IMS/ESA

Application Programming: Design Guide

Version 6

SC26-8728-05

IMS/ESA

Application Programming: Design Guide

Version 6

SC26-8728-05

Note

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

Sixth Edition (July 2000) (Softcopy Only)

This edition replaces and makes obsolete the previous edition, SC26–8728–04. This edition is available in softcopy format only. The technical changes for this edition are summarized under “Summary of Changes” on page xix and are indicated by a vertical bar to the left of a change.

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

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

IBM Corporation, BWE/H3

P.O. Box 49023

San Jose, CA, 95161-9023

U.S.A.

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

Note

Note

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

Contents

Notices

Programming Interface Information

This book is intended to help the programmer design application programs. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by IMS.

General-use programming interfaces allow the customer to write programs that obtain the services of IMS.

However, this book also documents Product-sensitive Programming Interface and Associated Guidance Information and Diagnosis, Modification or Tuning Information provided by IMS.

Product-sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of IMS. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive programming interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Product-sensitive programming interface

Product-sensitive Programming Interface and Associated Guidance Information...

End of Product-sensitive programming interface

Diagnosis, Modification or Tuning Information is provided to help the programmer perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of IMS.

Attention: Do not use this Diagnosis, Modification or Tuning Information as a programming interface.

Diagnosis, Modification or Tuning Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Product-sensitive programming interface

Diagnosis, Modification or Tuning Information...

End of Product-sensitive programming interface

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Advanced Function Printing
AFP
C/MVS
CICS
CICS/ESA
DATABASE 2
DB2
IBM
IMS
IMS/ESA
IPDS
Language Environment
MVS
MVS/ESA
MVS/XA
RACF
VisualGen
VTAM

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Product Names

In this book, the following licensed programs have shortened names:

- “C/C++ for MVS/ESA” is referred to as either “C/MVS” or “C++/MVS”.
- “COBOL for MVS & VM” is referred to as “COBOL”.
- “DB2 for MVS/ESA” is referred to as “DB2”.

- “IBM DataAtlas for OS/2” is referred to as “DataAtlas”.
- “Language Environment for MVS & VM” is referred to as “Language Environment”.
- “PL/I for MVS & VM” is referred to as “PL/I”.

Preface

This book is a guide for designing application programs for IMS/ESA (hereafter called IMS). It tells application programmers how to identify application data and analyze requirements for application processing. This book describes other tasks such as gathering requirements for database and message processing options and testing an application program.

Summary of Contents

The chapters in this book are applicable in both the IMS and CICS environments unless otherwise noted. Chapter content is as follows:

Introduction

Chapters 1 and 2 explain the basics for designing an application. The first chapter defines database concepts and terms, and the second chapter introduces the tasks of application design. These two introductory chapters also explain how to identify required application data and how to design a local view to describe that data.

Designing the Application

Chapters 3 through 6 describe the tasks involved in designing an application. These tasks include choosing the type of application program you need for both the IMS and CICS environments and gathering the information for the database administrator and system administrator.

Implementing the Design

Chapters 7 through 10 include considerations for using your application after the specifications have been developed. These considerations are using LU 6.2/APPC for distributing your application throughout the network, testing your application program after you have finished coding it, and documenting additional information about your program.

Prerequisite Knowledge

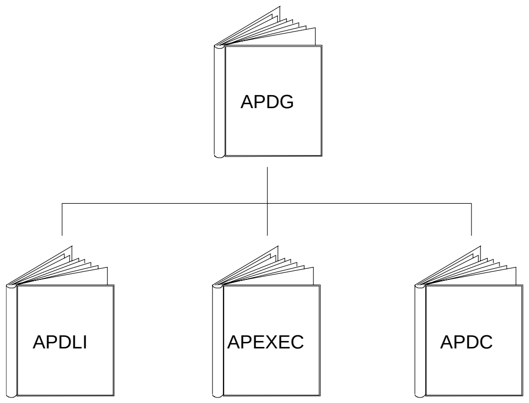
Before using this manual, you should understand basic IMS concepts and the IMS environments. IBM offers a wide variety of classroom and self-study courses to help you learn IMS. For a complete list, see the IMS home page on the World Wide Web at: http://www.ibm.com/ims

You will also find descriptions of basic IMS concepts and the IMS environments at the above Web site.

If you are a CICS user, you should understand a similar level of information for CICS. The IMS concepts explained in this manual are limited to those concepts pertinent to designing application programs. You should also know how to use COBOL, PL/I, Assembler language, Pascal, or C language.

How to Use This Book

This book is one of several books documenting the IMS application programming task. This book is the introductory book. The complete package of application programming materials is as follows:



- *IMS/ESA Application Programming: Design Guide* (APDG), the book you are currently reading, is the introductory application programming book and is also the place to find information common to all of the application programming environments.
- *IMS/ESA Application Programming: Database Manager* (APDLI) describes how to write an application program to process a database using DL/I calls. This book applies to both IMS and CICS environments.
- *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS* (APEXEC) describes how to write an application program to process the database using EXEC DLI commands.
- *IMS/ESA Application Programming: Transaction Manager* (APDC) describes how to write an application program to process messages using DC calls.

