.

Programmer Response: User determined.

GC

Explanation: An attempt was made to cross a Unit-of-Work (UOW) boundary. There was at least one call on the referenced PCB that changed position in the data base since the last sync point or after the program began executing. IMS/VS doesn't retrieve or insert a segment. Positioning is for the first segment following the current UOW boundary.

Programmer Response: User determined.

GD

Explanation: The program issued an ISRT call that did not have SSAs for all levels above the level of the segment being inserted. For at least one of the levels for which no SSA was specified, a prior call using this PCB established valid position on a segment. That position is no longer valid for one of these reasons:

- The segment has been deleted by a DLET call using a different DB PCB.
- The segment was retrieved using an alternate processing sequence, and a REPL or DLET call for this DB PCB caused the index for the existing position to be deleted.

Programmer Response: This is an information-only status code.

GE

Explanation: IMS/VS returns this status code when:

- DL/I is unable to find a segment that satisfies the segment described in a get call.
- For an ISRT call, DL/I can't find one of the parents of the segment you're inserting.

- The program issued a STAT call for ISAM/OSAM buffer pool statistics when the buffer pool doesn't exist.
- The program issued a STAT call for VSAM buffer subpool statistics when the subpools don't exist.
- The program issued a STAT call that specified a statistics function for ISAM/OSAM buffer pool statistics.

Programmer Response: The action you take depends on your program.

Note: In Fast Path application programs, if, in executing a GNP call, IMS/VS tries to retrieve a deleted sequential dependent segment, IMS/VS returns a GE status code. The I/O area will contain a length indication of 10 bytes and the original position of the delted segment.

GG

Explanation: IMS/VS returns this status code only to application programs with processing options of GOT or GON, after the program has issued one of the get calls. It means that the segment the program was trying to retrieve contained an invalid pointer. Position in the data base after a GG status code is just before the first root segment occurrence in the hierarchy. The PCB key feedback area will contain the length of the key of the last root segment accessed.

Programmer Response: Continue processing with another segment or terminate the program. It's possible that the call you received the GG status code on may be successful if you issue it again.

GK

Explanation: DL/I has returned a different segment type at the same hierarchic level for an unqualified GN or GNP.

Programmer Response: This is an information-only status code.

GL

Explanation: The program issued a LOG call that contained an invalid log call for user log records. The log code in a LOG call must be greater than X'A0'.

DL/I returns GL on a DEQ call when the first byte of the I/O area referenced in the call did not contain a valid DEQ class (A-J).

Programmer Response: If the program received this status code for a LOG call, check the log code in the call and correct it. If the program received this status code for a DEQ call, check the DEQ class code in the I/O area.

GP

Explanation: The program issued a GNP call when there is no parentage established, or the segment level specified in the GNP is not lower than the level of the established parent.

IMS/VS also returns this status code in Fast Path application programs when the program issues a GNP call that names a root segment.

Programmer Response: Check the GNP call and issue a call before the GNP to correctly establish parentage.

II

Explanation: The program issued an ISRT call that tried to insert a segment that already exists in the data base. Some of the reasons for receiving this status code are:

- A segment with an equal physical twin sequence field already exists for the parent.
- A segment with an equal logical twin sequence already exists for the parent.
- The logical parent has a logical child pointer, the logical child doesn't have a logical twin pointer, and the segment being inserted is the second logical child for that logical parent.
- The segment type doesn't have physical twin forward pointers and the segment being inserted is the second segment of this type for that parent, or it's the second HDAM root for one anchor point.
- The segment being inserted is in an inverted structure. (The immediate parent of this segment in the logical structure is actually its physical child in the physical structure.)
- A physically-paired logical child segment already exists with a sequence field equal to that of the segment you're inserting. For example, the segment could have been inserted with no duplication but when an attempt was made to position for the insert of its physical pair, it was found to have a duplicate key to an existing twin segment.
- In Fast Path application programs, IMS/VS returns this status code only when an attempt is made to insert

duplicate key segments in a DEDB (root segments only) or an MSDB.

Programmer Response: User determined.

IX

Explanation: The program issued an ISRT call that violated the insert rule for that segment. Some of the reasons that IMS/VS returns this status code are:

- The program tried to insert the logical child and logical parent, and the insert rule for the logical parent is physical and the logical parent does not exist.
- The program tried to insert the logical child and the logical parent and the insert rule is logical or virtual and the logical parent doesn't exist. In the I/O area, the key of the logical parent doesn't match the corresponding key in the concatenated key in the logical child.
- The program tried to insert a logical child, and the insert rule of the logical parent is virtual and the logical parent exists. In the I/O area, the key in the logical parent segment doesn't match the corresponding key in the concatenated key in the logical child.
- The program tried to insert a physically paired segment, where both sides of the physical pair are the same segment type and the physical and logical parent are the same occurrence.
- The program issued an ISRT call after an open, close, or I/O error status code.
- The program issued an ISRT call to a GSAM data base after receiving an AI or AO status code.

Programmer Response: Correct the ISRT call, or the program.

LB

Explanation: The segment that the program tried to load already exists in the data base. Other possible causes are:

- A segment with an equal physical twin sequence field already exists for the parent.
- A segment type doesn't have a physical twin forward pointer, and the segment being inserted is either the second segment of this segment type for the parent or the second HDAM root for one anchor point.
- An application program inserted a key of X'FF...FF' into a HISAM or HIDAM data base.

Programmer Response: Correct the ISRT call or find out if the load sequence is incorrect. Check with the DBA or the equivalent specialist at your installation.

LC

Explanation: The key field of the segment being inserted is out of sequence.

Programmer Response: Check the segment and determine where it should be loaded.

LD

Explanation: No parent has been loaded for the segment being inserted.

Programmer Response: Check the sequence of segments that have been loaded and determine where the parent should have been loaded.

LE

Explanation: The sequence of sibling segments being loaded is not the same as the sequence that's defined in the DBD.

Programmer Response: Check the sequence of the segments that are being loaded and correct.

Nb (N blank)

Explanation: Index maintenance is unable to handle the status code it received from the buffer handler. This status code will be included in message DFS0840I on the system console. DFS0840I gives the message "INDEX ERROR dbdname Nb (first 45 bytes of key)." The buffer handler usually returns messages giving specific information about the problem before IMS/VS issues message DFS0840I. If possible, IMS/VS continues processing; if not, IMS/VS terminates your program abnormally with a user abend code of 825.

Programmer Response: Review the status of the index to determine whether or not it should be rebuilt.

NE

Explanation: Indexing maintenance issued a DL/I call, and the segment has not been found. This status code will be included in message DFS0840I on the system console. DFS0840I gives the message

"INDEX ERROR (dbdname) NE (first 45 bytes of key)."

Programmer Response: Review the status of the index to determine whether or not it should be rebuilt.

NI

Explanation: There is a duplicate segment in a unique secondary index. While IMS/VS was inserting a replacing a source segment for a secondary index defined with a unique sequence field, the insertion of the segment was attempted but was unsuccessful because an index segment with the same key was found. One possible cause for a duplicate segment in the index is that the index DBD incorrectly specified a unique key value—secondary index only.

In an online application program, the call is backed out and the program receives an NI status code.

In a batch program, IMS/VS terminates the program abnormally with a code of 828.

Programmer Response: In a batch program, you should run batch backout to remove the effects of the inaccurate processing, since the ISRT call was partially completed when the 828 abnormal termination occurred. If duplicate secondary index entries occur, the index should be specified as nonunique, and an overflow entry-sequenced data set should be provided.

NO

Explanation: There was a BSAM, ISAM, VSAM, or OSAM physical I/O error during a data base call issued by indexing maintenance.

Programmer Response: Check the call and correct it.

QC

Explanation: An MPP or transaction-oriented BMP issued a successful CHKP call, but the message GU call issued internally by the CHKP call was unsuccessful. There are no more messages in the queue for the program.

Programmer Response: This is an information-only status code.

QD

Explanation: The program issued a message GN, but there are no more segments for this message.

Programmer Response: Process the message.

QE

Explanation: The program issued a message GN call before issuing a GU to the message queue. In message-driven Fast Path programs, this code applies to message calls only. This code also applies to GCMD calls in AOI programs. It means that the program issued a GCMD call before issuing a CMD call. This call is also returned when a program issues a ROLB without having issuing a successful message GU call during that sync interval.

Programmer Response: Correct the program by either:

- Issuing a GU call before the GN
- Issuing a CMD call before the GCMD
- Issuing a GU call before the ROLB

QF

Explanation: The length of the segment is less than 5 characters. The minimum length allowed is the length of the message text plus four control characters.

Programmer Response: Correct the segment.

QH

Explanation: There has been a terminal symbolic error. The output logical terminal name or transaction code is unknown to IMS/VS.

Programmer Response: Check the logical terminal name or transaction code and correct it.

RX

Explanation: The program issued a REPL call that violated the replace rule for that segment.

Programmer Response: Correct the call, or check with the DBA or the equivalent specialist at your installation.

UC

Explanation: A checkpoint record was written to the UCF journal data set. During the processing of an HD reorganization or reload or an initial load program under the supervision of the Utility Control Facility (UCF), a checkpoint record was written to the UCF journal data set. IMS/VS returns this

status code to indicate that the last ISRT call was correct and the initial load program may continue or it may perform a checkpointing procedure before continuing.

Programmer Response: This is an information-only status code.

UR

Explanation: Your initial load program is being restarted under UCF. The program terminated while executing under UCF. The job was resubmitted with a restart request.

Programmer Response: The program has to get itself back in step with data base loading. The program uses the I/O area and the DB PCB key feedback area to do this.

US

Explanation: The initial load program is about to stop processing. While processing an HD reorganization reload or user initial load program under the supervision of UCF, the operator replied to the WTOR from UCB and requested the current function to terminate. The last ISRT call was processed.

Programmer Response: The intial load program should checkpoint its data sets and return with a nonzero value in register 15.

UX

Explanation: A checkpoint record was written and processing stopped. This is a combination of UC and US status codes.

Programmer Response: See the descriptions of UC and US status codes.

VI

Explanation: An invalid length was supplied for a variable-length segment. The LL field of the variable-length segment is either too large or too small. The length of the segment must be equal to or less than the maximum length specified in the DBD. The length must be long enough to include the entire reference field; if the segment is a logical child, it must include the entire concatenated key of the logical parent and all sequence fields for the paired segment.

IMS/VS also returns this status code when an invalid record length is specified in a GSAM call.

Programmer Response: Correct the program.

XA

Explanation: The program tried to continue processing the conversation by passing the SPA to another program through a program-to-program message switch after already responding to the terminal.

Programmer Response: If a response has been sent, the SPA should be returned to IMS/VS. Correct the program.

XB

Explanation: The program has passed the SPA to another program but is trying to respond to the originating terminal.

Programmer Response: No response is allowed by a program that's passed control of the program through a program-to-program message switch.

XC

Explanation: The program inserted a message that has some bits in the Z1 field set. The Z1 field is reserved for IMS/VS.

Programmer Response: Correct the program to prevent it from setting those bits.

XD

Explanation: IMS/VS is terminating by a CHECKPOINT FREEZE or DUMPQ. IMS/VS returns this code to a BMP that has issued a CHKP call. If it's a transaction-oriented BMP, IMS/VS does not return a message.

IMS/VS also returns XD when a batch program issues a SYNC call.

Programmer Response: Terminate the program immediately. IMS/VS will terminate the program abnormally if the program issues another call.

XE

Explanation: A program tried to insert a SPA to an alternate express PCB.

Programmer Response: Regenerate the PSB and remove the EXPRESS=YES option from the PCB, or define another PCB that is not express to be used in the ISRT call.

XF

Explanation: IMS/VS is ignoring the ISRT call for the SPA because the referenced

alternate PCB had its destination set to a logical terminal but was not defined as ALTRESP=YES during PSB generation.

Programmer Response: Correct the application program or change the PSB generation for that alternate PCB to specify ALTRESP=YES.

XG

Explanation: IMS/VS ignored the ISRT call because the current conversation requires fixed-length SPAs and the ISRT call was to a program with a different length or variable-length SPA.

Programmer Response: Correct the program or the SPA definitions.

XX

Explanation: After initialization the XX status code indicates an IMS/VS error, probably with GSAM. An XX status code at initialization itself (before the program has issued its first call) may be a system, IMS/VS, or user error.

When the XX status code is issued from initialization, possible causes are:

- Insufficient storage
- Invalid DBD
- Invalid block size
- Invalid option
- GSAM error

Programmer Response: A subsequent GSAM call will result in an abnormal termination of the program. The program should terminate.

X1

Explanation: System error: an I/O error occurred while IMS/VS was reading or writing the SPA.

Programmer Response: Terminate the conversation.

X2

Explanation: The first ISRT call to a PCB whose destination is a conversational transaction code is not for the SPA. The SPA must be inserted with the first ISRT call.

Programmer Response: Insert the SPA, then reinsert the message segment.

X3

Explanation: The program modified the first 6 bytes of the SPA; the SPA is now invalid.

Programmer Response: Correct the program and restore the original bytes.

X4

Explanation: The program issued an ISRT call to pass the SPA to a nonconversational transaction code. It did this by referencing a PCB whose destination was set for the nonconversational transaction code. You can send the SPA only to transaction codes defined as conversational.

Programmer Response: Correct the ISRT call. Send only data segments.

X5

Explanation: The program issued more than one ISRT call to send the SPA to a PCB whose destination is a transaction code.

Programmer Response: Only one SPA is allowed per message. Correct the program.

X6

Explanation: An invalid transaction code name was inserted into the SPA.

Programmer Response: Correct the program to set the proper transaction code name.

X7

Explanation: The length of the SPA is incorrect. The program modified the first 6 bytes.

Programmer Response: Correct the SPA and the program.

X8

Explanation: There was a system or I/O error in attempting to queue a SPA on a transaction code PCB.

Programmer Response: Terminate the conversation.

X9

Explanation: The program tried to insert the SPA, but the length of the SPA is greater than the maximum I/O area size specified in the program's PSB.

Programmer Response: Correct the SPA, or change the I/O area's size specified on the IOASIZE keyword on the PSBGEN statement.

blanks (bb)

Explanation: The call was completed.

Programmer Response: Proceed with processing.

APPENDIXES

The appendixes provide four sample application programs, a sample status code error routine, and the DL/I Test Program control statements format:

- Appendix A: Sample Batch Program
- Appendix B: Sample Batch Message Program
- Appendix C: Sample Message Processing Program
- Appendix D: Sample Conversational MPP
- Appendix E: Sample Status Code Error Routine
- Appendix F: Using the DL/I Test Program (DFSDDLT0)

The purpose of providing the sample programs is to illustrate the structure of different IMS/VS application programs. The application programming in the programs has been kept to a minimum, and the processing performed is trivial in nature.

Each of the sample programs accesses the Parts data base described in the IMS/VS Version 1 Primer. The sample programs in Appendixes A, B, and D perform the same processing: each program updates the unit price field in the root segment.

The status code error routine is shown in "Appendix E. Sample Status Code Error Routine (DFSOAER)." This routine is also part of the Primer. "Checking Status Codes" describes this routine. Each of the sample routines uses this routine as its error routine.

APPENDIX A. SAMPLE BATCH PROGRAM

The sample batch program reads its input from a GSAM file. The GSAM input record contains a part number and the new price for that part number. The program updates the data base of the new price.

After updating the data base, the program lists the part number and the old and new prices in a GSAM output file.

If the part number is not valid/not a valid key, IMS/VS prints an error message.

When the program has processed all of the input records/transactions, the program prints a totals line giving the total numbers of valid and invalid transactions.