For definitions of terms used in this manual and references to related information in other manuals, please see *IMS/ESA Master Index and Glossary*.

Change Indicators

Technical changes are indicated in this publication by a vertical bar (|) to the left of the changed text. If a figure has changed, a vertical bar appears to the left of the figure caption.

Syntax Diagrams

The following rules apply to the syntax diagrams used in this book:

Arrow symbols

Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

►►───	Indicates the beginning of a statement.
───►	Indicates that the statement syntax is continued on the next line.
►───	Indicates that a statement is continued from the previous line.
───►◄	Indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the ►─── symbol and end with the ───► symbol.

Conventions

- Keywords, their allowable synonyms, and reserved parameters, appear in uppercase for MVS and OS/2 operating systems, and lowercase for UNIX operating systems. These items must be entered exactly as shown.

- Variables appear in lowercase italics (for example, *column-name*). They represent user-defined parameters or suboptions.
- When entering commands, separate parameters and keywords by at least one blank if there is no intervening punctuation.
- Enter punctuation marks (slashes, commas, periods, parentheses, quotation marks, equal signs) and numbers exactly as given.
- Footnotes are shown by a number in parentheses, for example, (1).
- A ƀ symbol indicates one blank position.

Required items

Required items appear on the horizontal line (the main path).

```
►►──REQUIRED_ITEM──────────────────────────────────────────────────►◄
```

Optional Items

Optional items appear below the main path.

```
►►──REQUIRED_ITEM──┬───────────────┬───────────────────────────────►◄
                   └─optional_item─┘
```

If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

```
                   ┌─optional_item─┐
►►──REQUIRED_ITEM──┴───────────────┴───────────────────────────────►◄
```

Multiple required or optional items

If you can choose from two or more items, they appear vertically in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.

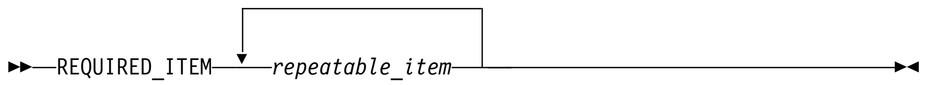
```
►►──REQUIRED_ITEM──┬─required_choice1─┬────────────────────────────►◄
                   └─required_choice2─┘
```

If choosing one of the items is optional, the entire stack appears below the main path.

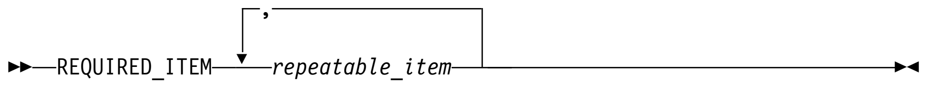
```
►►──REQUIRED_ITEM──┬──────────────────┬────────────────────────────►◄
                   ├─optional_choice1─┤
                   └─optional_choice2─┘
```

Repeatable items

An arrow returning to the left above the main line indicates that an item can be repeated.



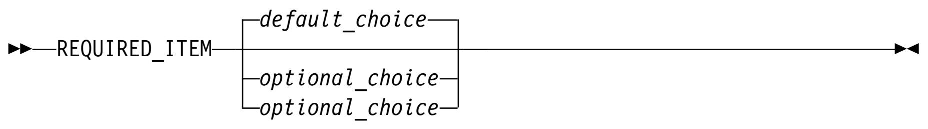
If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can specify more than one of the choices in the stack.

Default keywords

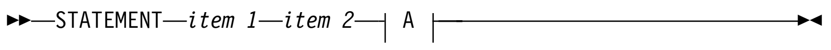
IBM-supplied default keywords appear above the main path, and the remaining choices are shown below the main path. In the parameter list following the syntax diagram, the default choices are underlined.



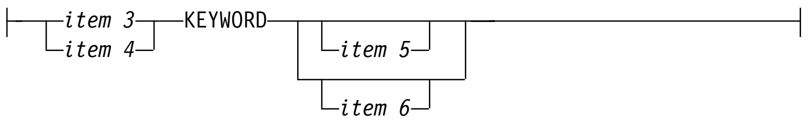
IMS-specific syntax information

Fragments

Sometimes a diagram must be split into fragments. The fragments are represented by a letter or fragment name, set off like this: | A |. The fragment follows the end of the main diagram. The following example shows the use of a fragment.



A:



Substitution-block

Sometimes a set of several parameters is represented by a substitution-block such as **<A>**. For example, in the imaginary /VERB command you could enter `/VERB LINE 1`, `/VERB EITHER LINE 1`, or `/VERB OR LINE 1`.

```
►►──/VERB──┬─────┬──LINE──line#──────────────────────────────►◄
           └─<A>─┘
```

where <A> is:

```
►►──┬─EITHER─┬──────────────────────────────────────────────►◄
    └─OR─────┘
```

Parameter endings

Parameters with number values end with the symbol '#', parameters that are names end with 'name', and parameters that can be generic end with '*'.

```
►►──/MSVERIFY──┬─MSNAME──msname──┬──────────────────────────►◄
               └─SYSID──sysid#───┘
```

The MSNAME keyword in the example supports a name value and the SYSID keyword supports a number value.

Summary of Changes

Changes to the Current Edition of the Book for Version 6

This edition, which is in softcopy format only, includes technical and editorial changes.

Changes to This Book for Version 6

This book contains new and changed information about the following functions:

- Distributed sync point
- Shared queues
- Nondiscardable Messages Enhancement

Editorial changes have been made throughout the book.

Library Changes for Version 6

The IMS/ESA Version 6 library differs from the IMS/ESA Version 5 library in these major respects:

- *IMS/ESA Common Queue Server Guide and Reference*

  This new book describes the IMS Common Queue Server (CQS).
- *IMS/ESA DBRC Guide and Reference*

  This new book describes all the functions of IMS Database Recovery Control (DBRC).
- The IMS Application Programming summary books (*IMS/ESA Application Programming: Database Manager Summary*, *IMS/ESA Application Programming: Transaction Manager Summary*, and *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS Summary*) are no longer included with the IMS library.
- The Softcopy Master Index is not included.
- All information about IRLM 1.5 and data sharing using IRLM 1.5 has been removed from the IMS V6 books. If you use IRLM 1.5, and want to migrate to using IRLM 2.1 and Sysplex data sharing, see *IMS/ESA Release Planning Guide*.
- The chapter that was titled ″Database Control (DBCTL) Interface″ in the *IMS/ESA Customization Guide* has been revised for Open Database Access (ODBA) and moved to ″Appendix A, Using the Database Resource Adapter (DRA)″ in the *IMS/ESA Application Programming: Database Manager*.

Chapter 1. Concepts and Terminology

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

In this Chapter:

Storing and Processing Information in a Database

This section describes how storing data in a database is different from other ways of storing data. The advantages of storing and processing data in a database are that all of the data needs to appear only once, and that each program must 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 database.

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 must store a lot of data in several places at once. The problem with this is that redundant data takes up space that could be used for something else and makes maintaining the files complex.

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 who visits the clinic, such as:

  Identification number
  Name
  Address
  Illnesses
  Date of each illness
  Date patient came to clinic for treatment
  Treatment given for each illness
  Doctor that prescribed treatment
  Charge for treatment

- The accounting department also keeps information about each patient. The information that the accounting department might keep for each patient is:

  Identification number
  Name
  Address
  Charge for treatment
  Amount of payments

- The information that the ophthalmology department might keep for each patient is:

Identification number

Name

Address

Illnesses relating to ophthalmology

Date of each illness

Names of members in patient's household

Relationship between 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 a department changes a piece of data, you must update the same data in each separate file. Therefore, it is difficult to keep the data in each department's files current. Current data might be in one file while old data is in another file.

Storing Data in a Combined File

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

Identification number

Name

Address

Illnesses

Date of each illness

Date patient came to clinic for treatment

Treatment given for each illness

Doctor that prescribed treatment

Charge for treatment

Amount of payments

Names of members in patient's household

Relationship between patient and each household member

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

Storing Data in a Database

Storing data in a database gives you the advantages of both separate files and combined files: all the data appears only once, and each program has access to the data that it needs. This means that:

- When you update a field, you do it in one place only.

- Because you store each piece of information only in one place, you cannot have an updated version of the information in one place and an out-of-date version in another place.
- Each program accesses only the data it needs.
- You can prevent programs from accessing private information.

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

- If you change the format of part of a database record, the change does not affect the programs that do not use the changed information.
- Programs are not affected by how the data is stored.

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

Database Hierarchies

In an IMS DB, 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.

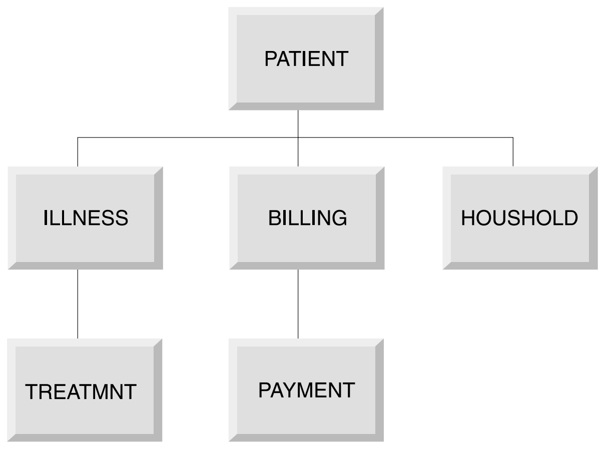


Figure 1. Medical Database Hierarchy

IMS connects the pieces of information in a database record by defining the relationships between the pieces of information that relate to the same subject. The result is a database hierarchy.

Example: In the medical database, the data that you are 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 is not associated with a patient. To be meaningful, ILLNESS, TREATMNT, BILLING, PAYMENT, and HOUSHOLD must always be associated with one of the clinic's patients.

You keep five kinds of information about each patient. The information about the patient's illnesses, billings, and household depends directly on the patient.

Storing and Processing Information

Information about the patient's treatment depends on the patient's illness, and information about the patient's payments depends on the patient's billings.

Each piece of data represented in Figure 1 on page 3 is called a *segment* in the hierarchy. Each segment contains one or more *fields* of information. The PATIENT segment, for example, contains all the information that relates strictly to the patient: the patient's identification number, name, and address.