The program uses symbolic checkpoint and restart.

```
IDENTIFICATION DIVISION.                                        EXA00110
PROGRAM-ID. 'SAMPLE1'.                                          EXA00120
REMARKS.                                                        EXA00140
    THIS IS A BATCH PROGRAM WHICH UPDATES THE                   EXA00150
    PRICE FIELD IN THE ROOT SEGMENT OF THE PARTS                EXA00160
    DATA BASE.                                                  EXA00170
ENVIRONMENT DIVISION.                                           EXA00180
DATA DIVISION.                                                  EXA00190
WORKING-STORAGE SECTION.                                        EXA00200
                                                                EXA00210
* DL/I FUNCTION CODES                                           EXA00220
                                                                EXA00230
77  GHU    PIC X(4) VALUE 'GHU '.                               EXA00240
77  GU     PIC X(4) VALUE 'GU  '.                               EXA00250
77  GN     PIC X(4) VALUE 'GN  '.                               EXA00260
77  ISRT   PIC X(4) VALUE 'ISRT'.                               EXA00270
77  REPL   PIC X(4) VALUE 'REPL'.                               EXA00280
77  XRST   PIC X(4) VALUE 'XRST'.                               EXA00290
77  CHKP   PIC X(4) VALUE 'CHKP'.                               EXA00300
                                                                EXA00310
* PARAMETER FIELDS FOR DFSOAER STATUS CODE CHECKING ROUTINE     EXA00320
                                                                EXA00330
01  DFSOAER-FIELDS.                                            EXA00340
    02  ERROPT      PIC X(4) VALUE '1   '.                     EXA00350
    02  BAD-DB-CALL PIC X(8) VALUE 'DBADCALL'.                 EXA00360
    02  BAD-DC-CALL PIC X(8) VALUE 'CBADCALL'.                 EXA00370
01  CHKPT-WORKAREAS.                                           EXA00380
                                                                EXA00390
* RESTART CHECKPOINT-ID RETURNED HERE IF PROGRAM RESTARTED      EXA00400
                                                                EXA00410
    02  RESTART-WORKAREA.                                      EXA00420
        04  RESTART-CHKPT PIC X(8) VALUE SPACES.               EXA00430
        04  FILLER        PIC X(4) VALUE SPACES.               EXA00440
                                                                EXA00450
* CHECKPOINT-ID INCREMENTED BY ONE AT EACH CHECKPOINT           EXA00460
                                                                EXA00470
    02  CHKPT-ID.                                              EXA00480
        04  FILLER        PIC X(4) VALUE 'SAM1'.               EXA00490
        04  CHKPT-ID-CTR  PIC 9(4) VALUE 0.                    EXA00500
                                                                EXA00510
*  TRANSACTION COUNTER USED TO DETERMINE CHECKPOINT FREQUENCY   EXA00520
                                                                EXA00530
    02  CHKPT-LIMIT       PIC S9(5) COMP-3 VALUE +0.           EXA00540
        88  CHKPT-LIMIT-REACHED            VALUE +50.          EXA00550
                                                                EXA00560
*  LENGTH FIELDS USED FOR XRST AND CHKP CALLS                   EXA00570
                                                                EXA00580
01  AREA-LENGTHS.                                              EXA00590
    02  IOAREA-LEN   PIC S9(5) COMP VALUE +80.                 EXA00600
    02  COUNTER-LEN  PIC S9(5) COMP VALUE +8.                  EXA00610
                                                                EXA00620
01  COUNTERS.                                                  EXA00630
    02  LINE-CTR     PIC S9(3)  COMP-3 VALUE +50.              EXA00640
        88  TOP-OF-PAGE                  VALUE +50.            EXA00650
    02  VALID-CTR    PIC S9(5) COMP-3 VALUE +0.                EXA00660
    02  INVALID-CTR  PIC S9(5) COMP-3 VALUE +0.                EXA00670
                                                                EXA00680
* END-SWITCH SET TO 1 IF GB REACHED ON INPUT GSAM FILE          EXA00690
                                                                EXA00700
01  END-SWITCH       PIC X VALUE '0'.                          EXA00710
    88  NO-MORE            VALUE '1'.                          EXA00720
                                                                EXA00730
01  INPUT-AREA.                                                EXA00740
    02  TRANCODE     PIC X(9).                                 EXA00750
    02  IN-PARTNO    PIC X(8).                                 EXA00760
    02  NEW-PRICE    PIC 9(6)V99.                              EXA00770
    02  FILLER       PIC X(100).                               EXA00780
                                                                EXA00790
01  OUTPUT-AREAS.                                              EXA00800
    02  OUTPUT-LINE.                                           EXA00810
        04  OUTPUT-ASA  PIC X.                                 EXA00820
        04  OUTPUT-DATA PIC X(80).                             EXA00830
    02  HEADING-LINE.                                          EXA00840
```

```
           04  FILLER PIC X(10) VALUE 'PART NO'.                    EXA00850
           04  FILLER PIC X(11) VALUE 'OLD PRICE'.                  EXA00860
           04  FILLER PIC X(11) VALUE 'NEW PRICE'.                  EXA00870
           04  FILLER PIC X(49) VALUE 'COMMENTS'.                   EXA00880
       02  DETAIL-LINE.                                             EXA00890
           04  OUT-PARTNO      PIC X(8).                            EXA00900
           04  FILLER          PIC X.                               EXA00910
           04  OUT-OLD-PRICE   PIC Z(6)9.99.                        EXA00920
           04  FILLER          PIC X.                               EXA00930
           04  OUT-NEW-PRICE   PIC Z(6)9.99.                        EXA00940
           04  FILLER          PIC XX.                              EXA00950
           04  COMMENTS        PIC X(40).                           EXA00960
       02  TOTAL-LINE.                                              EXA00970
           04 FILLER           PIC X(31) VALUE                      EXA00980
              'TRANSACTIONS PROCESSED - VALID '.                     EXA00990
           04  OUT-VALID       PIC Z(4)9.                           EXA01000
           04  FILLER          PIC X(10) VALUE ' INVALID '.         EXA01010
           04  OUT-INVALID     PIC Z(4)9.                           EXA01020
                                                                     EXA01030
   *  INPUT AREA FOR DATA BASE SEGMENT                              EXA01040
                                                                     EXA01050
   01  DB-IOAREA.                                                   EXA01060
       02  DB-PARTNO       PIC X(8).                                EXA01070
       02  FILLER          PIC X(45).                               EXA01080
       02  DB-PRICE        PIC 9(6)V99.                             EXA01090
       02  FILLER          PIC X(19).                               EXA01100
                                                                     EXA01110
   *  SEGMENT SEARCH ARGUMENT                                       EXA01120
                                                                     EXA01130
   01  SSA.                                                         EXA01140
       02  FILLER          PIC X(19) VALUE 'SE1PART (FE1PGPNR ='.   EXA01150
       02  SSA-PARTNO      PIC X(8).                                EXA01160
       02  FILLER          PIC X     VALUE ')'.                     EXA01170
                                                                     EXA01180
   01  ASA-CTL-CHARS.                                               EXA01190
       02  ASA-NEWPAGE     PIC X VALUE '1'.                         EXA01200
       02  ASA-SPACE-ONE   PIC X VALUE ' '.                         EXA01210
       02  ASA-SPACE-TWO   PIC X VALUE '0'.                         EXA01220
                                                                     EXA01230
   LINKAGE SECTION.                                                  EXA01240
                                                                     EXA01250
   * IOPCB USED FOR XRST AND CHECKPOINT CALLS                       EXA01260
                                                                     EXA01270
   01  IOPCB.                                                       EXA01280
       02  FILLER          PIC X(10).                               EXA01290
       02  TPSTATUS        PIC X(2).                                EXA01300
       02  FILLER          PIC X(20).                               EXA01310
                                                                     EXA01320
   * DATA BASE PCB FOR THE PARTS DATA BASE                          EXA01330
                                                                     EXA01340
   01  DBPCB.                                                       EXA01350
       02  FILLER          PIC X(10).                               EXA01360
       02  DBSTATUS        PIC X(2).                                EXA01370
       02  FILLER          PIC X(20).                               EXA01380
                                                                     EXA01390
   * GSAM INPUT PCB FOR THE INPUT DATA                              EXA01400
                                                                     EXA01410
   01  GSAMPCB-IN.                                                  EXA01420
       02  FILLER          PIC X(10).                               EXA01430
       02  GSTATUS-IN      PIC X(2).                                EXA01440
       02  FILLER          PIC X(20).                               EXA01450
                                                                     EXA01460
   * GSAM OUTPUT PCB FOR THE OUTPUT REPORT                          EXA01470
                                                                     EXA01480
   01  GSAMPCB-OUT.                                                 EXA01490
       02  FILLER          PIC X(10).                               EXA01500
       02  GSTATUS-OUT     PIC X(2).                                EXA01510
       02  FILLER          PIC X(20).                               EXA01520
                                                                     EXA01530
   PROCEDURE DIVISION.                                               EXA01540
                                                                     EXA01550
   * AT ENTRY IMS PASSES THE FOLLOWING PCB ADDRESSES:               EXA01560
                                                                     EXA01570
```

```
*      IOPCB       - USED FOR CHECKPOINT/RESTART CALLS                    EXA01580
*      DBPCB       - PARTS DATA BASE                                      EXA01590
*      GSAMPCB-IN  - INPUT DATA FILE                                      EXA01600
*      GSAMPCB-OUT - OUTPUT REPORT FILE                                   EXA01610
                                                                          EXA01620
     ENTRY 'DLITCBL' USING IOPCB,  DBPCB,                                 EXA01630
          GSAMPCB-IN, GSAMPCB-OUT.                                        EXA01640
                                                                          EXA01650
*  FIRST CALL IS THE XRST CALL                                            EXA01660
                                                                          EXA01670
     CALL 'CBLTDLI' USING XRST, IOPCB, IOAREA-LEN,                        EXA01680
          RESTART-WORKAREA, COUNTER-LEN, COUNTERS.                        EXA01690
     IF TPSTATUS NOT EQUAL SPACES                                         EXA01700
        THEN CALL 'DFSOAER' USING                                         EXA01710
             IOPCB, BAD-DC-CALL, COUNTERS, ERROPT.                        EXA01720
                                                                          EXA01730
* IF RESTART WORKAREA IS NOT BLANK, THEN PROGRAM IS BEING                 EXA01740
* RESTARTED - SO RESET THE CHECKPOINT-ID FIELD                            EXA01750
                                                                          EXA01760
     IF RESTART-WORKAREA NOT EQUAL SPACES                                 EXA01770
         MOVE RESTART-CHKPT TO CHKPT-ID                                   EXA01780
                                                                          EXA01790
* OTHERWISE TAKE A CHECKPOINT SO THAT PROGRAM CAN BE                      EXA01800
* COMPLETELY BACKED OUT TO THE BEGINNING IF NECESSARY                     EXA01810
                                                                          EXA01820
         ELSE PERFORM CHKPT-RTN.                                          EXA01830
                                                                          EXA01840
* MAIN LINE                                                              EXA01850
                                                                          EXA01860
     PERFORM READ-INPUT THRU READ-INPUT-END.                              EXA01870
     PERFORM PROCESS-INPUT THRU PROCESS-INPUT-END UNTIL NO-MORE.          EXA01880
     PERFORM PRINT-TOTAL-LINE THRU PRINT-TOTAL-LINE-END                   EXA01890
     GOBACK.                                                              EXA01900
                                                                          EXA01910
PROCESS-INPUT.                                                            EXA01920
                                                                          EXA01930
* HERE WE PROCESS THE INPUT MESSAGES                                      EXA01940
* USING THE PARTNUMBER WE READ THE DATABASE                               EXA01950
* AND UPDATE THE PRICE FIELD WITH THE NEW DATA                            EXA01960
                                                                          EXA01970
     MOVE SPACES TO DETAIL-LINE.                                          EXA01980
     MOVE IN-PARTNO TO SSA-PARTNO.                                        EXA01990
     PERFORM READ-DB THRU READ-DB-END.                                    EXA02000
     IF DBSTATUS = 'GE'                                                   EXA02010
       THEN MOVE 'NOT ON FILE' TO COMMENTS                                EXA02020
            MOVE IN-PARTNO TO OUT-PARTNO                                  EXA02030
            ADD 1 TO INVALID-CTR                                          EXA02040
       ELSE MOVE DB-PRICE TO OUT-OLD-PRICE                                EXA02050
            MOVE  NEW-PRICE TO DB-PRICE, OUT-NEW-PRICE                    EXA02060
            MOVE DB-PARTNO TO OUT-PARTNO                                  EXA02070
            PERFORM UPDATE-DB THRU UPDATE-DB-END                          EXA02080
            MOVE 'PRICE UPDATED' TO COMMENTS                              EXA02090
            ADD 1 TO VALID-CTR.                                           EXA02100
                                                                          EXA02110
     PERFORM PRINT-LINE THRU PRINT-LINE-END                               EXA02120
                                                                          EXA02130
* INCREMENT THE CHKPT COUNTER BY ONE FOR EACH TRANSACTION                 EXA02140
                                                                          EXA02150
    ADD 1 TO CHKPT-LIMIT.                                                 EXA02160
                                                                          EXA02170
     IF CHKPT-LIMIT-REACHED                                               EXA02180
                                                                          EXA02190
* INCREMENT THE CHECKPOINT-ID COUNTER                                     EXA02200
* ISSUE A CHECKPOINT CALL                                                 EXA02210
* AND RESET THE CHECKPOINT FREQUENCY COUNTER                              EXA02220
                                                                          EXA02230
       THEN ADD 1 TO CHKPT-ID-CTR                                         EXA02240
            PERFORM CHKPT-RTN THRU CHKPT-RTN-END                          EXA02250
            MOVE 0 TO CHKPT-LIMIT.                                        EXA02260
                                                                          EXA02270
* READ THE NEXT MESSAGE                                                  EXA02280
                                                                          EXA02290
     PERFORM READ-INPUT THRU READ-INPUT-END.                              EXA02300
```

```
                                                                        EXA02310
PROCESS-INPUT-END.                                                      EXA02320
    EXIT.                                                               EXA02330
                                                                        EXA02340
*  PRINT THE REPORT SHOWING THE UPDATES                                 EXA02350
*  THE LINES ARE WRITTEN TO A GSAM FILE WHICH                           EXA02360
*  CAN BE SPOOLED TO A PRINTER IN A SUBSEQUENT                          EXA02370
*  JOB STEP                                                             EXA02380
                                                                        EXA02390
PRINT-LINE.                                                             EXA02400
                                                                        EXA02410
* IF PAGE IS FULL, PRINT A HEADING LINE AND RESET THE                   EXA02420
* LINE-COUNTER BEFORE PRINTING THE DETAIL LINE                          EXA02430
                                                                        EXA02440
    IF TOP-OF-PAGE                                                      EXA02450
       THEN MOVE HEADING-LINE TO OUTPUT-DATA                            EXA02460
            MOVE ASA-NEWPAGE TO OUTPUT-ASA                              EXA02470
            MOVE 0 TO LINE-CTR                                          EXA02480
            PERFORM ISRT-GSAM-OUTPUT THRU ISRT-GSAM-OUTPUT-END.         EXA02490
            MOVE ASA-SPACE-TWO TO OUTPUT-ASA.                           EXA02500
                                                                        EXA02510
    MOVE DETAIL-LINE TO OUTPUT-DATA                                     EXA02520
    PERFORM ISRT-GSAM-OUTPUT THRU ISRT-GSAM-OUTPUT-END.                 EXA02530
                                                                        EXA02540
* INCREMENT LINE COUNTER. IF FIRST DETAIL LINE ON PAGE HAS BEEN         EXA02550
* PRINTED RESET THE ASA CONTROL CHARACTER TO SINGLE SPACING             EXA02560
                                                                        EXA02570
    ADD 1 TO LINE-CTR                                                   EXA02580
    IF LINE-CTR = 1 MOVE ASA-SPACE-ONE TO OUTPUT-ASA.                   EXA02590
                                                                        EXA02600
PRINT-LINE-END.                                                         EXA02610
    EXIT.                                                               EXA02620
                                                                        EXA02630
PRINT-TOTAL-LINE.                                                       EXA02640
    MOVE VALID-CTR TO OUT-VALID.                                        EXA02650
    MOVE INVALID-CTR TO OUT-INVALID.                                    EXA02660
    MOVE TOTAL-LINE TO OUTPUT-DATA.                                     EXA02670
    MOVE ASA-NEWPAGE TO OUTPUT-ASA                                      EXA02680
    PERFORM ISRT-GSAM-OUTPUT THRU ISRT-GSAM-OUTPUT-END.                 EXA02690
    MOVE SPACES TO OUTPUT-ASA.                                          EXA02700
                                                                        EXA02710
PRINT-TOTAL-LINE-END.                                                   EXA02720
    EXIT.                                                               EXA02730
                                                                        EXA02740
* THE FOLLOWING PROCEDURES EXECUTE THE DL/I CALLS AGAINST               EXA02750
* THE GSAM INPUT AND OUTPUT FILES, AND THE DATA BASE.                   EXA02760
* NO APPLICATION PROCESSING IS PERFORMED IN THESE ROUTINES.             EXA02770
                                                                        EXA02780
READ-INPUT.                                                             EXA02790
    CALL 'CBLTDLI' USING GN, GSAMPCB-IN, INPUT-AREA.                    EXA02800
    IF GSTATUS-IN = 'GB'                                                EXA02810
        THEN MOVE 1 TO END-SWITCH                                       EXA02820
        ELSE IF GSTATUS-IN NOT EQUAL SPACES                             EXA02830
             THEN CALL 'DFSOAER' USING                                  EXA02840
                  GSAMPCB-IN, BAD-DB-CALL, INPUT-AREA, ERROPT.          EXA02850
READ-INPUT-END.                                                         EXA02860
    EXIT.                                                               EXA02870
                                                                        EXA02880
READ-DB.                                                                EXA02890
    CALL 'CBLTDLI' USING GHU, DBPCB, DB-IOAREA, SSA.                    EXA02900
    IF DBSTATUS = SPACES OR 'GE'                                        EXA02910
    THEN NEXT SENTENCE                                                  EXA02920
    ELSE CALL 'DFSOAER' USING                                           EXA02930
            DBPCB, BAD-DB-CALL, DB-IOAREA, ERROPT.                      EXA02940
READ-DB-END.                                                            EXA02950
    EXIT.                                                               EXA02960
                                                                        EXA02970
UPDATE-DB.                                                              EXA02980
    CALL 'CBLTDLI' USING REPL, DBPCB, DB-IOAREA.                        EXA02990
    IF DBSTATUS NOT EQUAL SPACES                                        EXA03000
    THEN CALL 'DFSOAER' USING                                           EXA03010
            DBPCB, BAD-DB-CALL, DB-IOAREA, ERROPT.                      EXA03020
UPDATE-DB-END.                                                          EXA03030
```

```
    EXIT.                                                              EXA03040
                                                                       EXA03050
ISRT-GSAM-OUTPUT.                                                      EXA03060
    CALL 'CBLTDLI' USING ISRT, GSAMPCB-OUT, OUTPUT-LINE.               EXA03070
    IF GSTATUS-OUT NOT EQUAL SPACES                                    EXA03080
    THEN CALL 'DFSOAER' USING                                          EXA03090
        GSAMPCB-OUT, BAD-DB-CALL, OUTPUT-LINE, ERROPT.                 EXA03100
ISRT-GSAM-OUTPUT-END.                                                  EXA03110
    EXIT.                                                              EXA03120
                                                                       EXA03130
CHKPT-RTN.                                                             EXA03140
    CALL 'CBLTDLI' USING CHKP, IOPCB, IOAREA-LEN, CHKPT-ID,            EXA03150
        COUNTER-LEN, COUNTERS.                                         EXA03160
    IF TPSTATUS NOT EQUAL SPACES                                       EXA03170
          CALL 'DFSOAER' USING                                         EXA03180
                IOPCB, BAD-DC-CALL, CHKPT-ID, ERROPT.                  EXA03190
CHKPT-RTN-END.                                                         EXA03200
    EXIT.                                                              EXA03210
```

APPENDIX B. SAMPLE BATCH MESSAGE PROGRAM

This sample program is a transaction-oriented BMP that updates the price field. The program gets its input from the message queue and updates the price field of the root segment. When the BMP prints the totals of the valid and invalid transactions that have been processed, it sends them to an alternate PCB. Before issuing the ISRT call to send this message, the program uses the CHNG call to set the destination of the PCB. One reason you might use an alternate PCB in this situation is to send the output to a hardcopy terminal/printer in the user's department.