Definitions: A *segment* is the smallest unit of data that an application program can retrieve from the database. A *field* is the smallest unit of a segment.

The PATIENT segment in the medical database is the *root segment*. The segments below the root segment are the *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 dependents, because they are at a lower level in the hierarchy.

A database record is a single root segment (root segment *occurrence*) and all of its dependents. In the medical example, a database record is all the information about one patient.

Definitions: A *root segment* is the highest-level segment. A dependent is a segment below a root segment. A root segment occurrence is a database record and all of its dependents.

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

Your Program's View of the Data

IMS uses two kinds of control blocks to enable application programs to be independent of the way in which you store the data in the database, the database description (DBD) and the database program communication block (DB PCB).

Database Description (DBD)

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

For example, the DBD for the medical database hierarchy shown in Figure 1 on page 3 would describe the physical structure of the hierarchy and each of the six segment types in the hierarchy: PATIENT, ILLNESS, TREATMNT, BILLING, PAYMENT, and HOUSHOLD.

Related Reading: For more information on generating DBDs, see *IMS/ESA Utilities Reference: Database Manager*.

Database Program Communication Block (DB PCB)

A *database program communication block (DB PCB)* defines an application program's view of the database. An application program often needs to process only some of the segments in a database. A PCB defines which of the segments in the database the program is allowed to access—which segments the program is sensitive to. The data structures that are available to the program contain only segments that the program is sensitive to. The PCB also defines how the application program is allowed to process the segments in the data structure: whether the program can only read the segments, or whether it can update them as well.

To obtain the highest level of data availability, your PCBs should request the fewest number of sensitive segments and the least capability needed to complete the task.

All the DB PCBs for a single application program are contained in a program specification block (PSB). A program might use only one DB PCB (if it processes only one data structure) or it might use several DB PCBs, one for each data structure.

Related Reading: For more information on generating PSBs, see *IMS/ESA Utilities Reference: Database Manager*.

Figure 2 illustrates the concept of defining a view for an application program. 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.

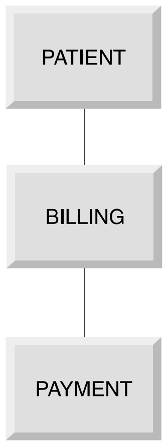


Figure 2. Accounting Program’s View of the Database

A program that updates the database 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 on page 6 for this program.

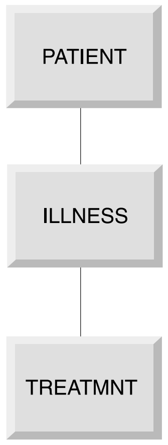


Figure 3. Patient Illness Program's View of the Database

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

An application program processes only the segments in a database that it requires; therefore, if you change the format of a segment that is not processed, you do not 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 database, a program can also be sensitive to only certain fields within a segment. If you change a segment or field that the program is not sensitive to, it does not affect the program. You define segment and *field-level* sensitivity during PSBGEN.

Definition:*Field-level* sensitivity is when a program is sensitive to only certain fields within a segment.

Related Reading: For more information, see *IMS/ESA Administration Guide: Database Manager*.

Processing a Database Record

To process the information in the database, your application program communicates with IMS in three ways:

- Passing control—IMS passes control to your application program through an entry statement in your program. Your program returns control to IMS when it has finished its processing.

  When you are running a CICS online program, CICS passes control to your application program, and your program schedules a PSB to make IMS requests. Your program returns control to CICS. If you are running a batch or BMP program, IMS passes control to your program with an existing PSB scheduled.
- Communicating processing requests—You communicate processing requests to IMS in one of two ways:
  – In IMS, you issue DL/I calls to process the database.
  – In CICS, you can issue either DL/I calls or EXEC DLI commands. EXEC DLI commands more closely resemble a higher-level language than do DL/I calls.
- Exchanging information using DL/I calls—Your program exchanges information in two areas:

– A DL/I call reports the results of your request in a control block and the AIB communication block when using one of the AIB interfaces. For programs written using DL/I calls, this control block is the database program communication block (DB PCB). For programs written using EXEC DLI commands, this control block is the DLI interface block (DIB). The contents of the DIB reflect the status of the last DL/I command executed in the program. Your program includes a mask of the appropriate control block and uses this mask to check the results of the request.
– When you request a segment from the database, IMS returns the segment to your I/O area. When you want to update a segment in the database, you place the new value of the segment in the I/O area.

An application program can read and update a database. When you update a database, you can replace, delete, or add segments. In IMS, you indicate in the DL/I call the segment you want to process, and whether you want to read or update it. In CICS, you can indicate what you want using either a DL/I call or an EXEC command.

A Look at the Tasks Ahead of You

The following tasks are involved in developing an IMS application, and the programs that are part of the application.

Designing the Application

Application program design varies from installation to installation, and from one application to another. Therefore, this book does not try to cover the early tasks that are part of designing an application program. Instead, it covers only the tasks that you are concerned with after the early specifications for the application have been developed. These subtasks are:

Analyzing Application Data Requirements

Two important parts of application design are defining the data that each of the business processes in the application requires and designing a local view for each of the business processes. “Chapter 2. Introduction to Application Design” on page 9 explains these tasks.