This program uses symbolic checkpoint and restart.

```
IDENTIFICATION DIVISION.                                                EXA00110
PROGRAM-ID. 'SAMPLE2'.                                                  EXA00120
ENVIRONMENT DIVISION.                                                   EXA00130
DATA DIVISION.                                                          EXA00140
WORKING-STORAGE SECTION.                                                EXA00150
                                                                        EXA00160
* DL/I FUNCTION CODES                                                   EXA00170
                                                                        EXA00180
77  GHU    PIC X(4) VALUE 'GHU '.                                       EXA00190
77  GU     PIC X(4) VALUE 'GU  '.                                       EXA00200
77  GN     PIC X(4) VALUE 'GN  '.                                       EXA00210
77  ISRT   PIC X(4) VALUE 'ISRT'.                                       EXA00220
77  REPL   PIC X(4) VALUE 'REPL'.                                       EXA00230
77  XRST   PIC X(4) VALUE 'XRST'.                                       EXA00240
77  CHKP   PIC X(4) VALUE 'CHKP'.                                       EXA00250
77  CHNG   PIC X(4) VALUE 'CHNG'.                                       EXA00260
                                                                        EXA00270
* DESTINATION LTERM-NAME FOR MODIFIABLE PCB                             EXA00280
                                                                        EXA00290
77  SUPER-LTERM     PIC X(8) VALUE 'PARTSUPR'.                         EXA00300
                                                                        EXA00310
* PARAMETER FIELDS FOR USE BY DFSOAER STATUS CODE CHECKING ROUTINE      EXA00320
                                                                        EXA00330
01  DFSOAER-FIELDS.                                                    EXA00340
    02  ERROPT      PIC X(4) VALUE '1   '.                             EXA00350
    02  BAD-DB-CALL PIC X(8) VALUE 'DBADCALL'.                         EXA00360
    02  BAD-DC-CALL PIC X(8) VALUE 'CBADCALL'.                         EXA00370
                                                                        EXA00380
01  CHKPT-WORKAREAS.                                                   EXA00390
    02  RESTART-WORKAREA.                                              EXA00400
        04  RESTART-CHKPT PIC X(8) VALUE SPACES.                       EXA00410
        04  FILLER        PIC X(4) VALUE SPACES.                       EXA00420
    02  CHKPT-ID.                                                      EXA00430
        04  FILLER        PIC X(4) VALUE 'SAM2'.                       EXA00440
        04  CHKPT-ID-CTR  PIC 9(4) VALUE 0.                            EXA00450
    02  CHKPT-LIMIT       PIC S9(5) COMP-3 VALUE +0.                   EXA00460
        88  CHKPT-LIMIT-REACHED            VALUE +50.                  EXA00470
01  AREA-LENGTHS.                                                      EXA00480
    02  IOAREA-LEN        PIC S9(5) COMP VALUE +80.                    EXA00490
    02  COUNTER-LEN       PIC S9(5) COMP VALUE +8.                     EXA00500
01  COUNTERS.                                                          EXA00510
    02  LINE-CTR          PIC S9(3)  COMP-3 VALUE +50.                 EXA00520
        88  TOP-OF-PAGE                     VALUE +50.                 EXA00530
    02  VALID-CTR         PIC S9(5) COMP-3 VALUE +0.                   EXA00540
    02  INVALID-CTR       PIC S9(5) COMP-3 VALUE +0.                   EXA00550
01  SWITCHES.                                                          EXA00560
    02  END-SWITCH        PIC X VALUE '0'.                             EXA00570
        88  NO-MORE             VALUE '1'.                             EXA00580
    02  CLOSE-SWITCH          PIC X VALUE '0'.                         EXA00590
        88  CLOSE-DOWN          VALUE '1'.                             EXA00600
                                                                        EXA00610
01  INPUT-MSG.                                                         EXA00620
    02  IN-LL1      PIC S9(3) COMP.                                    EXA00630
    02  IN-ZZ1      PIC S9(3) COMP.                                    EXA00640
    02  TRANCODE    PIC X(9).                                          EXA00650
    02  IN-PARTNO   PIC X(8).                                          EXA00660
    02  NEW-PRICE   PIC 9(6)V99.                                       EXA00670
    02  FILLER      PIC X(100).                                        EXA00680
                                                                        EXA00690
01  CHKPT-AREA REDEFINES INPUT-MSG.                                    EXA00700
    02  PASS-CHKPT  PIC X(8).                                          EXA00710
    02  FILLER      PIC X(100).                                        EXA00720
                                                                        EXA00730
                                                                        EXA00740
01  OUTPUT-AREAS.                                                      EXA00750
    02  OUTPUT-MSG.                                                    EXA00760
        04  OUT-LL  PIC S9(3) COMP VALUE +85.                          EXA00770
        04  OUT-ZZ  PIC S9(3) COMP VALUE +0.                           EXA00780
        04  OUTPUT-LINE.                                               EXA00790
            06  OUTPUT-ASA    PIC X.                                   EXA00800
            06  OUTPUT-DATA   PIC X(80).                               EXA00810
    02  HEADING-LINE.                                                  EXA00820
        04  FILLER  PIC X(9)  VALUE 'PART NO'.                         EXA00830
```

```
           04  FILLER  PIC X(11) VALUE 'OLD PRICE'.                    EXA00840
           04  FILLER  PIC X(11) VALUE 'NEW PRICE'.                    EXA00850
           04  FILLER  PIC X(49) VALUE 'COMMENTS'.                     EXA00860
       02  DETAIL-LINE.                                                EXA00870
           04  OUT-PARTNO          PIC X(8).                           EXA00880
           04  FILLER              PIC X.                              EXA00890
           04  OUT-OLD-PRICE       PIC Z(6)9.99.                       EXA00900
           04  FILLER              PIC X.                              EXA00910
           04  OUT-NEW-PRICE       PIC Z(6)9.99.                       EXA00920
           04  FILLER              PIC X.                              EXA00930
           04  COMMENTS            PIC X(40).                          EXA00940
       02  TOTAL-LINE.                                                 EXA00950
           04 FILLER               PIC X(40) VALUE                     EXA00960
              'TRANSACTIONS PROCESSED - VALID '.                        EXA00970
           04  OUT-VALID           PIC Z(4)9.                          EXA00980
           04  FILLER              PIC X(10) VALUE ' INVALID '.        EXA00990
           04  OUT-INVALID         PIC Z(4)9.                          EXA01000
                                                                        EXA01010
   01  DB-IOAREA.                                                      EXA01020
       02  DB-PARTNO      PIC X(8).                                    EXA01030
       02  FILLER         PIC X(45).                                   EXA01040
       02  DB-PRICE       PIC 9(6)V99.                                 EXA01050
       02  FILLER         PIC X(19).                                   EXA01060
                                                                        EXA01070
   01  ASA-CTL-CHARS.                                                  EXA01080
       02  ASA-NEWPAGE    PIC X VALUE '1'.                             EXA01090
       02  ASA-SPACE-ONE  PIC X VALUE ' '.                             EXA01100
       02  ASA-SPACE-TWO  PIC X VALUE '0'.                             EXA01110
                                                                        EXA01120
   * SEGMENT SEARCH ARGUMENT                                           EXA01130
                                                                        EXA01140
   01  SSA.                                                            EXA01150
       02  FILLER         PIC X(19) VALUE 'SE1PART (FE1PGPNR ='.       EXA01160
       02  SSA-PARTNO     PIC X(8).                                    EXA01170
       02  FILLER         PIC X     VALUE ')'.                         EXA01180
                                                                        EXA01190
   LINKAGE SECTION.                                                     EXA01200
                                                                        EXA01210
   * IOPCB FOR RETRIEVING MESSAGES, AND ISSUING CHKP/XRST CALLS        EXA01220
                                                                        EXA01230
   01  IOPCB.                                                          EXA01240
       02  FILLER       PIC X(10).                                     EXA01250
       02  TPSTATUS     PIC X(2).                                      EXA01260
       02  FILLER       PIC X(20).                                     EXA01270
                                                                        EXA01280
   * MODIFIABLE ALTERNATE PCB USED TO SWITCH A TOTALS MESSAGE          EXA01290
   * TO A SUPERVISOR'S TERMINAL                                        EXA01300
                                                                        EXA01310
   01  ALTPCB.                                                         EXA01320
       02  ALTPCB-DEST  PIC X(8).                                      EXA01330
       02  FILLER       PIC X(2).                                      EXA01340
       02  ALTSTATUS    PIC X(2).                                      EXA01350
       02  FILLER       PIC X(10).                                     EXA01360
                                                                        EXA01370
   * DATA BASE PCB FOR THE PARTS DATA BASE                             EXA01380
                                                                        EXA01390
   01  DBPCB.                                                          EXA01400
       02  FILLER       PIC X(10).                                     EXA01410
       02  DBSTATUS     PIC X(2).                                      EXA01420
       02  FILLER       PIC X(20).                                     EXA01430
                                                                        EXA01440
   * GSAM PCB FOR THE OUTPUT REPORT FILE                               EXA01450
                                                                        EXA01460
   01  GSAMPCB-OUT.                                                    EXA01470
       02  FILLER       PIC X(10).                                     EXA01480
       02  GSTATUS-OUT  PIC X(2).                                      EXA01490
       02  FILLER       PIC X(20).                                     EXA01500
                                                                        EXA01510
   PROCEDURE DIVISION.                                                  EXA01520
                                                                        EXA01530
   * ON ENTRY IMS PASSES THE FOLLOWING PCB ADDRESSES:                  EXA01540
                                                                        EXA01550
   * IOPCB       - INPUT TRANSACTIONS FROM THE MESSAGE QUEUE           EXA01560
```

```
* ALTPCB      - LTERM FOR SWITCHING TOTALS MESSAGE                    EXA01570
* DBPCB       - PARTS DATA BASE                                       EXA01580
* GSAMPCB-OUT - OUTPUT REPORT FILE                                    EXA01590
                                                                      EXA01600
    ENTRY 'DLITCBL' USING IOPCB, ALTPCB, DBPCB, GSAMPCB-OUT.          EXA01610
                                                                      EXA01620
* FIRST CALL IS THE XRST                                             EXA01630
                                                                      EXA01640
                                                                      EXA01650
    CALL 'CBLTDLI' USING XRST, IOPCB, IOAREA-LEN,                     EXA01660
         RESTART-WORKAREA, COUNTER-LEN, COUNTERS.                     EXA01670
    IF TPSTATUS NOT EQUAL SPACES                                      EXA01680
       THEN CALL 'DFSOAER' USING                                      EXA01690
            IOPCB, BAD-DC-CALL, COUNTERS, ERROPT.                     EXA01700
                                                                      EXA01710
*   IF THE RESTART WORKAREA IS NOT BLANK, THE PROGRAM                 EXA01720
*   IS BEING RESTARTED, SO RESET THE CHECKPOINT ID FIELD              EXA01730
                                                                      EXA01740
    IF RESTART-WORKAREA NOT EQUAL SPACES                              EXA01750
        MOVE RESTART-CHKPT TO CHKPT-ID.                               EXA01760
                                                                      EXA01770
*   READ THE FIRST MESSAGE                                            EXA01780
                                                                      EXA01790
    PERFORM READ-INPUT THRU READ-INPUT-END                            EXA01800
                                                                      EXA01810
*   PROCESS THE MESSAGES UNTIL A QC STATUS CODE IS RECEIVED           EXA01820
*   OR AN XD ON A CHECKPOINT CALL WHICH INDICATES THAT THE            EXA01830
*   ONLINE SYSTEM IS BEING CLOSED DOWN                                EXA01840
                                                                      EXA01850
    PERFORM PROCESS-INPUT THRU PROCESS-INPUT-END                      EXA01860
       UNTIL NO-MORE OR CLOSE-DOWN.                                   EXA01870
                                                                      EXA01880
*   PRINT THE TOTALS LINE AND SEND A MESSAGE TO                       EXA01890
*   TO THE SUPERVISOR'S TERMINAL VIA THE ALTERNATE PCB                EXA01900
                                                                      EXA01910
    PERFORM PRINT-TOTALS THRU PRINT-TOTALS-END.                       EXA01920
    GOBACK.                                                           EXA01930
                                                                      EXA01940
PROCESS-INPUT.                                                        EXA01950
    MOVE SPACES TO DETAIL-LINE.                                       EXA01960
    MOVE IN-PARTNO TO SSA-PARTNO.                                     EXA01970
    PERFORM READ-DB THRU READ-DB-END.                                 EXA01980
    IF DBSTATUS = 'GE'                                                EXA01990
      THEN MOVE 'NOT ON FILE' TO COMMENTS                             EXA02000
           MOVE IN-PARTNO TO OUT-PARTNO                               EXA02010
           ADD 1 TO INVALID-CTR                                       EXA02020
      ELSE MOVE DB-PRICE TO OUT-OLD-PRICE                             EXA02030
           MOVE  NEW-PRICE TO DB-PRICE, OUT-NEW-PRICE                 EXA02040
           MOVE DB-PARTNO TO OUT-PARTNO                               EXA02050
           PERFORM UPDATE-DB THRU UPDATE-DB-END                       EXA02060
           MOVE 'PRICE UPDATED' TO COMMENTS                           EXA02070
           ADD 1 TO VALID-CTR.                                        EXA02080
                                                                      EXA02090
    PERFORM PRINT-LINE THRU PRINT-LINE-END                            EXA02100
    ADD 1 TO CHKPT-LIMIT.                                             EXA02110
                                                                      EXA02120
* IF THE CHECKPOINT-LIMIT HAS BEEN REACHED, TAKE                     EXA02130
* A CHECKPOINT AND INCREMENT THE ID COUNTER                          EXA02140
* THIS WILL ALSO CAUSE A MESSAGE TO BE RETURNED                      EXA02150
                                                                      EXA02160
    IF CHKPT-LIMIT-REACHED                                            EXA02170
        THEN ADD 1 TO CHKPT-ID-CTR                                    EXA02180
             MOVE CHKPT-ID TO PASS-CHKPT                              EXA02190
             PERFORM CHKPT-RTN THRU CHKPT-RTN-END                     EXA02200
             MOVE 0 TO CHKPT-LIMIT                                    EXA02210
                                                                      EXA02220
* OTHERWISE READ THE NEXT MESSAGE FROM THE QUEUE                     EXA02230
                                                                      EXA02240
        ELSE PERFORM READ-INPUT THRU READ-INPUT-END.                  EXA02250
                                                                      EXA02260
PROCESS-INPUT-END.                                                    EXA02270
    EXIT.                                                             EXA02280
                                                                      EXA02290
```

```
PRINT-LINE.                                                              EXA02300
    IF TOP-OF-PAGE                                                       EXA02310
       THEN MOVE HEADING-LINE TO OUTPUT-DATA                             EXA02320
            MOVE ASA-NEWPAGE TO OUTPUT-ASA                               EXA02330
            MOVE 0 TO LINE-CTR                                           EXA02340
            PERFORM ISRT-GSAM-OUTPUT THRU ISRT-GSAM-OUTPUT-END           EXA02350
            MOVE ASA-SPACE-TWO TO OUTPUT-ASA.                            EXA02360
                                                                         EXA02370
    MOVE DETAIL-LINE TO OUTPUT-DATA                                      EXA02380
    PERFORM ISRT-GSAM-OUTPUT THRU ISRT-GSAM-OUTPUT-END                   EXA02390
    ADD 1 TO LINE-CTR                                                    EXA02400
    IF LINE-CTR = 1 MOVE ASA-SPACE-ONE TO OUTPUT-ASA.                    EXA02410
PRINT-LINE-END.                                                          EXA02420
    EXIT.                                                                EXA02430
                                                                         EXA02440
PRINT-TOTALS.                                                            EXA02450
    MOVE VALID-CTR TO OUT-VALID                                          EXA02460
    MOVE INVALID-CTR TO OUT-INVALID                                      EXA02470
    MOVE TOTAL-LINE TO OUTPUT-DATA                                       EXA02480
    MOVE ASA-NEWPAGE TO OUTPUT-ASA                                       EXA02490
    PERFORM ISRT-GSAM-OUTPUT THRU ISRT-GSAM-OUTPUT-END                   EXA02500
    MOVE SPACES TO OUTPUT-ASA                                            EXA02510
    PERFORM ISRT-ALTPCB THRU ISRT-ALTPCB-END.                            EXA02520
PRINT-TOTALS-END.                                                        EXA02530
    EXIT.                                                                EXA02540
                                                                         EXA02550
* THE FOLLOWING ROUTINES EXECUTE THE DL/I CALLS                          EXA02560
* BUT DO NOT DO ANY APPLICATION PROCESSING                               EXA02570
                                                                         EXA02580
READ-INPUT.                                                              EXA02590
    CALL 'CBLTDLI' USING GU, IOPCB, INPUT-MSG.                           EXA02600
    IF TPSTATUS = 'QC'                                                   EXA02610
        THEN MOVE 1 TO END-SWITCH                                        EXA02620
        ELSE IF TPSTATUS NOT EQUAL SPACES                                EXA02630
             THEN CALL 'DFSOAER' USING                                   EXA02640
                  IOPCB, BAD-DC-CALL, INPUT-MSG, ERROPT.                 EXA02650
READ-INPUT-END.                                                          EXA02660
    EXIT.                                                                EXA02670
                                                                         EXA02680
READ-DB.                                                                 EXA02690
    CALL 'CBLTDLI' USING GHU, DBPCB, DB-IOAREA, SSA.                     EXA02700
    IF DBSTATUS = SPACES OR 'GE'                                         EXA02710
        THEN NEXT SENTENCE                                               EXA02720
        ELSE CALL 'DFSOAER' USING                                        EXA02730
                  DBPCB, BAD-DB-CALL, DB-IOAREA, ERROPT.                 EXA02740
READ-DB-END.                                                             EXA02750
    EXIT.                                                                EXA02760
                                                                         EXA02770
UPDATE-DB.                                                               EXA02780
    CALL 'CBLTDLI' USING REPL, DBPCB, DB-IOAREA.                         EXA02790
    IF DBSTATUS NOT EQUAL SPACES                                         EXA02800
        THEN CALL 'DFSOAER' USING                                        EXA02810
                  DBPCB, BAD-DB-CALL, DB-IOAREA, ERROPT.                 EXA02820
UPDATE-DB-END.                                                           EXA02830
    EXIT.                                                                EXA02840
                                                                         EXA02850
ISRT-GSAM-OUTPUT.                                                        EXA02860
    CALL 'CBLTDLI' USING ISRT, GSAMPCB-OUT, OUTPUT-LINE.                 EXA02870
    IF GSTATUS-OUT NOT EQUAL SPACES                                      EXA02880
        THEN CALL 'DFSOAER' USING                                        EXA02890
            GSAMPCB-OUT, BAD-DB-CALL, OUTPUT-LINE, ERROPT.               EXA02900
ISRT-GSAM-OUTPUT-END.                                                    EXA02910
    EXIT.                                                                EXA02920
                                                                         EXA02930
ISRT-ALTPCB.                                                             EXA02940
    CALL 'CBLTDLI' USING CHNG, ALTPCB, SUPER-LTERM                       EXA02950
    IF ALTSTATUS NOT EQUAL SPACES                                        EXA02960
        THEN CALL 'DFSOAER' USING                                        EXA02970
             ALTPCB, BAD-DC-CALL, SUPER-LTERM, ERROPT.                   EXA02980
    CALL 'CBLTDLI' USING ISRT, ALTPCB, OUTPUT-MSG.                       EXA02990
    IF ALTSTATUS NOT EQUAL SPACES                                        EXA03000
        THEN CALL 'DFSOAER' USING                                        EXA03010
           ALTPCB, BAD-DC-CALL, OUTPUT-MSG, ERROPT.                      EXA03020
```

```
ISRT-ALTPCB-END.                                                        EXA03030
    EXIT.                                                               EXA03040
                                                                        EXA03050
CHKPT-RTN.                                                              EXA03060
    CALL 'CBLTDLI' USING CHKP, IOPCB, IOAREA-LEN, CHKPT-AREA,           EXA03070
        COUNTER-LEN, COUNTERS.                                          EXA03080
    IF TPSTATUS = 'XD'                                                  EXA03090
        THEN MOVE 1 TO CLOSE-SWITCH                                     EXA03100
        ELSE IF TPSTATUS = 'QC'                                         EXA03110
             THEN MOVE 1 TO END-SWITCH                                  EXA03120
             ELSE IF TPSTATUS NOT EQUAL SPACES                          EXA03130
                  CALL 'DFSOAER' USING                                  EXA03140
                     IOPCB, BAD-DC-CALL, CHKPT-ID, ERROPT.              EXA03150
CHKPT-RTN-END.                                                          EXA03160
    EXIT.                                                               EXA03170
                                                                        EXA00440
```

APPENDIX C. SAMPLE MESSAGE PROCESSING PROGRAM

This program processes the Primer sample parts data base.

```
PE4NINQ: PROCEDURE (C1PC_PTR,D1PC_PTR) OPTIONS (MAIN);                  00001000
                                                                        00002000
/* * * D E C L A R A T I O N S * * */                                   00003000
                                                                        00004000
  DCL 1 C1PC BASED (C1PC_PTR),                                          00005000
        2 FILL CHAR (10),                                               00006000
        2 STAT CHAR (2),                                                00007000
      1 D1PC BASED (D1PC_PTR) LIKE C1PC;                                00008000
                                                                        00009000
  DCL 1 INPUT_MESSAGE,                                                  00010000
        2 FILL1 CHAR (6),                                               00011000
        2 TRANS_CODE CHAR (9),                                          00012000
        2 FE0OGCNR CHAR (6),                                            00013000
        2 FILL2 CHAR (60),                                              00014000
                                                                        00015000
      1 OUT_MESSAGE,                                                    00016000
        2 OUT_LL INIT (111) FIXED BINARY (31),                          00017000
        2 OUT_ZZ INIT (0) FIXED BINARY (15),                            00018000
        2 OUT_DETAILS,                                                  00019000
          3 FE2PCNUM CHAR (6),                                          00020000
          3 (FE2PCNAM,                                                  00021000
             FE2PCADR,                                                  00022000
             FE2PCCTY) CHAR (20),                                       00023000
          3 FE2PCPCD CHAR (6),                                          00024000
        2 OUT_ERROR CHAR (35),                                          00025000
                                                                        00026000
      1 SE2PCUST,                                                       00027000
        2 CUST_DETAILS LIKE OUT_DETAILS,                                00028000
        2 FILL CHAR (40),                                               00029000
                                                                        00030000
      1 CUSTOMER_SSA,                                                   00031000
        2 FILL1 CHAR (19) INIT ('SE2PCUST(FE2PCNUM ='),                 00032000
        2 SSA_CNUM CHAR (6),                                            00033000
        2 FILL2 CHAR (1) INIT (')');                                    00034000
                                                                        00035000
  DCL ((GU INIT ('GU'),                                                 00036000
        ISRT INIT ('ISRT'),                                             00037000
        ERROPT INIT ('1')) CHAR (4),                                    00038000
       (MODNAME INIT ('0E4CNI01'),                                      00039000
        BAD_CALL INIT ('BAD CALL')) CHAR (8),                           00040000
       (THREE INIT (3),                                                 00041000
        FOUR INIT (4)) FIXED BINARY (31)) STATIC,                       00042000
      (C1PC_PTR,D1PC_PTR) POINTER,                                      00043000
      (PLITDLI, DFS0AER OPTIONS (ASSEMBLER)) ENTRY;                     00044000
                                                                        00045000
/* * * P R O C E S S    M E S S A G E S * * */                          00046000
                                                                        00047000
READ_MESSAGE:                                                           00048000
                                                                        00049000
  CALL PLITDLI (THREE,GU,C1PC_PTR,INPUT_MESSAGE);                       00050000
  IF C1PC.STAT = 'QC' THEN RETURN;                                      00051000
  IF C1PC.STAT ¬= ' '                                                   00052000
     THEN CALL DFS0AER (C1PC,BAD_CALL,INPUT_MESSAGE,ERROPT);            00053000
  SSA_CNUM = FE0OGCNR;                                                  00054000
                                                                        00055000
/* * * R E A D   C U S T O M E R   D A T A   B A S E * * */             00056000
                                                                        00057000
  CALL PLITDLI (FOUR,GU,D1PC_PTR,SE2PCUST,CUSTOMER_SSA);                00058000
  IF D1PC.STAT = ' ' THEN DO;                                           00059000
     OUT_DETAILS = CUST_DETAILS;                                        00060000
     OUT_ERROR = ' ';                                                   00061000
     END;                                                               00062000
  ELSE IF D1PC.STAT = 'GE' THEN DO;                                     00063000
     OUT_ERROR = 'INVALID NUMBER - PLEASE RE-ENTER';                    00064000
     OUT_DETAILS = ' ';                                                 00065000
     END;                                                               00066000
  ELSE CALL DFS0AER (D1PC,BAD_CALL,SE2PCUST,ERROPT);                    00067000
                                                                        00068000
/* * * I N S E R T   M E S S A G E * * */                               00069000
                                                                        00070000
  CALL PLITDLI (FOUR,ISRT,C1PC_PTR,OUT_MESSAGE,MODNAME);                00071000
  IF C1PC.STAT ¬= ' '                                                   00072000
     THEN CALL DFS0AER (C1PC,BAD_CALL,OUT_MESSAGE,ERROPT);              00073000
```

```
GO TO READ_MESSAGE;                                          00074000
                                                             00075000
END PE4NINQ;                                                 00076000
```

APPENDIX D. SAMPLE CONVERSATIONAL MPP

This program updates the price field of the root segment of a new price for that part. There are two passes in the conversation:

- To start the conversation, the person at the terminal enters the transaction code and the number of the part whose price will be updated.

  The progam retrieves the root segment for that part from the parts data base by qualifying the SSA of the part number. The program also saves the part number and the current price from the root segment in the SPA, then sends an output message to the terminal that gives the current price.

  If the part number that was entered is invalid, the program sends an error message to the terminal and ends the conversation by inserting blanks in the area of the SPA that contains the transaction code (the first 8 bytes).

- The person at the terminal then enters the new price. Using the part number stored in the SPA, the program retrieves the root segment and checks to see if the price in the SPA matches the price in the data base segment. If the price in the data base hasn't been updated, the program updates the data base with the new price and sends a message to the terminal giving the old and new prices. The program terminates the conversation by inserting blanks in the transaction code area of the SPA. The reason that the program has to check the price during pass 2 is that you can't enqueue a data base record across passes of a conversation. For example, someone at another terminal could have entered the same transaction and completed it before the first person entered the data for pass.

```
IDENTIFICATION DIVISION.                                                EXA00110
PROGRAM-ID. 'SAMPLE4'.                                                  EXA00120
REMARKS.                                                                EXA00130
    THIS PROGRAM IS A CONVERSATIONAL MESSAGE PROCESSING                 EXA00140
    PROGRAM WHICH UPDATES THE PRICE FIELD IN THE ROOT                   EXA00150
    SEGMENT OF THE PARTS DATA BASE.                                     EXA00160
ENVIRONMENT DIVISION.                                                   EXA00170
DATA DIVISION.                                                          EXA00180
WORKING-STORAGE SECTION.                                                EXA00190
                                                                        EXA00200
* DL/I FUNCTION CODES                                                   EXA00210
                                                                        EXA00220
77  FUNC  PIC X(4).                                                    EXA00230
77  GU    PIC X(4) VALUE 'GU  '.                                        EXA00240
77  GN    PIC X(4) VALUE 'GN  '.                                        EXA00250
77  ISRT  PIC X(4) VALUE 'ISRT'.                                       EXA00260
77  REPL  PIC X(4) VALUE 'REPL'.                                       EXA00270
                                                                        EXA00280
* THIS SWITCH IS SET TO 1 IF A QC IS RETURNED WHEN                      EXA00290
* RETRIEVING THE NEXT MESSAGE.                                          EXA00300
                                                                        EXA00310
77  END-SWITCH PIC X VALUE '0'.                                        EXA00320
    88 NO-MORE-INPUT VALUE '1'.                                         EXA00330
                                                                        EXA00340
* PARAMETERS USED BY DFSOAER STATUS CODE CHECKING ROUTINE               EXA00350
                                                                        EXA00360
01  DFSOAER-FIELDS.                                                    EXA00370
    02  ERROPT PIC X(4) VALUE '1   '.                                  EXA00380
    02  BAD-DB-CALL PIC X(8) VALUE 'DBADCALL'.                         EXA00390
    02  BAD-DC-CALL PIC X(8) VALUE 'CBADCALL'.                         EXA00400
                                                                        EXA00410
                                                                        EXA00420
* SCRATCH PAD AREA                                                      EXA00430
                                                                        EXA00440
01  SPA.                                                               EXA00450
    02  FILLER  PIC X(6).                                              EXA00460
    02  SPA-TRANCODE    PIC X(8).                                      EXA00470
                                                                        EXA00480
* THIS FIELD IS SET TO 1 DURING PASS 1 PROCESSING                       EXA00490
                                                                        EXA00500
    02  PASS-COUNT  PIC S9(3) COMP.                                    EXA00510
        88  FIRST-PASS VALUE +0.                                       EXA00520
    02  SPA-PARTNO  PIC X(8).                                          EXA00530
    02  SPA-OLD-PRICE   PIC 9(6)V99.                                   EXA00540
    02  FILLER PIC X(100).                                             EXA00550
                                                                        EXA00560
01  PASS1-INPUT.                                                       EXA00570
    02  IN-LL1  PIC S9(3) COMP.                                        EXA00580
    02  IN-ZZ1  PIC S9(3) COMP.                                        EXA00590
    02  IN-PARTNO   PIC X(8).                                          EXA00600
    02  FILLER  PIC X(80).                                             EXA00610
                                                                        EXA00620
01  PASS2-INPUT.                                                       EXA00630
    02  IN-LL2  PIC S9(3) COMP.                                        EXA00640
    02  IN-ZZ2  PIC S9(3) COMP.                                        EXA00650
    02  NEW-PRICE PIC 9(6)V99.                                         EXA00660
    02  FILLER  PIC X(100).                                            EXA00670
                                                                        EXA00680
01  OUTPUT-MSG.                                                        EXA00690
    02  OUT-LL  PIC S9(3) COMP VALUE +72.                              EXA00700
    02  OUT-ZZ  PIC S9(3) COMP VALUE +0.                               EXA00710
    02  OUT-DATA.                                                      EXA00720
        04  OUT-PARTNO  PIC X(8).                                      EXA00730
        04  OUT-OLD-PRICE  PIC Z(6)9.99.                               EXA00740
        04  OUT-NEW-PRICE  PIC Z(6)9.99.                               EXA00750
        04  COMMENTS PIC X(40).                                        EXA00760
                                                                        EXA00770
01  DB-IOAREA.                                                         EXA00780
    02  DB-PARTNO    PIC X(8).                                         EXA00790
    02  FILLER       PIC X(45).                                        EXA00800
    02  DB-PRICE     PIC 9(6)V99.                                      EXA00810
    02  FILLER       PIC X(19).                                        EXA00820
                                                                        EXA00830
```

```
* SEGMENT SEARCH ARGUMENT                                              EXA00840
                                                                       EXA00850
01  SSA.                                                              EXA00860
    02  FILLER PIC X(19) VALUE 'SE1PART (FE1PGPNR ='.                 EXA00870
    02  SSA-PARTNO PIC X(8).                                          EXA00880
    02  FILLER PIC X VALUE ')'.                                       EXA00890
                                                                       EXA00900
LINKAGE SECTION.                                                       EXA00910
                                                                       EXA00920
01  IOPCB.                                                            EXA00930
    02  FILLER  PIC X(10).                                            EXA00940
    02  TPSTATUS PIC X(2).                                            EXA00950
    02  FILLER  PIC X(20).                                            EXA00960
                                                                       EXA00970
01  DBPCB.                                                            EXA00980
    02  FILLER  PIC X(10).                                            EXA00990
    02  DBSTATUS PIC X(2).                                            EXA01000
    02  FILLER  PIC X(20).                                            EXA01010
                                                                       EXA01020
PROCEDURE DIVISION.                                                    EXA01030
                                                                       EXA01040
* ON ENTRY IMS PASSES THE FOLLOWING PCB ADDRESSES:                     EXA01050
                                                                       EXA01060
* IOPCB - FOR RETRIEVING THE SPA AND INPUT MESSAGE SEGMENT             EXA01070
* DBPCB - PARTS DATA BASE                                             EXA01080
                                                                       EXA01090
    ENTRY 'DLITCBL' USING IOPCB, DBPCB.                                EXA01100
    PERFORM READ-SPA THRU READ-SPA-END                                 EXA01110
    PERFORM MAINLINE THRU MAINLINE-END                                 EXA01120
        UNTIL NO-MORE-INPUT.                                           EXA01130
    GOBACK.                                                            EXA01140
                                                                       EXA01150
MAINLINE.                                                              EXA01160
                                                                       EXA01170
* HERE WE CHECK A FIELD IN THE SPA TO SEE IF IT IS                     EXA01180
* BINARY ZEROES, WHICH INDICATES THAT THIS IS THE                      EXA01190
* FIRST PASS OF THE CONVERSATION.                                      EXA01200
                                                                       EXA01210
                                                                       EXA01220
    IF FIRST-PASS PERFORM PASS1 THRU PASS1-END                         EXA01230
              ELSE PERFORM PASS2 THRU PASS2-END.                       EXA01240
                                                                       EXA01250
* AFTER PERFORMING THE NECESSARY DATA BASE PROCESSING                  EXA01260
* WE INSERT THE SCRATCH PAD AREA AND A MESSAGE TO                      EXA01270
* TO THE OPERATOR.                                                    EXA01280
* THEN WE RETRIEVE ANOTHER MESSAGE FROM THE INPUT MESSAGE              EXA01290
* QUEUE. IF A QC STATUS CODE IS RECEIVED, A SWITCH                     EXA01300
* IS SET ON, AND THE PROGRAM WILL TERMINATE.                           EXA01310
* OTHERWISE THE MAINLINE LOOP IS REPEATED.                             EXA01320
                                                                       EXA01330
    PERFORM ISRT-SPA THRU ISRT-SPA-END                                 EXA01340
    PERFORM ISRT-MSG THRU ISRT-MSG-END                                 EXA01350
    PERFORM READ-SPA THRU READ-SPA-END.                                EXA01360
                                                                       EXA01370
MAINLINE-END.                                                          EXA01380
    EXIT.                                                              EXA01390
                                                                       EXA01400
* PASS 1 PROCESSING                                                    EXA01410
                                                                       EXA01420
PASS1.                                                                 EXA01430
* READ THE INPUT MESSAGE CONTAINING THE PART NUMBER                    EXA01440
    PERFORM READ-PASS1 THRU READ-PASS1-END.                            EXA01450
    MOVE SPACES TO OUT-DATA                                            EXA01460
                                                                       EXA01470
* SET UP THE SSA AND FUNCTION CODE FOR THE DATA BASE CALL              EXA01480
    MOVE IN-PARTNO TO SSA-PARTNO, OUT-PARTNO                           EXA01490
    MOVE GU TO FUNC.                                                   EXA01500
    PERFORM READ-DB THRU READ-DB-END                                   EXA01510
                                                                       EXA01520
* IF THE PART NUMBER IS AN INVALID KEY, SET UP AN                      EXA01530
* ERROR MESSAGE FOR THE OPERATOR, AND BLANK THE TRANSACTION            EXA01540
* CODE IN THE SPA TO TERMINATE THE CONVERSATION                        EXA01550
                                                                       EXA01560
```

```
    IF DBSTATUS = 'GE'                                                  EXA01570
        MOVE 'NOT ON FILE' TO COMMENTS                                  EXA01580
        MOVE SPACES TO SPA-TRANCODE                                     EXA01590
                                                                        EXA01600
* IF THE PART NUMBER IS A VALID KEY, STORE THE KEY AND                  EXA01610
* THE CURRENT PRICE IN THE SPA, AND CHANGE THE SPA                      EXA01620
* INDICATOR FIELD TO 1, TO INDICATE THAT PASS 1 HAS BEEN               EXA01630
* SUCCESSFULLY COMPLETED.                                              EXA01640
                                                                        EXA01650
    ELSE MOVE DB-PRICE TO SPA-OLD-PRICE, OUT-OLD-PRICE                  EXA01660
         MOVE IN-PARTNO TO SPA-PARTNO                                   EXA01670
         MOVE +1 TO PASS-COUNT                                          EXA01680
         MOVE 'ENTER NEW PRICE' TO COMMENTS.                            EXA01690
                                                                        EXA01700
PASS1-END.                                                              EXA01710
    EXIT.                                                               EXA01720
                                                                        EXA01730
* PASS 2 PROCESSING                                                     EXA01740
                                                                        EXA01750
PASS2.                                                                  EXA01760
                                                                        EXA01770
* READ THE INPUT MESSAGE                                               EXA01780
                                                                        EXA01790
    PERFORM READ-PASS2 THRU READ-PASS2-END.                             EXA01800
                                                                        EXA01810
* SET UP THE SSA AND FUNCTION CODE FOR THE DATA BASE CALL               EXA01820
* AND MOVE THE OLD PRICE TO THE OUTPUT MESSAGE AREA                     EXA01830
                                                                        EXA01840
    MOVE SPACES TO OUT-DATA                                             EXA01850
    MOVE SPA-PARTNO TO SSA-PARTNO, OUT-PARTNO                           EXA01860
    MOVE SPA-OLD-PRICE TO OUT-OLD-PRICE                                 EXA01870
    MOVE 'GHU' TO FUNC                                                  EXA01880
    PERFORM READ-DB THRU READ-DB-END                                    EXA01890
                                                                        EXA01900
* IF THE DATA BASE RECORD HAS BEEN DELETED SINCE PASS 1                 EXA01910
* (BY SOME OTHER TRANSACTION TYPE), SEND AN ERROR                       EXA01920
* MESSAGE TO THE OPERATOR                                              EXA01930
                                                                        EXA01940
    IF DBSTATUS = 'GE'                                                  EXA01950
        MOVE 'NOT ON FILE' TO COMMENTS                                  EXA01960
                                                                        EXA01970
* OTHERWISE UPDATE THE DATA BASE AND MOVE THE NEW                       EXA01980
* PRICE TO THE OUTPUT MESSAGE AREA                                     EXA01990
                                                                        EXA02000
    ELSE IF SPA-OLD-PRICE = DB-PRICE                                    EXA02010
         THEN MOVE NEW-PRICE TO DB-PRICE, OUT-NEW-PRICE                 EXA02020
             PERFORM UPDATE-DB THRU UPDATE-DB-END                       EXA02030
             MOVE ' PRICE CHANGED' TO COMMENTS                          EXA02040
         ELSE MOVE ' PRICE ALREADY CHANGED' TO COMMENTS                 EXA02050
              MOVE DB-PRICE TO OUT-NEW-PRICE.                           EXA02060
                                                                        EXA02070
* BLANK THE TRANSACTION CODE IN THE SPA TO TERMINATE THE                EXA02080
* CONVERSATION AT THE END OF THIS PASS                                 EXA02090
                                                                        EXA02100
    MOVE SPACES TO SPA-TRANCODE.                                        EXA02110
                                                                        EXA02120
PASS2-END.                                                              EXA02130
    EXIT.                                                               EXA02140
                                                                        EXA02150
* THE FOLLOWING SUB ROUTINES PERFORM THE DLI CALLS                      EXA02160
* BUT DO NO APPLICATION PROCESSING OTHER THAN                           EXA02170
* CHECKING FOR VALID STATUS CODES.                                     EXA02180
                                                                        EXA02190
READ-SPA.                                                               EXA02200
    CALL 'CBLTDLI' USING GU, IOPCB, SPA.                                EXA02210
    IF TPSTATUS = 'QC' MOVE '1' TO END-SWITCH                           EXA02220
    ELSE IF TPSTATUS NOT EQUAL SPACES                                   EXA02230
         THEN CALL 'DFSOAER' USING                                      EXA02240
             IOPCB, BAD-DC-CALL, SPA, ERROPT.                           EXA02250
READ-SPA-END.                                                           EXA02260
    EXIT.                                                               EXA02270
                                                                        EXA02280
READ-PASS1.                                                             EXA02290
```

```
    CALL 'CBLTDLI' USING GN, IOPCB, PASS1-INPUT.                    EXA02300
    IF TPSTATUS NOT EQUAL SPACES                                    EXA02310
        CALL 'DFSOAER' USING                                        EXA02320
             IOPCB, BAD-DC-CALL, PASS1-INPUT, ERROPT.               EXA02330
READ-PASS1-END.                                                     EXA02340
    EXIT.                                                           EXA02350
                                                                    EXA02360
READ-PASS2.                                                         EXA02370
    CALL 'CBLTDLI' USING GN, IOPCB, PASS2-INPUT.                    EXA02380
    IF TPSTATUS NOT EQUAL SPACES                                    EXA02390
        THEN CALL 'DFSOAER' USING                                   EXA02400
             IOPCB, BAD-DC-CALL, PASS2-INPUT, ERROPT.               EXA02410
READ-PASS2-END.                                                     EXA02420
    EXIT.                                                           EXA02430
                                                                    EXA02440
READ-DB.                                                            EXA02450
    CALL 'CBLTDLI' USING FUNC, DBPCB, DB-IOAREA, SSA.               EXA02460
    IF DBSTATUS = SPACES OR 'GE'                                    EXA02470
    THEN NEXT SENTENCE                                              EXA02480
    ELSE CALL 'DFSOAER' USING                                       EXA02490
             DBPCB, BAD-DB-CALL, DB-IOAREA, ERROPT.                 EXA02500
READ-DB-END.                                                        EXA02510
    EXIT.                                                           EXA02520
                                                                    EXA02530
UPDATE-DB.                                                          EXA02540
    CALL 'CBLTDLI' USING REPL, DBPCB, DB-IOAREA.                    EXA02550
    IF DBSTATUS NOT EQUAL SPACES                                    EXA02560
    THEN CALL 'DFSOAER' USING                                       EXA02570
             DBPCB, BAD-DB-CALL, DB-IOAREA, ERROPT.                 EXA02580
UPDATE-DB-END.                                                      EXA02590
    EXIT.                                                           EXA02600
                                                                    EXA02610
ISRT-SPA.                                                           EXA02620
    CALL 'CBLTDLI' USING ISRT, IOPCB, SPA.                          EXA02630
    IF TPSTATUS NOT EQUAL SPACES                                    EXA02640
    THEN CALL 'DFSOAER' USING                                       EXA02650
            IOPCB, BAD-DC-CALL, SPA, ERROPT.                        EXA02660
ISRT-SPA-END.                                                       EXA02670
    EXIT.                                                           EXA02680
                                                                    EXA02690
ISRT-MSG.                                                           EXA02700
    CALL 'CBLTDLI' USING ISRT, IOPCB, OUTPUT-MSG.                   EXA02710
    IF TPSTATUS NOT EQUAL SPACES                                    EXA02720
    THEN CALL 'DFSOAER' USING                                       EXA02730
             IOPCB, BAD-DC-CALL, OUTPUT-MSG, ERROPT.                EXA02740
ISRT-MSG-END.                                                       EXA02750
    EXIT.                                                           EXA02760
--------------------
```