Analyzing Application Processing Requirements

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 that is available with different types of application programs. “Chapter 3. Analyzing IMS Application Processing Requirements” on page 29 and “Chapter 4. Analyzing CICS Application Processing Requirements” on page 51 explain the processing and application requirements that each type of program satisfies.

Gathering Requirements for Database Options

You then need to look at the database options that can most efficiently meet the requirements, and gather information about your application’s data requirements that relates to each of the options. “Chapter 5. Gathering Requirements for Database Options” on page 69 explains these options and helps you gather information about your application that will be helpful to the database administrator in choosing these options.

Gathering Requirements for Message Processing Options

If your application communicates with terminals and other application programs, look at the message processing options and the requirements they satisfy. “Chapter 6. Gathering Requirements for Message Processing Options” on page 91 page 91 explains the IMS message processing options and helps you to gather information about your application that is helpful in choosing these options.

Related Reading: If you are designing a CICS/ESA application, you will find information about gathering requirements in *CICS/ESA Application Programming Guide*.

Developing Specifications

Developing specifications involves defining what your application is to do, and how it will be done. This task depends completely on the specific application and installation, and is thereforenot described in this book.

Implementing the Design

When the specifications for each of the programs in the application have been developed, you can structure and code the programs according to those specifications. The subtasks are:

Writing the Database Processing Part of the Program

When the program design is complete, you can structure and code your requests and data areas based on the programming specifications that have been developed.

Related Reading: The following books contain information about this task:

– *IMS/ESA Application Programming: Database Manager*

– *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*

Writing the Message Processing Part of the Program

If you are writing a program that communicates with terminals and other programs, you need to structure and code the message processing part of the program.

Related Reading: *IMS/ESA Application Programming: Transaction Manager*

Analyzing APPC/IMS Requirements

The LU 6.2 feature of IMS TM enables your application to be distributed throughout the network. “Chapter 7. Application Design for APPC” on page 103 tells how to use LU 6.2 and the IMS TM application programs. This chapter describes the considerations for modifying these application programs to communicate with other application programs and shows the results of conversations.

Testing an Application Program

When you have completed coding your program, you test it by itself and then as part of a system. “Chapter 9. Testing an IMS Application Program” on page 135 and “Chapter 10. Testing a CICS Application Program” on page 155 give you some guidelines.

Documenting an Application Program

Documenting a program is an ongoing process throughout the project and is most effective when done incrementally. When the program is completely tested, additional information remains that you need to supply for people who use and maintain your program. “Chapter 12. Documenting an Application Program” on page 169 gives you some suggestions about the information you should record about your program.

Chapter 2. Introduction to Application Design

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. Application design ends when each of the parts of the application system—for example, the programs, the databases, the display screens, and the message formats—have been designed.

An Overview of Application Design

The application design process varies from installation to installation and from application to application. The overview that is 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.

The purpose of this overview is to give you a frame of reference so that you can 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 individuals involved in each task, and their titles, might differ according to the installation. The tasks are as follows:

- Use installation standards

  Throughout the design process, be 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 databases and terminals)
  – Formats for screens and messages
  – Control of and access to the database
  – Programming and conventions (for common routines and macros)

  Setting up standards in these areas is usually an ongoing task that is the responsibility of database and system administrators.

- Follow your installation's security standards

  Security protects the installation's resources from unauthorized access and use. As with defining standards, designing an adequate security system is often an ongoing task. As an application is modified or expanded, often the security must 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 system administration. These standards are based on the requirements of the applications at the installation.

  Some security concerns are:

  – Access to and use of the databases
  – Access to terminals
  – Distribution of application output

- – Control of program modification
- – Transaction and command entry.

Related Reading:“Providing Data Security” on page 84 and “Identifying Online Security Requirements” on page 91 give some suggestions about the kind of information that you can gather concerning your application’s security requirements. This information can be helpful to database administration and system administration in implementing database and data communications security.

- Define 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. After you have listed the required data, you can name the data and document it. “Identifying Application Data” on page 11 describes these parts of data definition.

- Provide input for database design

  To design a database that meets the requirements of all the applications that will process it, 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” on page 16 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.

- Design application programs

  When 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. The specifications you develop for the programs should include:

  - – Security requirements
  - – Input and output data formats and volumes
  - – Data verification and validation requirements
  - – Logic specifications
  - – Performance requirements
  - – Recovery requirements
  - – Linkage requirements and conventions
  - – Data availability considerations

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

- Document 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 is 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 database, your IMS system, and the programs in the application—especially if any part of the application must be changed in the future. Documenting application design is done most thoroughly when it is done during the design process, instead of at the end of it.

- Convert an existing application

One of the main aspects in converting an existing application to IMS 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:

– Record layouts of all records used by the application
– Number of data element occurrences for each data element
– Structure of any existing related databases

Identifying Application Data

Two important aspects of application design are identifying the application data and describing the data that a particular business process requires.

One of the steps of identifying application data is to thoroughly understand the processing the user wants performed. You need to understand the input data and the required output data in order to define the data requirements of the application. You also need to understand the business processes that are involved in the 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 analyzing the required application data, you can categorize the data as either an entity or a data element.