APPENDIX E. SAMPLE STATUS CODE ERROR ROUTINE (DFS0AER)

This sample status code error routine is provided as an example of an error routine. All of the sample programs call it. It is part of the Primer function.

```
*********************************************************************  00001000
*        IMS/VS PRIMER SAMPLE STATUS CODE ERROR ROUTINE                00002000
*********************************************************************  00003000
*                                                                      00004000
*********************************************************************  00005000
         TITLE  'IMS/VS PRIMER SAMPLE PCB DSECT MACRO'                 00006000
*********************************************************************  00007000
*                                                                      00008000
         MACRO                                                        00009000
         M00PCB &TYPE=DB,      PCB TYPE TO BE GENERATED (DB OR DC)    *00010000
               &PCB=D1PC       DSECT NAME,MAX 4 CHAR, ALSO USED,      00011000
.*                             AS FIRST 4 CHAR OF PCB SUB FIELDS.     00012000
         LCLC  &NM                                                    00013000
&NM      SETC  'D1PC'                                                 00014000
         AIF   (T'&PCB EQ 'O').L000                                   00015000
         AIF   (K'&PCB LE 4).L100   CORRECT LENGTH ?                  00016000
         MNOTE *,'  PCB= OPERAND IS TOO LONG, TRUNCATED TO 4CHAR'     00017000
.L100    ANOP                                                         00018000
&NM      SETC  '&PCB'(1,4)                                            00019000
.L000    ANOP                                                         00020000
         AIF   ('&TYPE' EQ 'DB').DB000                                00021000
         AIF   ('&TYPE' EQ 'DC').DC000                                00022000
         MNOTE *,'TYPE= OPERAND IS INVALID, ''DB'' ASSUMED '          00023000
.*                                                                    00024000
.*                                                                    00025000
.*                            DATABASE PCB:                           00026000
.*                           ------------------------------           00027000
.DB000   ANOP                                                         00028000
.*                                                                    00029000
&NM      DSECT              LAYOUT OF A DL/I DATABASE PCB             00030000
&NM.DBDN DS    CL8          DATABASE NAME                             00031000
&NM.LEVL DS    CL2          SEGMENT HIERARCHY LEVEL                   00032000
&NM.STAT DS    CL2          STATUSCODE                                00033000
&NM.PROC DS    CL4          PROCESSING OPTIONS                        00034000
&NM.RESV DS    CL4          RESERVED                                  00035000
&NM.SEGN DS    CL8          SEGMENTNAME FEEDBACK                      00036000
&NM.KFBL DS    F            LENGTH OF KEY FEEDBACK AREA               00037000
&NM.NSSG DS    F            NUMBER OF SENSITIVE SEGMENTS              00038000
&NM.KFBA DS    0C           KEY FEEDBACK AREA                         00039000
         MEXIT                                                        00040000
.*                                                                    00041000
.*                                                                    00042000
.*                              DC  - PCB:                            00043000
.*                             ------------------------               00044000
.DC000   ANOP                                                         00045000
.*                                                                    00046000
&NM      DSECT               LAYOUT OF A DC-PCB                       00047000
&NM.LTNM DS   CL8             SOURCE/DESTINATION LTERM NAME           00048000
&NM.RESV DS   CL2             RESERVED                                00049000
&NM.STAT DS   CL2             STATUSCODE                              00050000
&NM.PRFX DS   0CL12           INPUTPREFIX                             00051000
&NM.DATE DS   PL4             CURRENT DATE                            00052000
&NM.TIME DS   PL4             CURRENT TIME                            00053000
&NM.MSEQ DS   F               INPUT MESSAGE SEQUENCE NUMBER           00054000
&NM.MODN DS   CL8             MESSAGE FORMAT OUTPUT DESCRIPTION NAME  00055000
         MEXIT                                                        00056000
         MEND                                                         00057000
*                                                                     00058000
*********************************************************************  00059000
*        END OF IMS/VS PRIMER SAMPLE MACROS                            00060000
*********************************************************************  00061000
*                                                                      00062000
         LCLC  &PGMID                                                 00063000
         LCLC  &INVOMAX                                               00064000
&PGMID   SETC  'DFSOAER'                                              00065000
&INVOMAX SETC  '20'             DEFAULT VALUE FOR MAX. NUMBER OF CALLS00066000
*                                                                     00067000
*********************************************************************  00068000
         TITLE  'IMS/VS PRIMER SAMPLE STATUS CODE ERROR ROUTINE &PGMID'00069000
*********************************************************************  00070000
*                                                                      00071000
*                                                                      00072000
*                                                                      00073000
```

```
*        FUNCTION: TO BE CALLED BY IMS/VS APPLICATION PROGRAMS          00074000
*                  IF AN UNEXPECTED STATUS CODE WAS RECEIVED.           00075000
*                                                                       00076000
*        PROCESS:  PRINT ESSENTIAL PCB INFORMATION,                     00077000
*                  A CALL ID AND UP TO NINE PROGRAM AREAS.              00078000
*                  A DDNAME OF D00AERR IS REQUIRED.                     00079000
*                  RETURN TO CALLER IS MADE IF REQUESTED,               00080000
*                  AND NO ERRORS FOUND.                                 00081000
*                                                                       00082000
*                  MESSAGE DFS3125A IS ISUED IF REQUESTED BY CALLER.    00083000
*                  DEPENDENT UPON THE REPLY, THE ROUTINE                00084000
*                  WILL FORCE EITHER A PROGRAM LOOP, AN ABEND,          00085000
*                  OR RETURN TO CALLER.                                 00086000
*                                                                       00087000
*        ABEND:    WE WILL ISSUE USER ABEND 3400 IF:                    00088000
*                   1. REQUESTED BY USER                                00089000
*                   2. MAX NUMBER OF INVOCATIONS IS REACHED,            00090000
*                      SET BY GLOBAL  &INVOMAX                          00091000
*                   3. ERRORS IN CALL PARAMETERS ARE DETECTED           00092000
*                                                                       00093000
*                                                                       00094000
*        INPUT:    UPON ENTRY R1 MUST POINT TO PARMLIST:                00095000
*                  WITH ADDRESSES OF AT LEAST 4 PARMS:                  00096000
*                          1. A(PCB) EITHER A DB- OR A DC-PCB           00097000
*                          2. A(IDENTIFIER(8 BYTES) OF THE CALL)        00098000
*                               WHERE D....... DENOTES A DB-CALL        00099000
*                               AND   C....... DENOTES A DC-CALL        00100000
*                          3. A(AREA1) WE WILL DISPLAY 76 CHARACTERS    00101000
*                          4. A(OPTIONFIELDS)  A 4BYTE FIELD WHERE      00102000
*                                BYTE 1: C'1' = ABEND, NO RETURN        00103000
*                                               THIS IS NORMAL CASE.    00104000
*                                        C'0' = RETURN TO CALLER        00105000
*                                               THIS ENABLES MULTIPLE   00106000
*                                               INVOCATIONS, E.G.       00107000
*                                               FOR TESTING PURPOSES    00108000
*                                               IN THAT CASE A 'FINAL'  00109000
*                                               INVOCATION IS NEEDED:   00110000
*                                        C'2' = FINAL INVOCATION        00111000
*                                               PLUS RETURN TO CALLER.  00112000
*                                        C'3' = MESSAGE DFS3125A        00113000
*                                               REQUESTED BY CALLER     00114000
*                                BYTE 2-4: NOT USED                     00115000
*                           5-12. A(AREA2)...A(AREA9) OPTIONAL          00116000
*                                   ONLY 76 CHARACTERS OF EACH AREA     00117000
*                                   WILL BE LISTED                      00118000
*                                                                       00119000
*        OUTPUT  : FOR EACH PRINT REQUEST, ESSENTIAL PCB INFORMATION    00120000
*                  IS PRINTED AND UP TO 9 USER AREAS.                   00121000
*                                                                       00122000
*        MESSAGES: DFS3125A IF REQUESTED BY CALLER                      00123000
*                                                                       00124000
*                                                                       00125000
*        OS MACRO'S USED:    OPEN/CLOSE/DCB/PUT/ABEND                   00126000
*                                                                       00127000
*                                                                       00128000
*                                                                       00129000
*        REGISTER USAGE:                                                00130000
*        ---------------------------------------------                  00131000
*        REGISTER  EQUATED   USAGE                                      00132000
*        --------  --------  ------------------------------             00133000
*        0         R0        OS/VS LINKAGE                              00134000
*        1         R1        OS/VS LINKAGE                              00135000
*        2         R2        WORK                                       00136000
*        3         R3        WORK                                       00137000
*        4         R4        WORK                                       00138000
*        8         DBPCBR    DB PCB REGISTER                            00139000
*        9         DBPCBR    DC PCB REGISTER                            00140000
*        11        BASE1B    PROGRAM BASE REGISTER                      00141000
*        13        R13       PROGRAM SAVE AREA ADDRESS                  00142000
*        14        R14       OS/VS LINKAGE                              00143000
*        15        R15       OS/VS LINKAGE                              00144000
*                                                                       00145000
*********************************************************************** 00146000
```

```
*                                                                       00147000
         EJECT                                                          00148000
&PGMID   CSECT                                                          00149000
         B       IDLEN(0,R15)    BRANCH AROUND ID                       00150000
         DC      CL8'&PGMID'        PROGRAM ID.                         00151000
         DC      CL8'&SYSDATE'                                          00152000
         DC      CL8'&SYSTIME'                                          00153000
IDLEN    EQU     *-&PGMID                                               00154000
*                                                                       00155000
         STM     14,11,12(13)    SAVE REGISTERS, INCLUDING PL1'S        00156000
*                                                                       00157000
*                                PROVIDE A BASE REGISTER:               00158000
         BALR    BASE1,0                                                00159000
         USING   *,BASE1                                                00160000
*                                                                       00161000
*                                CHAIN OLD AND NEW SAVEAREAS            00162000
         ST      R13,SAVR+4                                             00163000
         LR      R10,R13                                                00164000
         LA      R13,SAVR                                               00165000
         ST      R13,8(R10)                                             00166000
         SR      R15,R15         CLEAR                                  00167000
*                                                                       00168000
         EJECT                                                          00169000
*********************************************************************** 00170000
MAIN     EQU     *                                                      00171000
*********************************************************************** 00172000
*                                PICK UP PARAMETERS RECEIVED:           00173000
*                                ------------------------------------   00174000
         MVC     PARM1ADR,00(R1)                                        00175000
         MVC     PARM2ADR,04(R1)                                        00176000
         MVC     PARM3ADR,08(R1)                                        00177000
         MVC     PARM4ADR,12(R1)                                        00178000
         MVI     PARM4ADR,X'00'     REMOVE HIORDER                      00179000
         LR      R3,R1                                                  00180000
*                                                                       00181000
*                                CHECK FOR DFS3125 MESSAGE REQUEST      00182000
*                                ------------------------------------   00183000
*                                                                       00184000
         L       R2,PARM4ADR     GET CONTROL BYTE                       00185000
         CLI     0(R2),C'3'      IS IT THREE(MESSAGE REQUEST) ?         00186000
         BE      MAIN400         IF SO, PROCESS IN MAIN400              00187000
*                                                                       00188000
*                                CHECK FIRST TIME SWITCH                00189000
*                                ------------------------------------   00190000
         CLI     SWITCH,C'1'                                            00191000
         BH      ABEND2          PGM RETURNED AFTER FINAL INVOCATION    00192000
         BL      MAIN100         OKE, PROCEED                           00193000
         MVI     SWITCH,C'0'     SET FIRST TIME SWITCH OFF              00194000
*                                                                       00195000
*                                OPEN DCB IF FIRST TIME THROUGH         00196000
*                                ------------------------               00197000
         OPEN    (D00AERR,(OUTPUT))                                     00198000
*                                                                       00199000
*                                DETERMINE WHAT TO DO:                  00200000
*                                --------------------                   00201000
MAIN100  EQU     *                                                      00202000
         CLI     0(R2),C'2'      FINAL INVOCATION ?                     00203000
         BE      MAIN900                                                00204000
         BH      ABEND           UNKNOWN CONDITION                      00205000
         L       R2,PARM2ADR     ADR OF CALLID                          00206000
         LTR     R2,R2           TEST IT                                00207000
         BZ      ABEND           IF 0, INVALID CALL, ABEND              00208000
         CLI     0(R2),C'D'      IS IT A DB PCB ?                       00209000
         BNE     MAIN500                                                00210000
         BAL     R14,PRTDB       PRINT DB-PCB FIELDS                    00211000
         B       END                                                    00212000
*                                                                       00213000
MAIN400  EQU     *               MESSAGE REQUESTED                      00214000
         BAL     R14,MESSAGE                                            00215000
         L       R4,INVOKECT                                            00216000
         LA      R4,1(R4)        ADD ONE TO NUMBER OF INVOCATIONS       00217000
         ST      R4,INVOKECT                                            00218000
         CLC     INVOKECT,INVOKMAX                                      00219000
```

```
         BE    ABEND2          ABEND IF MAX NUMBER IS REACHED            00220000
         B     RETURN                                                    00221000
*                                                                        00222000
MAIN500  EQU   *                                                        00223000
         CLI   0(R2),C'C'      IS IT A DC PCB ?                          00224000
         BNE   ABEND           INVALID CALL-ID, ABEND                    00225000
         BAL   R14,PRTDC       PRINT DC-PCB FIELDS                       00226000
         B     END                                                       00227000
*                                                                        00228000
MAIN900  EQU    *                                                       00229000
         MVI   SWITCH,C'2'     PROVIDE ABEND IF HE COMES BACK            00230000
         PUT   D00AERR,ENDLINE      PRINT FINAL LINE                     00231000
         CLOSE (D00AERR)       CLOSE OUTPUT-DCB                          00232000
         B     RETURN                                                    00233000
*                                                                        00234000
************************************************************************ 00235000
END      EQU   *                                                        00236000
************************************************************************ 00237000
*                                                                        00238000
         CLC   INVOKECT,INVOKMAX   MAXIMUM NR OF INVOCATIONS REACHED ?  00239000
         BE    ABEND               IF SO , ABEND                        00240000
         L     R2,PARM4ADR         4TH PARM WAS'OPTIONS'                00241000
         CLI   0(R2),C'0'          RETURN REQUESTED?                    00242000
         BE    RETURN              IF YES, CONTINUE                     00243000
ABEND    CLOSE (D00AERR)           CLOSE DCB                            00244000
ABEND2   ABEND 3400,DUMP                                                00245000
*                                                                        00246000
************************************************************************ 00247000
RETURN   EQU   *                   RETURN TO CALLER                     00248000
************************************************************************ 00249000
*                                                                        00250000
         L     R13,SAVR+4                                                00251000
         LM    14,11,12(R13)                                             00252000
         SR    R15,R15                                                   00253000
         BR    R14                                                       00254000
*                                                                        00255000
************************************************************************ 00256000
         TITLE '&PGMID: PRINT PCB-FIELDS --   DB-PCB '                  00257000
************************************************************************ 00258000
*                                                                        00259000
************************************************************************ 00260000
PRTDB    EQU   *                   DB-PCB FIELDS TO BE PRINTED          00261000
************************************************************************ 00262000
*                                                                        00263000
         ST    R14,SAVER14         RETURN ADDR                          00264000
*                                                                        00265000
*                                  FIND PCB TO BE USED:                 00266000
*                                  ------------------------------------ 00267000
         L     DBPCBR,PARM1ADR     ADR OF PCB                           00268000
         LTR   DBPCBR,DBPCBR       WAS IT SUPPLIED ?                    00269000
         BZ    END                 INVALID,SKIP PROCESSING              00270000
         USING DBPC,DBPCBR         ADDRESSABILITY OF PCB FIELDS         00271000
*                                                                        00272000
PRTDB100 EQU   *                                                        00273000
*                                  BUILD 1ST LINE AND DISPLAY IT        00274000
*                                  ------------------------------------ 00275000
         MVC   LN1HEAD(2),=C'DB'                                        00276000
         L     R2,PARM2ADR             ADR OF CALLID                    00277000
         MVC   LN1IDEN(8),0(R2)    CALL IDENTIFIER                      00278000
*                                                                        00279000
         L     R4,INVOKECT                                              00280000
         LA    R4,1(R4)            INCREMENT                            00281000
         ST    R4,INVOKECT         UPDATE COUNT                         00282000
         CVD   R4,WORK2                                                 00283000
         UNPK  WORK1(8),WORK2(8)                                        00284000
         OI    WORK1+07,X'F0'                                           00285000
         MVC   LN1COUNT(4),WORK1+4                                      00286000
*                                                                        00287000
         PUT   D00AERR,LINE1                                            00288000
*                                                                        00289000
*                                  BUILD 2ND LINE AND DISPLAY IT        00290000
*                                  ------------------------------------ 00291000
PRTDB200 EQU   *                                                        00292000
```

```
         MVC   LN2DBDN(DBDNLEN),DBPCDBDN                                00293000
         MVC   LN2LEVL(LEVLLEN),DBPCLEVL                                00294000
         MVC   LN2STAT(STATLEN),DBPCSTAT                                00295000
         MVC   LN2PROC(PROCLEN),DBPCPROC                                00296000
         MVC   LN2SEGN(SEGNLEN),DBPCSEGN                                00297000
         L     R4,DBPCKFBL                                              00298000
         CVD   R4,WORK2                                                 00299000
         UNPK  WORK1(8),WORK2(8)                                        00300000
         OI    WORK1+07,X'F0'                                           00301000
         MVC   LN2KFBL(4),WORK1+4                                       00302000
         L     R4,DBPCNSSG                                              00303000
         CVD   R4,WORK2                                                 00304000
         UNPK  WORK1(8),WORK2(8)                                        00305000
         OI    WORK1+07,X'F0'                                           00306000
         MVC   LN2NSSG(4),WORK1+4                                       00307000
         PUT   D00AERR,LINE2                                            00308000
*                                                                       00309000
PRTDB300 EQU   *                                                        00310000
*                           BUILD 3RD LINE  AND DISPLAY  IT             00311000
*                           ---------------------------------------     00312000
         L     R4,DBPCKFBL   LENGTH OF KFB DATA                         00313000
         LTR   R4,R4         ZERO ?                                     00314000
         BZ    PRTDB320      SKIP                                       00315000
         C     R4,=F'73'     EXCEED MAX LENGTH FOR 1 PRINTLINE          00316000
         BNH   PRTDB310      NO - OK                                    00317000
         LA    R4,73         YES- SET MAX                               00318000
PRTDB310 EQU   *                                                        00319000
         BCTR  R4,0          MIN 1 FOR EX                               00320000
         EX    R4,MOVEKFB    MOVE KFB DATA                              00321000
PRTDB320 EQU   *                                                        00322000
         PUT   D00AERR,LINE3                                            00323000
         MVC   LN3KFBA(1),SPACES         CLEAR IT                       00324000
         MVC   LN3KFBA+1(72),LN3KFBA    AFTER USAGE                     00325000
*                                                                       00326000
*                           PRINT AREA1 AND OPTIONALS                   00327000
*                           ---------------------------------------     00328000
PRTDB009 L     R2,PARM3ADR   ADDRESS OF AREA1                           00329000
         LA    R4,241(0,0)   LOAD 'F1' IN R4                            00330000
         LA    R3,12(0,R3)                                              00331000
PRTDB010 STC   R4,LINENUM    SET AREA NUMBER                            00332000
         MVC   LN4AREA(74),0(R2) GET DATA                               00333000
         PUT   D00AERR,LINE4                                            00334000
         CLI   0(R3),X'00'  LAST PARAMETER ?                            00335000
         BNE   PRTDB900      READY                                      00336000
         LA    R3,4(0,R3)    STEP TO NEXT AREA ADDRESS                  00337000
         L     R2,0(R3)      LOAD AREA ADDRESS                          00338000
         LA    R4,1(0,R4)    ADD ONE TO AREA NUMBER                     00339000
         B     PRTDB010                                                 00340000
*                           READY:                                      00341000
*                           ---------------------------------------     00342000
PRTDB900 EQU   *                                                        00343000
         L     R14,SAVER14   RETURN ADDRESS                             00344000
         BR    R14           RETURN                                     00345000
MOVEKFB  MVC   LN3KFBA(1),DBPCKFBA                                      00346000
         EJECT                                                          00347000
*                                                                       00348000
*********************************************************************** 00349000
PRTDC    EQU   *             DC - PCB FIELDS TO BE PRINTED              00350000
*********************************************************************** 00351000
*                                                                       00352000
         ST    R14,SAVER14   RETURN ADDRESS                             00353000
*                                                                       00354000
*                           FIND PCB TO BE USED:                        00355000
*                           ---------------------------------------     00356000
*                                                                       00357000
         L     DCPCBR,PARM1ADR    GET PCB ADDRESS                       00358000
         LTR   DCPCBR,DCPCBR      WAS IT SUPPLIED?                      00359000
         BZ    ABEND              IF NOT GOTO ABEND.                    00360000
         USING DCPC,DCPCBR                                              00361000
*                                                                       00362000
*                           BUILD 1ST LINE AND DISPLAY IT               00363000
*                           ------------------------------              00364000
         MVC   LN1HEAD(2),=C'DC'    IT IS A DC-PCB                      00365000
```

```
         L     R2,PARM2ADR     ADDR OF  IDENTIFIER                          00366000
         MVC   LN1IDEN(8),0(R2)    INTO OUTPUTLINE                         00367000
         L     R4,INVOKECT     NR OF TIMES INVOKED                         00368000
         LA    R4,1(R4)        UP BY 1                                     00369000
         ST    R4,INVOKECT     UPDATE COUNTER FIELD                        00370000
         CVD   R4,WORK2                                                    00371000
         UNPK  WORK1(8),WORK2(8)                                           00372000
         OI    WORK1+07,X'F0'                                              00373000
         MVC   LN1COUNT(4),WORK1+4    INTO OUTPUT LINE                     00374000
         PUT   D00AERR,LINE1     DISPLAY IT                                00375000
*                           BUILD DC PCB LINE6                            00376000
*                           ------------------------------------          00377000
         MVC   LN6LTNM,DCPCLTNM                                            00378000
         MVC   LN6STAT,DCPCSTAT                                            00379000
         MVC   LN6MODN,DCPCMODN                                            00380000
         CLC   DCPCDATE,SPACES   IS IT A DUMMY/ALTERNATE PCB?              00381000
         BNE   PRTDC009                                                    00382000
         MVC   LN6DATE,SPACES                                              00383000
         MVC   LN6TIME,SPACES                                              00384000
         MVC   LN6MSEQ,SPACES                                              00385000
         PUT   D00AERR,LINE6                                               00385500
         B     PRTDB009          GO PRINT PROGRAM AREAS                    00386000
PRTDC009 UNPK  LN6DATE,DCPCDATE                                            00387000
         OI    LN6DATE+7,X'F0'                                             00388000
         UNPK  LN6TIME,DCPCTIME                                            00389000
         OI    LN6TIME+7,X'F0'                                             00390000
         L     R4,DCPCMSEQ                                                 00391000
         CVD   R4,WORK2                                                    00392000
         UNPK  LN6MSEQ,WORK2(8)                                            00393000
         OI    LN6MSEQ+7,X'F0'                                             00394000
         PUT   D00AERR,LINE6                                               00395000
         B     PRTDB009          GO PRINT PROGRAM AREAS                    00396000
*                                                                          00397000
********************************************************************       00398000
MESSAGE  EQU   *                 ISSUE DFS3125A MESSAGE                    00399000
********************************************************************       00400000
*                                                                          00401000
         MVC   WTORECB,=F'0'     CLEAR ECB                    @BM10815     00401500
         WTOR  'DFS3125A PRIMER SAMPLE TEST, REPLY CONT, LOOP, ABEND, O*    00402000
               R CANCEL JOB',REPLY,5,WTORECB,ROUTCDE=11                    00403000
         WAIT  ECB=WTORECB                                                 00404000
         CLC   REPLY(4),=C'CONT'                                           00405000
         BE    RETURN            RETURN TO CALLER                          00406000
         CLC   REPLY(5),=C'ABEND'                                          00407000
         BE    ABEND2            ABEND                                     00408000
LOOP     CLC   REPLY(4),=C'LOOP'                                           00409000
         BE    LOOP              LOOP                                      00410000
         B     MESSAGE           WRONG REPLY TRY AGAIN                     00411000
*                                                                          00412000
         TITLE '&PGMID:  PRINT PCB-FIELDS --  EQUATES,CONSTANTS,AREAS '    00413000
********************************************************************       00414000
*        CONSTANTS, EQUATES AND DATA-AREAS                                 00415000
********************************************************************       00416000
*                           REGISTER EQUATES:                             00417000
*                           ------------------------------------          00418000
R0       EQU   0                                                          00419000
R1       EQU   1                                                          00420000
R2       EQU   2                                                          00421000
R3       EQU   3                                                          00422000
R4       EQU   4                                                          00423000
R5       EQU   5                                                          00424000
R6       EQU   6                                                          00425000
R7       EQU   7                                                          00426000
DBPCBR   EQU   8                                                          00427000
DCPCBR   EQU   9                                                          00428000
R10      EQU   10                                                         00429000
BASE1    EQU   11                                                         00430000
R12      EQU   12                                                         00431000
R13      EQU   13                                                         00432000
R14      EQU   14                                                         00433000
R15      EQU   15                                                         00434000
*                                                                         00435000
*                                                                         00436000
```

```
         DS    0D                                                      00437000
WORK1    DC    D'0'                                                    00438000
WORK2    DC    D'0'                                                    00439000
DEVT     DC    2A(0)                                                   00440000
DDNMFLD  DC    CL8'D00AERR'                                            00441000
WTORECB  DC    F'0'                                                    00442000
SAVR     DC    20A(0)                                                  00443000
SAVER14  DC    A(0)                                                    00444000
PARM1ADR DC    A(0)                                                    00445000
PARM2ADR DC    A(0)                                                    00446000
PARM3ADR DC    A(0)                                                    00447000
PARM4ADR DC    A(0)                                                    00448000
INVOKMAX DC    A(&INVOMAX)          MAXIMUM NR OF INVOCATIONS          00449000
INVOKECT DC    A(0)                 COUNTER FIELD                      00450000
SWITCH   DC    C'1'                 FIRST TIME SWITCH                  00451000
REPLY    DS    CL5                  MESSAGE REPLY FIELD                00452000
         EJECT                                                         00453000
LINE1    EQU   *                                                       00454000
         DC    CL01'1'         ASA                                     00455000
LN1HEAD  DC    CL24'DB-PCB FIELDS PRINTOUT'                            00456000
         DC    CL04'ID= '                                              00457000
LN1IDEN  DC    CL08' '       CALL  IDENTIFIER                          00458000
         DC    CL02' '                                                 00459000
         DC    CL06'COUNT='                                            00460000
LN1COUNT DC    CL04'  '                                                00461000
         DC    CL31' '         FILLER                                  00462000
*                                                                      00463000
LINE2    EQU   *                                                       00464000
         DC    CL01' '         ASA                                     00465000
         DC    CL06' DBDN='                                            00466000
LN2DBDN  DC    CL08' '                                                 00467000
         DC    CL06' LEVL='                                            00468000
LN2LEVL  DC    CL02' '                                                 00469000
         DC    CL06' STAT='                                            00470000
LN2STAT  DC    CL02' '                                                 00471000
         DC    CL06' PROC='                                            00472000
LN2PROC  DC    CL04' '                                                 00473000
         DC    CL06' SEGN='                                            00474000
LN2SEGN  DC    CL08' '                                                 00475000
         DC    CL06' KFBL='                                            00476000
LN2KFBL  DC    CL04' '                                                 00477000
         DC    CL06' NSSG='                                            00478000
LN2NSSG  DC    CL04' '                                                 00479000
         DC    CL05' '         FILLER                                  00480000
*                                                                      00481000
LINE3    EQU   *                                                       00482000
         DC    CL01' '         ASA                                     00483000
         DC    CL06' KFBA='                                            00484000
LN3KFBA  DC    CL73' '                                                 00485000
*                                                                      00486000
LINE4    EQU   *                                                       00487000
         DC    CL01' '                                                 00488000
         DC    CL05' AREA'                                             00489000
LINENUM  DC    CL02' :'                                                00490000
LN4AREA  DC    CL74' '                                                 00491000
*                                                                      00492000
SPACES   DC    CL8' '                                                  00493000
LINE6    EQU   *                                                       00494000
         DC    CL01' '                                                 00495000
         DC    CL08' LTNAME='                                          00496000
LN6LTNM  DC    CL08' '                                                 00497000
         DC    CL06' STAT='                                            00498000
LN6STAT  DC    CL02' '                                                 00499000
         DC    CL06' DATE='                                            00500000
LN6DATE  DC    CL08' '                                                 00501000
         DC    CL06' TIME='                                            00502000
LN6TIME  DC    CL08' '                                                 00503000
         DC    CL06' SEQ='                                             00504000
LN6MSEQ  DC    CL08' '                                                 00505000
         DC    CL05' MOD='                                             00506000
LN6MODN  DC    CL08' '                                                 00507000
ENDLINE  DC    CL81'0*****NO MORE ERROR PRINTS REQUESTED*****'         00508000
*                                                                      00509000
```

```
         LTORG                                                          00510000
         EJECT                                                          00511000
*****************************************************************     00512000
*                              D C B                                    00513000
*****************************************************************     00514000
D00AERR  DCB    DSORG=PS,LRECL=80,RECFM=FA,MACRF=(PM),                 C00515000
               BLKSIZE=80,DDNAME=D00AERR                                00516000
         EJECT                                                          00517000
*****************************************************************     00518000
*                            DSECTS                                     00519000
*****************************************************************     00520000
*                                                                       00521000
*                                                                       00522000
LEVLLEN  EQU   2                                                        00523000
STATLEN  EQU   2                                                        00524000
PROCLEN  EQU   4                                                        00525000
SEGNLEN  EQU   8                                                        00526000
KFBLLEN  EQU   4                                                        00527000
NSSGLEN  EQU   4                                                        00528000
DBDNLEN  EQU   8                                                        00529000
LTNMLEN  EQU   8                                                        00530000
DATELEN  EQU   4                                                        00531000
TIMELEN  EQU   4                                                        00532000
MSEQLEN  EQU   4                                                        00533000
MODNLEN  EQU   8                                                        00534000
*                                                                       00535000
         M00PCB TYPE=DB,PCB=DBPC                                        00536000
         M00PCB TYPE=DC,PCB=DCPC                                        00537000
         END                                                            00538000
*****************************************************************     00539000
*        END IMS/VS PRIMER SAMPLE STATUS ERROR PRINT ROUTINE DFS0AER    00540000
*****************************************************************     00541000
```