Definitions: An *entity* is anything about which information can be stored. A *data element* is the smallest named unit of data pertaining to an entity. It is information that describes the entity.

Example: In an education application, "students" and "courses" are both entities; these are two subjects about which you collect and process data. Table 1 shows some data elements that relate to the student and course entities.

Table 1. Entities and Data Elements

Entity	**Data Elements**
Student	Student Name
	Student Number
Course	Course Name
	Course Number
	Course Length

When you store this data in an IMS database, groups of data elements are potential segments in the hierarchy. Each data element is a potential field in that segment.

Listing Data Elements

Example: To identify application data, consider a company that provides technical education to its customers. The education company has one headquarters office, called HQ, and several local education centers, called Ed Centers.

Identifying Application Data

A class is a single offering of a course on a specific date at a particular Ed Center. One course might have several offerings at different Ed Centers; each of these is a separate class. HQ is responsible for developing all 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 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 Figure 4.

```
CHICAGO                                                    1/04/96

     TRANSISTOR THEORY                      41837
     10 DAYS
     INSTRUCTOR(S): BENSON, R.J.            DATE: 1/14/96

   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 4. Current Roster

To list the data elements for a particular business process, look at the required output. The current roster shown in Figure 4 is the roster for the class, "Transistor Theory" to be given in the Chicago Ed Center, starting on January 14, 1996, for ten days. Each course has a course code associated with it—in this case, 41837. The code for a particular course is always the same. For example, if Transistor Theory is also given in New York, the course code is still 41837. The roster also gives the names of the instructors who are teaching the course. Although the example only shows one instructor, a course might 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 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 (1/04/96). The current date is an example of data that is only output data; it is generated by the operating system and is not stored in the database.

The bottom-left corner gives a summary of the class status. This data is not included in the input data. These values are determined by the program during processing.

When you list the data elements, abbreviating them is helpful, because you will be referring to them frequently when you design the local view.

The data elements list for current roster is:

EDCNTR	Name of Ed Center giving class
DATE	Date class starts
CRSNAME	Name of course
CRSCODE	Course code
LENGTH	Length of course
INSTRS	Names of instructors teaching class
STUSEQ#	Student’s sequence number
STUNAME	Student’s name
CUST	Name of student’s company
LOCTN	Location of student’s company
STATUS	Student’s status in class—confirmed, wait list, or canceled
ABSENCE	Number of days student was absent
GRADE	Student’s grade for the course

After you have listed the data elements, choose the major entity that these elements describe. In this case, the major entity is class. Although a lot of information exists about each student and some information exists about the course in general, together all this information relates to a specific class. If the information about each student (for example, status, absence, grade) is not related to a particular class, 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 mean nothing unless you know what class they describe.

Naming Data Elements

Some of the data elements your application uses might already exist and be named. After you have listed the data elements, find out if any of them exist by checking with the database administrator (DBA) at your installation.

Before you start naming data elements, be aware of the naming standards at your installation. When you name data elements, use the most descriptive names possible. Remember that, because other applications probably use at least some of the same data, the names should mean the same thing to everyone. Try not to limit the name’s meaning only to your application.

Recommendation: 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, to be understood, must be seen in the context of a particular application.

One of the problems with using local names is that you can develop synonyms, two names for the same data element.

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.

When you choose data element names, use qualifiers so that each name can mean only one thing.

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. This is called a *homonym*. You can solve the homonym 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#).

Definition: A *homonym* is one word for two different things.

Choose data element names that identify the element and describe it precisely. Make your data element names:

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 have determined what data elements a business process requires, record as much information about each of the data elements as possible. This information is useful to the DBA. Be aware of any standards 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 varies 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 are familiar and also to those who are unfamiliar with the application.