APPENDIX F. USING THE DL/I TEST PROGRAM (DFSDDLT0)

CONTROL STATEMENTS

In the control statement formats below, the "$" indicates those fields which are usually filled in; the absence of the "$" indicates that the field can be left blank and the default used. If position 1 is left blank on any control statement, the statement type defaults to the prior statement type.

STATUS STATEMENT

The STATUS statement establishes print options and determines the PCB that subsequent calls are to be issued against.

The format of the STATUS statement is as follows:

	Position		Contents
$	1	=	S identifies this as a STATUS statement.
	2	=	Output device option. Blank - Use PRINTDD when in a DLI region; use I/O PCB in the MSG region. 1 - Use PRINTDD in MSG region if the DD statement is provided; otherwise, use I/O PCB. A - Same as if 1, and disregard all other fields in this STATUS statement.
	3	=	Print comment option. Blank - Do not print. 1 - Print always. 2 - Print only if compare done and equal.
	4	=	Not used.
	5	=	Print call option. 1 - Print always 2 - Print only if compare done and equal.
	6	=	Not used.
	7	=	Print compare option. Blank - Do not print. 1 - Print always. 2 - Print only if compare done and equal.
	8	=	Blank.
	9	=	Print PCB option. Blank - Do not print. 1 - Print always. 2 - Print only if compare done and equal.
	10	=	Not used.

11 = Print segment option.

Blank - Do not print.
1 - Print always.
2 - Print only if compare done and unequal.

EBCDIC characters are printed as they appear in the segment. Hexadecimal characters are displayed in two lines with the high-order four bits printed above the low-order four bits. The low-order four bits of data are printed on the same line as the EBCDIC data. Hexadecimal data is read from top to bottom, left to right.

12 = Set task time.

1 - Time each call.
2 - Time on unequal compares.
5 - Time and BFSP trace each call.
6 - Time and BFSP trace on unequal compares.

13 = Set real time.

1 - Time each call.
2 - Time on unequal compares.
5 - Time and BFSP trace each call.
6 - Time and BFSP trace on unequal compares.

14 - 15 = Reserved.

16 - 23 = DBD name.

This determines the PCB against which subsequent calls will be issued; hence, it must be a DBD name given in one of the PCBs in the PSB. The default PCB is the first data-base-PCB in the PSB. If positions 16 through 23 are blank, the current PCB is used. If positions 16 through 18 are blank, and positions 19 through 23 are not blank, then the nonblank positions are interpreted as the relative number of the desired data-base-PCB in the PSB. The number must be right-justified to position 23, but need not contain leading zeros. The user must insure that the relative DB PCB exists in the PSB because no checks are made to insure that a proper PCB is obtained in this manner.

24 = Print status option.

Blank - Use orint option and print this statement.
1 - Do not use print option in this statement.
2 - Do not print this STATUS statement.
3 - Do not print this STATUS statement or use print option.

25 - 28 = PCB processing option. This is optional and is only used when two PCBs have the same DBD name but different processing options. If nonblank, it is used in addition to the DBD name in positions 16 through 23 to select which PCB in the PSB to use. This must appear as it does in the processing option of the PCB desired.

29 - 80 = Not used.

If no STATUS statement is read, the default PCB is the first DB PCB in the PSB, and the print status option is 2. New STATUS

statements can be anywhere in the SYSIN stream, changing either the data base to be referenced or the options.

COMMENTS STATEMENT

There are two types of COMMENTS statements. The first, the unconditional statement, allows for unlimited comments, all of which are printed. The second type, the conditional statement, allows only limited comments, which are printed or not depending on other factors as described below.

Unconditional

	Position		Contents
$	1	=	U specifies an unconditional COMMENTS statement.
	2 - 80	=	Comments - any number of unconditional COMMENTS statements are allowed; they are printed when read. Time and date of printing are printed with each unconditional COMMENTS statement.

Conditional

	Position		Contents
$	1	=	T specifies a conditional COMMENTS statement.
	2 - 80	=	Comments - up to 5 conditional COMMENTS statements per call are allowed; no continuing mark in position 72 is required. Printing is conditioned as the STATUS statement. Printing is deferred until after the following call and optional compare are executed, but prior to the printing of the following call.

CALL STATEMENT

The CALL statement identifies the type of IMS/VS call to be made, and supplies information to be used by the call.

	Position		Contents
$	1	=	L identifies this as either a CALL or DATA statement
	3	=	SSA level (optional).
	4	=	Format options:

Blank - For formatted calls with intervening blanks in positions 24, 34, and 37.
U - If columns 16 onward are unformatted, with no blanks separating fields.
V - For the first statement describing a variable-length segment, when inserting or replacing only one variable-length segment. It is also used for the first statement describing the first segment of multiple variable-length segments.
M - For the second through last statement that begin data for a variable-length segment, when inserting or replacing multiple variable-length segments.
P - When inserting or replacing through path calls. It is used only in the first statement of fixed-length segment statements in path calls containing

both variable- and fixed-length segments.

5 - 8 = Number of times to repeat this call (optional) in the range of 0001 through 9999.

$ 10 - 13 = DL/I call function.

DATA - Indicates that this statement contains data to be used in an ISRT, REPL, SNAP, CHKP, or LOG call. See the following section on DATA statements for usage.
CONT - For a continuation statement for field data that was too long for previous CALL statement.

$ 16 - 23 = SSA segment name.

24 = Not used.

$ 25 = (,if segment is qualified.

26 - 33 = SSA field name.

34 = Not used.

$ 35 - 36 = DL/I call opertor or opertors.

37 = Not used.

$ 38 - XX = Field value (where the maximum value of XX=70).

$ XX + 1 =), end character.

$ 72 = Nonblank, if more SSAs. Blank, if this is the only or last SSA.

Position 3, the SSA level, is usually blank. If blank, the first CALL statement fills SSA 1, and each following CALL statement fills the next lower SSA. If the SSA level, position 3, is nonblank, the statement fills the SSA at that level, and the following CALL statement fills the next lower SSA.

Position 4 contains a U to indicate an alternative format for the CALL statement. In this case, from position 16 on is the exact SSA with no intervening blanks in positions 24, 34, and 37. If command calls (for example, *D) are to be used, then the U must be specified.

Positions 5 through 8 are usually blank, but if used, must be right-justified. The identical call is repeated as specified in positions 5 through 8.

Positions 10 through 13 contain the DL/I call function. The call function is required only for the first SSA of the call. If left blank, the call function from the previous CALL statement is used.

Positions 16 through 23 contain the segment name, if the call uses an SSA.

If there are mutilple SSAs in the call, each SSA should be entered in positions 16 through 23 of a separate statement. A nonblank in position 72 of any statement indicates that another SSA follows. Position 1 and 10 through 13 are blank for the second through last SSAs.

If the field value extends past 71, there is a nonblank in position 72 and CONT in positions 10 through 13 of the next statement, with the field value continued starting in position 16. Maximum field value is 256 bytes.

An alternative format for the CALL statement is available by putting a U in position 4. If you use this option, you must start

the exact SSA in position 16, with no intervening blanks in positions 24, 34, and 37. To continue an unformatted SSA, put a nonblank character in position 72, a U in position 4, and CONT in positions 10 through 13 of the next statement. Include the data of the SSA that is continuing in positions 16 through 71. Maximum size for an SSA is 290 bytes.

The maximum number of SSAs for this program is the same as the IMS/VS limit, which is 15.

DATA STATEMENT

DATA statements provide IMS/VS with segment information required for ISRT, REPL, SNAP, LOG, and CHKP calls.

For an IRST, REPL, SNAP, LOG, or CHKP call, statements containing segment data must follow immediately after the last (noncontinued) CALL statement. The DATA statements must have an L in column 1, and DATA in positions 10 through 13. The segment data appears in positions 16 through 71. Data continuation is indicated with a nonblank in position 72. On the continuation statement, positions 1 through 15 are blank, and the data is resumed in position 16. The maximum segment size in a batch region is based on the PSB I/O area size. This size may be specified by the user during PSBGEN, or it is calculated by the ACB utility. When running in an online region, a maximum size of 30736 is available.

Note: On ISRT calls, the last SSA can have only the segment name, with no qualification or continuation.

When inserting or replacing variable-length segments, as defined in a DBDGEN, or including variable-length data for a CHKP or LOG call, position 4 of the CALL statement must contain either a V or an M. V must be used if only one segment of variable length is being processed. Positions 5 through 8 must contain the length of the data, right-justified, with leading zeros. This value is converted to binary, and becomes the first two bytes of segment data. Segment-data-statements can be continued, as described above with the subsequent statements blank in positions 1 through 15, and the data starting in position 16.

If multiple variable-length segments are required (that is, concatenated logical child/logical parent segments both of which are variable length) for the first segment, there must be a V in position 4 and the length of that segment in positions 5 through 8. If that segment is longer than 56 bytes, then the data is continued as above, except that the last card to contain data for this segment must have a nonblank in position 72. The next statement applies to the next variable-length segment, and must contain an M in position 4 and the length of this segment in positions 5 through 8. Any number of variable-length segments can be concatenated in this manner. The M or V and the length must appear only in statements that begin data for a variable-length segment.

When inserting or replacing through path calls, a P position 4 causes the length field to be used as the length the segment will occupy in the user I/O area, without the length (LL) field of variable-length segments, as in the instructions for M, above. V, M, and P can be mixed in successive statements. The P appears in only the first statement of fixed-length segment DATA statements, in path calls which contain both variable- and fixed-length segments.

Parameter Length, SNAP Calls

On SNAP calls, the length of the SNAP parameters must be in positions 5 through 8. This number must be equal to the length of the SNAP parameters starting in position 16 plus an additional 2 bytes. The TEST program converts the length to binary and places

it in the first half-word of the user I/O area passed to DL/I. The parameters from position 16 are placed in the I/O area immediately following this half-word. If positions 5 through 8 are blank, a default of 22 is used as the parameter length.

All parameters are passed without change, with the following exceptions:

1. If the SNAP destination field specifies "DCB-addr" or ddname of PRINTDD, and if a PRINTDD statement is supplied to the test program, the test program replaces this parameter with the DCB address of the test program PRINTDD data set.

2. If running DFSDDLT0 in a dependent region, the results of a SNAP call are routed to the dependent region PRINTDD DCB in systems where the PRINTDD DCB is accessable. In systems such as MVS where the dependent region PRINTDD DCB is inaccessable from the control region, the default is the log data set.

Parameter Length, LOG Call

The LOG call is normally used with the I/O PCB. It can used in batch mode only if the CMPAT option of the PSBGEN statement is specified.

The LOG call can be specified in two ways:

1. A LOG call statement followed by a DATA statement with an L in column 1, a V in column 4, and the record length (in decimal) in columns 5 through 8, right-justified, and padded with zeros. For example:

```
Column      Column      Column      Column
  1           4          10          16

  L                      LOG
  L           V0016      DATA        00ASEGMENT ONE
```

   When this method is used, the first halfword of the record is eliminated. However, the specified length must include the 2 bytes that are eliminated.

2. A LOG call statement followed by a DATA statement with an L in column 1 and the record length (in binary) as the first halfword of the record. The second halfword of the record is binary zeros. For example:

```
Column      Column      Column      Column
  1           4          10          16

  L                      LOG
  L                      DATA        1000BSEGMENT TWO
```

   When this method is used, columns 5 through 8 should be blank.

Segment Length and Checking, All Calls

Because this program does not know segment lengths, the length of the segment displayed on REPL or ISRT calls is the number of DATA statements that have been read, times 56. IMS/VS knows the segment length and uses the proper length.

This program does no checking for errors in the call; invalid functions, segments, fields, operators, or field lengths are not detected by this program. The results os invalid statements passed to IMS/VS will be unpredictable.

COMPARE STATEMENT FOR PCB COMPARISONS

This is the format of the COMPARE statement used for PCB comparisons.

Position		Contents
1	=	E identifies this as a COMPARE statement.
2	=	H indicates hold COMPARE statement (see below for details). Blank indicates a reset of the hold condition or a single COMPARE statement.
3	=	Option requested if results of the compare are unequal: Blank - Use the default for the SNAP option. The normal default is 5. For information on how to change the default, see thecription of the "Option Statement." 1 - To request a SNAP of the complete I/O buffer pool. 2 - To request a SNAP of the entire region. This option is valid only for batch regions. 4 - To request a SNAP of the DL/I blocks. 8 - To abort this step and go to the end of the job. S - To SNAP subpools 0 through 127.

Note: Multiple functions of the first 4 options can be obtained by summing their respective hexadecimal values. For example, a value of 5 a request for a print of the I/) buffers and the DL/I blocks; and a value of D snaps the I/O pool, snaps the DL/I blocks, and aborts the program run.

4	=	Extended SNAP options, if the results of a compare are unequal: Blank - To ignore this extended option. P - To SNAP the complete buffer pool. S - To SNAP subpools 0 through 127. Note: In no case will an area be snapped twice; that is, a combination of 1P in positions 3 and 4 results in just one snap of the buffer pool. Similarly, a combination of SS results in just one snap of subpools 0 through 127.
5 - 6	=	Segment level.
7	=	Not used.
8 - 9	=	Status code, or one of the following: XX - Do not check status code. OK - Allow blank, GA, or GK.
10	=	Not used.
11 - 18	=	Segment name.
20 - 22	=	Length of feedback key.
23	=	Not used.
24 - XX	=	Concatenated key feedback.

72 = Nonblank to continue key feedback.

The COMPARE statement is optional. It can be used to do regression testing of known data bases, or to call for a print of blocks or buffer pool(s).

Any fields left blank are not compared to the corresponding field in the PCB. Since a blank is a valid status code, to not compare status codes, put XX in positions 8 and 9. To accept any valid status code, (that is, blank, GA, or GK), use OK in postions 8 and 9.

To execute the same COMPARE after each call, put an H in position 2. This is useful when loading a data base to compare to a blank status code only. Since the compare was done, the current control statement type is E in position 1; the next control statement read must therefore have its type in position 1 or it will default to E. The HOLD-COMPARE statement stays in effect until another COMPARE statement is read. If a new COMPARE statement is read, two compares will be done for the preceding call, since the HOLD-COMPARE and optional printing are done prior to reading the new COMPARE statement.

The total number of unequal compares will be reflected in the condition code returned for that step.

COMPARE STATEMENT FOR I/O AREA COMPARISONS

This is the format of the COMPARE statement used for I/O area comparisons.

Position		Contents
$ 1	=	E identifies this as a COMPARE statement.
3	=	Length field option. Blank - The LL field of the segment is not included in the comparison; only data is compared. L - The length in positions 5 through 8 is converted to binary and compared against the LL field of the segment.
4	=	Segment length option. Blank - Not a variable-length segment or nonpath call data compare. M - For the second or subsequent variable-length segment of a path call, or a concatenated logical child/logical parent segment. P - For a fixed-length segment in a path call. V - For a variable-length segment only, or for the first variable-length segment of multiple variable-length segments in a path call or for a concatenated logical child/logical parent segment.
5 - 8	=	nnnn, length of a variable-length segment, right-justified with leading zeros. If position 4 contains V, P, or M, then a value must appear in positions 5 through 8. If position 3 contains an L, this value is compared against the LL field of the returned segment. If position 3 is blank and the segment is not in a path call, then this value is used as the length of the comparison. The rules for continuations are the same as those described for the variable-length segment DATA

statement in the description of the CALL statement.

If this is a path call comparison, and position 4 contains P, then the value in positions 5 through 8 must be the exact length of the fixed segment used in the path call.

10 - 13 = DATA, this has to be specified in the first COMPARE DATA statement only.

16 - 71 = Data against which the segment is to be compared.

72 = Continuation or end of COMPARE statement:

Blank - Identifies the last COMPARE DATA statement for the current call, and causes the comparison to be made.
Nonblank - If the comparison data exceeds 56 characters, data is continued in positions 16 through 71 of the subsequent statements for a maximum total of 1500 bytes.

This COMPARE statement is optional. Its purpose is to COMPARE the segment returned by IMS/VS to the data in this statement to verify that the correct segment was retrieved.

The length in positions 5 through 8 is optional except as already noted; if present, this length is used in the COMPARE and in the display. If no length is specified, the shorter of either the length of data moved to the I/O area by IMS/VS, or the number of DATA statements read times 56 is used for the length of the comparison and display.

If both a COMPARE DATA and a COMPARE PCB statement are present, the COMPARE DATA statement must precede the COMPARE PCB statement.

The conditions for printing the COMPARE DATA statement are the same as for printing a COMPARE PCB statement; position 7 of the STATUS statement is used. The same unequal switch is set for either the COMPARE DATA or COMPARE PCB. However, if control block displays are requested for unequal comparisons, a COMPARE PCB statement is required to request these options.

The total number of unequal comparisons will be reflected in the condition code returned for that step.

OPTION STATEMENT

The purpose of the OPTION statement is to set the default SNAP option and/or the number of unequal comparisons before aborting the step. The default value for the number of unequal comparisons before aborting is 5.

The format of the statement is explained below.

Position **Contents**

1 = O identifies this a s an OPTION statement.

2 - 80 = Free-from coding.

The first operand is SNAP=x, where "x" ix the default SNAP option to be used. For an explanation of the possible values of "x", see the description of the "COMPARE Statement for PCB Comparisons."

The second operand is ABORT=xxxx, where "xxxx" is a 4-digit numeric value that sets the number of unequal comparisons before aborting the step.

Use of the following example of the OPTION statement will cause the DL/I test program to operate as it did prior to the release of IMS/VS Version 1, Modification Level 1, that is , it reinstates the old SWAP options:

```
Column
  1

  ObSNAP=b,ABORT=9999
```

SPECIAL CONTROL STATEMENTS

PUNCH STATEMENT

The PUNCH control statement provides the facility for this program to produce an output data set consisting of the PCB COMPARE statements, the user I/O area COMPARE statements, all other control statements read, or any combination of the above. An example of the use of this facility is to code the call, but not the COMPARE statements for a new test. Then, after verifying that the calls were executed as anticipated, another run is made where the PUNCH statement is used to cause the test program to merge the proper COMPARE statements, based on the results of the call, with the CALL statements read, producing a new output data set. This is then used as input for subsequent regression tests. If segments in an existing data base are changed, the use of this control statement causes a new test data set to be produced with the proper COMPARE statements. This eliminates the need to manually change the COMPARE statements because of a change in the segments of the test data base.

The PCB COMPARE statements are produced based on the information in the PCB after the call is completed. The COMPARE DATA statements are produced based on the data in the I/O area after the call is completed. All input control statements, other than COMPARE statements, can be produced to provide a new composite test with the new COMPARE statements properly merged. The data set produced can be sequenced.

Since the key feedback area of the PCB COMPARE statement can be long, two options are provided for producing these COMPARE statements. Either the complete key feedback can be provided, or the portion of the key feedback that does not fit on one statement can be dropped. Forty-eight bytes of key feedback fit on the first statement.

Getting the full data from the I/O area into the data COMPARE statement might also be excessive. An option is to put it all on the data COMPARE statements, or put only the first 56 bytes on the first statement and drop the rest. The test program compares only the first 56 bytes if it receives only one COMPARE DATA statement.

The PUNCH statement format is as follows:

Position		Contents
$ 1 - 3	=	CTI identifies this statement type.
$ 10 - 13	=	Punch control: PUNC - Begin punching. NPUN - Stop punching.
$ 16	=	Starts keyword parameters controlling the various options. These keywords are:

PCBL - To produce the full PCB COMPARE statement.
PCBS - To produce the PCB COMPARE, dropping the key feedback if it exceeds one statement.
DATAL _ To produce the complete COMPARE DATA statements.
DATAS - To produce only one statement of COMPARE DATA.
OTHER - To reproduce all control statements except COMPARE control statements.
START - To punch the starting sequence number in columns 73 through 80. Eight numeric characters must follow the START= parameter; leading and/or trailing zeros are required.
INCR - To add the increment to the sequence number of each statement. Four numeric characters must follow the INCR= parameter; leading and/or trailing zeros are required.

Some examples of the PUNCH control statement are:

```
CTL     PUNC  PCBL,DATAL,OTHER,START=00000010,INCR=0010
CTL     NPUN
```

PUNCH DD STATEMENT

The DD statement for the output data set is labelled PUNCH; the data set characteristics are fixed, unblocked, with a logical record length of 80.

An example of the PUNCHDD statement is:

```
//PUNCHDD     DD  SYSOUT=B
```

SYSIN2 DD STATEMENT

The data set specified by the SYSIN DD statement is the normal input data set for this program. It is sometimes desirable when processing an input data set that is on direct access or tape, to override or insert some control statements into this input stream. This is especially useful to obtain a SNAP after a particular call.

To provide this capability, a second input data set (SYSIN2) will be read if the DD statement is present in the JCL for the step. The records from the SYSIN2 data set are merged with records from the SYSIN data set, and the merged records become the input for this program.

The merging is done based on the sequence numbers in positions 73 through 80, and is a two-step process: first, positions 73 and 74 of SYSIN2 must be equal to the corresponding positions of SYSIN; then the merge is done based on positions 75 to 80.

This peculiarity of merging allows for multiple data sets (each with a different high-order sequence number in 73 and 74) that have been concatenated to form SYSIN, in other than positions 73 and 74 numeric sequence. The two-step merge logic permits SYSIN2 input to be merged appropriately into each of the concatenated data sets.

When the sequence numbers are equal, SYSIN2 overrides SYSIN.

Any statements or records in this data set must contain sequence numbers in columns 73 through 80. They will replace the same sequence number in the SYSIN data set, or be inserted in proper sequence if the number in SYSIN2 does not exist in SYSIN.

Replacement or merging is done only for the run being made. The original SYSIN data is not changed.

OTHER CONTROL STATEMENTS

Position		Contents
1 - 4	=	DLCK: To issue an OS/VS checkpoint, followed by a DL/I checkpoint. For any dependent region, DLCK gives an OS/VS checkpoint to a DD statement labelled CHKDD whose DSORG=PO. This is followed by a DL/I checkpoint call. The use of this control statement will cause all subsequent CHKP calls to issue the OS/VS checkpoint unless a statement with USCKOFF in columns 1 through 7 precedes the CHKP call. CHKP: Same as DLCK.
10 - 17	=	Contains a 1- to 8-character checkpoint ID (left justified).
1 - 4	=	WTOR: puts message in remainder of statement on system console and waits for any reply, then continues.
1 - 3	=	WTO: same as WTOR, but does not wait for reply.
1	=	. or N: used as last statement in a data set that can be concatenated with other SYSIN data sets.
1 - 5	=	ABEND: To issues user ABEND 252 with the DUMP option.

SPECIAL CALL STATEMENTS

Position		Contents
$ 1	=	L identifies this as a CALL statement.
5 - 8	=	Number of times to repeat a series of calls with a range from 0001 through 9999 (default is 1).
$ 10 - 13	=	Stacking control cards: STAK - Start stacking control cards for later execution. END - Stop stacking control cards and begin execution. The STAK function makes it possible to repeat a series of calls which have been read from SYSIN and held in storage. All control statements between the STAK card and the END card are read and saved. When the END card is encountered, the series of calls is executed as many times as the number punched in positions 5 through 8 of the STAK card. This can be used to test exclusive control and scheduling by having two different regions executing stacks of calls concurrently. SKIP - Skip SYSINs until START statement statement encountered. START - Start making DL/I calls again. STAT - Print the current buffer pool statistics. When this call is used, IOASIZE in the PSB must be specified specified as greater than 360 bytes.

16 - 20 = One of the following values is used to obtain the type and form of statistics required:

VBASF - To obtain the full VSAM data base subpool statistics in a formatted form.
VBASU - To obtain the full VSAM data base subpool statistics in an unformatted form.
VBASS - To obtain a summary of the VSAM data base subpool statistics in formatted form.
DBASF - To obtain the full ISAM/OSAM data base buffer pool statistics in unformatted form.
DBASS - To obtain a summary of the ISAM/OSAM data base buffer pool statistics in formatted form.

Note that for VSAM statistics, a separate set of values is provided for each VSAM subpool defined, and a final set of values is provided to summarize all VSAM subpool values. The buffer size in the final totals is the total size of all buffers in all VSAM subpools.

SNAP - Issue the DL/I SNAP call to print the DL/I blocks.

EXECUTION IN DIFFERENT REGIONS

This program is designed to operate in a DL/I or BMP region but can also be executed in a MSG region. The input and output devices are dynamically established based on the type of region in which the program is executing. In a BMP or DL/I region, the EXEC statement allows the program name to be different from the PSB name. There is no problem executing calls against any data base in a BMP or DL/I region. In a MSG region, the program name must be the same as the PSB name. In order to execute in a MSG region, the DFSDDLTO program must be given the name or an alias of the PSB named in the IMS/VS definition.

When in a DL/I region, input is read from SYSIN and output is written to PRINTDD.

When in a BMP region, if a symbolic input terminal was specified as the fourth parameter of the EXEC statement, input is obtained from that SMB, and output is sent to the I/O PCB. The name of the I/O PCB can be specified as the fifth parameter of the EXEC statement. If SMB is not specified on the EXEC statement, SYSIN is used for input and PRINTDD is used for output, as in the DL/I region.

In the MSG region, the I/O PCB is used for both input and output unless position 2 of the STATUS statement is either a 1 or an A. In either of these cases, PRINTDD is used for output if the DD card is present in the JCL for that message region. A limit of 50 lines per schedule is sent to the I/O PCB and, after that, PRINTDD is used for output if present. If PRINTDD is not present, the program terminates.

If PRINTDD is specified in either a BMP or MPP region, SNAP output will be routed to the IMS/VS control region PRINTDD DD card.

Because the input is in fixed form, it is difficult to key it from a terminal. For ease of entry, however, Message Format Service (MFS) facilities can be used from a terminal to create the fixed-format input. One way to test DL/I in a message region, using this program, is to first execute another message program

which, based on a message from the terminal, reads control statements stored as a member of a partitioned data set. Insert these control statements into an SMB. This program is then scheduled by IMS/VS to process those transactions. This allows the same control statements to be used to execute in any region type.

SUGGESTIONS ON USING THE DL/I TEST PROGRAM

1. To load a data base:

   This program is applicable for loading small data bases, because all calls and data must be provided to it rather than it generating data. It can be used to load large volume data bases if the control statements were generated as a sequential data set.

2. To display a data base:

   To display a data base, the following sequence of control statements can be used.

```
S 1 2 2 2 1   DBNAME   Display comments and segment
L         GN           DO 1 Get Next
EH      OK             Hold compare, GA, GK, OK, terminate
                       on GB
L   9999 GN            DO 9,999 Get Next calls
```

3. To do regression testing:

   This program can be used for regression testing. By using a known data base, calls can be issued and the results compared to expected results using COMPARE statements. The program then can determine if DL/I calls are being executed correctly. By making the print options of the STATUS statement all twos, only those calls not satisfied properly are displayed.

4. To use as a debugging aid:

   When doing debugging work, usually a print of the DL/I blocks is required. By use of COMPARE statements, the blocks can be displayed at appropriate times. Sometimes the blocks are needed even though the call is executed correctly, such as the call before the failing call. In those cases, a SNAP call can be inserted. This causes the blocks to be displayed even though the call was executed correctly.

   An alternative method of doing a SNAP call when running DFSDDLTO is to use a COMPARE statement after the call, forcing the program to do an unequal compare. For example:

Column 1	Column 3-4	Column 11
E	5P	SNAP

   The SNAP call compares against the segment name of SNAP, gets an unequal compare, and as a result of the SNAP options in columns 3 and 4, snaps the complete I/O buffer pool, the DL/I blocks, and the complete buffer pool.

5. To verify how a call is executed:

   Because it is easy to execute a particular call, this program can be used to verify how a particular call is handled. This is of value when DL/I is suspected of not operating correctly in a specific situation. The calls that are suspected can be issued using this program, and the results examined.

DL/I TEST PROGRAM JCL REQUIREMENTS

JOB This statement initiates the job.

EXEC This statement specifies the program name, or invokes a cataloged procedure. The required format is:

```
PGM=DFSRRC00,PARM='AAA,DFSDDLTO,BBBBBBBB,
    CCCCCCCC,DDDDDDDD'
```

where AAA is the region type and BBBBBBBB is the name of the PSB to be used. Parameters CCCCCCCC and DDDDDDDD are optional, and can be used to specify symbolic input terminal and output terminal names, respectively. Refer to the section "Member Name IMSBATCH" in the IMS/VS System Programming Reference Manual for other parameters that can be used.

STEPLIB DD Defines the partitioned data set named IMSVS.RESLIB. If EXIT routine modules are used, they should be placed into this library or into another PDS concatenated to this library.

IMS DD This statement defines two concatenated data sets. The first DD statement defines the library containing the PSB to be used by the test program. The second DD statement defines the library containing the DBD of the data base to be processed.

database DD This statement references a specific data base. There should be one statement for each data base to be processed. In each statement the ddname must agree with the ddname specified in the DBD.

IEFRDER DD This statement defines the log data set, if one is desired. a dd dummy statement may be used if a log is not desired. One form or the other of this statement is required.

PRINTDD DD This statement defines the output data set for the test program, including displays of control blocks using the SNAP call. It must conform to the OS SNAP data set requirements.

SYSDUMP DD This statement is optional and is used by the test program only when normal termination is not possible.

SYSIN DD This statement defines the control statement input data set.

SYSIN2 DD This is an optional secondary input statement. See the description of "Special Control Statement Formats" for details.

Note: SYSIN may be members of a partioned data set; if they are to be concatenated together, the last statement must be a period (.) or an N statement. This prevents the last statement in the previous concatenation to be used twice.

SAMPLE JCL FOR THE DL/I TEST PROGRAM

```
//JCLSAMP JOB ACCOUNTING,NAME,MSGLEVEL=(1,1),MSGCLASS=3,PRTY=8
//GET EXEC PGM=DFSRRC00,PARM='DLI,DFSDDLT0,PSBNAME'
//STEPLIB  DD   DSN=IMSVS.RESLIB,DISP=SHR
//IMS DD DSN=IMSVS.PSBLIB,DISP=(SHR,PASS)
//    DD DSN=IMSVS.DBDLIB,DISP=(SHR,PASS)
//DDCARD DD DSN=DATASET,DISP=(OLD,KEEP)
//IEFRDER DD DUMMY
//PRINTDD DD SYSOUT=A
//SYSUDUMP DD SYSOUT=A
//SYSIN DD *
S 1 1 1 1      DBNAME
/*
```

INDEX

A

B

D

E

F

G

H

I

J

K

L

Q

R

S

T

U

IMS/VS Version 1
Application Programming:
Designing and Coding
SH20-9026-8

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Previous TNL _____

Fold on two lines, tape, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

SH20-9026-8

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

IMS/VS Version 1 Application Programming: Designing and Coding (File No. S370-50) Printed in U.S.A. SH20-9026-8

IMS/VS Version 1
Application Programming:
Designing and Coding
SH20-9026-8

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Previous TNL _____

Fold on two lines, tape, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

SH20-9026-8

Reader's Comment Form

old and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

old and tape | Please do not staple | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

IMS/VS Version 1 Application Programming: Designing and Coding (File No. S370-50) Printed in U.S.A. SH20-9026-8

IMS/VS Version 1
Application Programming:
Designing and Coding
SH20-9026-8

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Previous TNL _____

Fold on two lines, tape, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

IMS/VS Version 1 Application Programming: Designing and Coding (File No. S370-50) Printed in U.S.A. SH20-9026-8

IMS/VS Version 1 Application Programming: Designing and Coding (File No. S370-50) Printed in U.S.A. SH20-9026-0

SH20-9026-8

IMS/VS Version 1 Application Programming: Designing and Coding (File No. S370-50) Printed in U.S.A. SH20-9026-8

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601