The length of the data element

This determines segment size and segment format.

The character format

The programmer needs to know if the data is alphanumeric, hexadecimal, packed decimal, or binary.

The range of possible values for the element

This is important for validity checking.

The default value

The programmer also needs this information.

The number of data element occurrences

This information helps the DBA to determine the required space for this data, and it affects performance considerations.

How the business process affects the data element

Whether the data element is read or updated determines the processing option that is coded in the PSB for the application program.

You should also record control information about the data. Such information should address the following questions:

- What action should the program take when the data it attempts to access is not available?
- If the format of a particular data element changes, which business processes does that affect? For example, if an education database has as one of its data elements a five-digit code for each course, and the code is changed to six digits, which business processes does 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 Table 2. The amount and type of data that you record depends on the standards at your installation.

Table 2. Example of Data Elements Information Form

ID #	Data Element Name	Length	Char. Format	Allowed Values	Null Values	Default Value	Number of Occurrences
5	Course Code	5 bytes	Hexa- decimal	0010090000	00000	N/A	There are 200 courses in the curriculum. An average of 10 are new or revised per year. An average of 5 are dropped per year.
25	Status	4 bytes	Alpha- numeric	CONF WAIT CANC	blanks	WAIT	1 per student
36	Student Name	20 bytes	Alpha- numeric	Alpha only	blanks	N/A	There are 3–100 students per class with an average of 40 per class.

Identifying Application Data

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 a particular data element already exists, and if it does, its characteristics. With the DB/DC Data Dictionary, and its successor, DataAtlas (a part of the IBM VisualGen Team Suite), you can determine online what segments exist in a particular database and what fields those segments contain. You can use either tool to create reports involving the same information.

Related Reading: For information on these products, see *OS/VS DB/DC Data Dictionary Applications Guide*, *Introducing VisualGen*, and *VisualGen: Running Application on MVS*.

Designing a Local View

A local view is a description of the data that an individual business process requires. It includes the following:

- A list of the data elements
- A conceptual data structure that shows how you have grouped data elements by the entities that they describe
- The relationships between each of the groups of data elements

Definition: A group of data elements is called a *data aggregate*. When you have grouped data elements by the entity they describe, you can determine the relationships between the data 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. This process, called *data structuring*, is a way to analyze the relationships among the data elements a business process requires, not a way to design a database. The decisions about segment formats and contents belong to the DBA. The information you develop is input for designing a database.

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

Grouping Data Elements into Hierarchies

The data elements that describe a data aggregate, the student, might be represented by the descriptive names STUSEQ#, STUNAME, CUST, LOCTN, STATUS, ABSENCE, and GRADE. We call this group of data elements the student data aggregate.

Data elements have values and names. In the student data elements example, 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—they describe all the students in the class 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 uniquely identify that group. This is the data aggregate's *key*, which is the data element or group of data elements in the aggregate that uniquely identifies the aggregate. Sometimes you must use more than one data element for the key in order to uniquely identify the aggregate.

By following the three steps explained in this section, you can develop a conceptual data structure for a business process's data. However, you are not developing the logical data structure for the program that performs the business process. 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 each data element with its controlling keys.

Step 1. Isolating Repeating Data Elements: Look at a single occurrence of the data aggregate. Table 3 shows what this looks like for the class aggregate.

Table 3. Single Occurrence of Class Aggregate

Data Element	Class Aggregate Occurrence
EDCNTR	CHICAGO
DATE(START)	1/14/96
CRSNAME	TRANSISTOR THEORY
CRS CODE	41837
LENGTH	10 DAYS
INSTRS	multiple
STUSEQ#	multiple
STUNAME	multiple
CUST	multiple
LOCTN	multiple
STATUS	multiple
ABSENCE	multiple
GRADE	multiple

The data elements defined as multiple are the elements that repeat. Separate the data elements that repeat by shifting 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 5 on page 18. In that figure, the data elements in each box form an aggregate. The entire figure depicts a data structure.

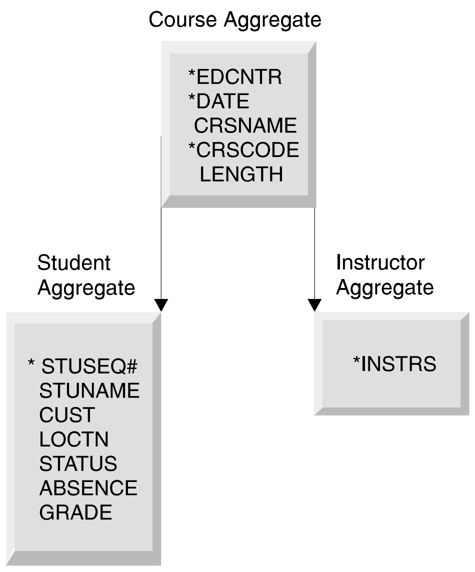


Figure 5. Current Roster Step 1

The asterisks in Figure 5 identify 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 a 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.

In the example shown in Figure 5, the student aggregate and instructor aggregate are both dependents of the course aggregate. A dependent aggregate's key includes the concatenated keys of all the aggregates above the dependent aggregate. This is because a dependent's key does not 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 the information about the student associated with that number. This number would be meaningless, however, if it were not associated with a particular course. But, because the key for the student aggregate is made up of Ed Center, date, and course code, you would know which class the student was in.

Figure 5 shows these aggregates with the following keys:

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

Step 2. Isolating Duplicate Aggregate Values: Look at multiple occurrences of the aggregate—in this case, the values you might have for two classes. Table 4 shows multiple occurrences of the same data elements. As you look at this figure, check for duplicate values. Remember that both occurrences describe one course.

Table 4. Multiple Occurrences of Class Aggregate

Data Element List	Occurrence 1	Occurrence 2
EDCNTR	CHICAGO	NEW YORK
DATE(START)	1/14/96	3/10/96
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

The data elements defined as multiple are the data elements that repeat. The values in these elements are not the same. The aggregate is always unique for a particular class.

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

In Table 4, CRSNAME, CRSCODE, and LENGTH are the fields that 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 6 on page 20.

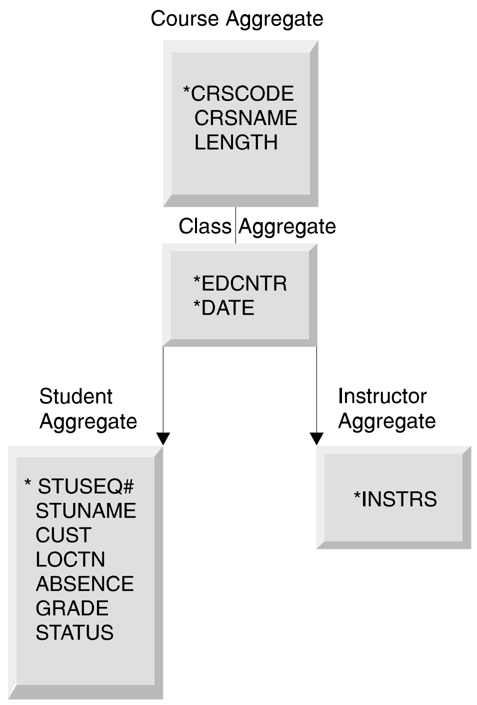


Figure 6. Current Roster Step 2

Step 3. Grouping Data Elements with their Controlling Keys: This 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 depends only on part of the key, separate the data element along with the partial key on which it depends.

In this example, CUST and LOCTN do not depend on the STUSEQ#. They are related to the student, but they do not 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, either. They are separate from all of these things. Because 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. The following figure shows what the structure looks like when you separate CUST and LOCTN.

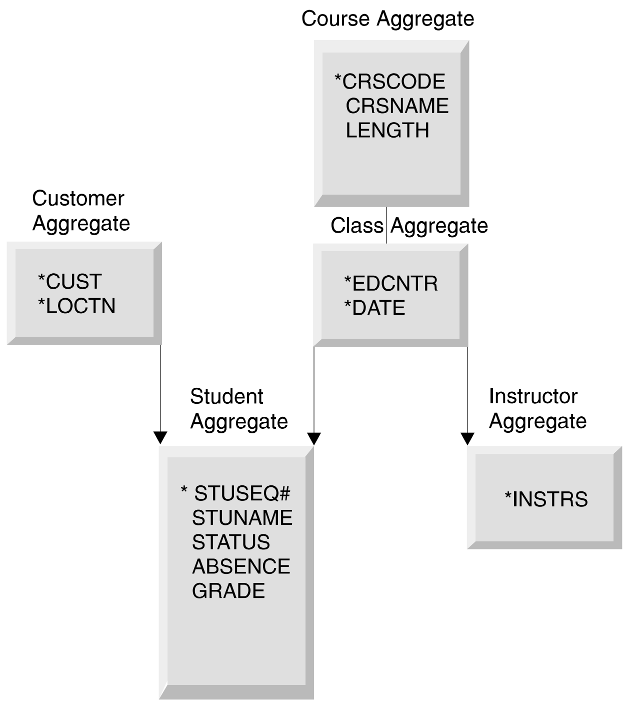


Figure 7. Current Roster Step 3

Figure 7 shows these aggregates and the following keys:

Course aggregate	CRSCODE
Class aggregate	CRSCODE, EDCNTR, DATE
Customer aggregate	CUST, LOCTN
Student aggregate	(when viewed from the customer aggregate in Figure 7, instead of from the course aggregate, in Figure 6 on page 20) CUST, LOCTN, STUSEQ, CRSCODE, EDCNTR, DATE
Instructor aggregate	CRSCODE, EDCNTR, DATE, INSTRS

Deciding on the arrangement of the customer and location information is part of designing a database. Data structuring should separate any inconsistent data elements from the rest of the data elements.

Determining Mappings

When 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 you record mappings is that they reflect relationships between segments in the data structure that you have developed. If you store this information in an IMS database, the DBA can construct a database hierarchy that satisfies all the local views, based on the mappings. In determining mappings, it is easier to refer to the data aggregates by their keys, rather than by their collected data elements.

The two possible relationships between any two data aggregates are:

- One-to-many

For each segment A, one or more occurrences of segment B exist. For example, each class maps to one or more students.

Mapping notation shows this in the following way:

Class $\longleftarrow\!\!\!\twoheadrightarrow$ **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 there each child can be associated with more than one parent.

  Related Reading: For more information about analyzing data requirements, see *IMS/ESA Administration Guide: Database Manager*.

  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 the IMS full-function database, you can solve this kind of processing conflict by using secondary indexing or logical relationships. “Understanding How Data Structure Conflicts Are Resolved” on page 76 explains how to use these tools.

The mappings for the current roster are:

- **Course** $\longleftarrow\!\!\!\twoheadrightarrow$ **Class**

  For each course, there might be several classes scheduled, but a class is associated with only one course.
- **Class** $\longleftarrow\!\!\!\twoheadrightarrow$ **Student**

  A class has many students enrolled in it, but a student might be in only one class offering of this course.
- **Class** $\longleftarrow\!\!\!\twoheadrightarrow$ **Instructor**

  A class might have more than one instructor, but an instructor only teaches one class at a time.
- **Customer/location** $\longleftarrow\!\!\!\twoheadrightarrow$ **Student**

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

Local View Examples

This section presents three more examples of designing a local view: The Schedule of Classes, the Instructor Skills Report, and the Instructor Schedules. It does not explain how to design 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 the data elements within one aggregate have the same controlling key. Isolate any that do not.

3. Determine the mappings between the data aggregates in the data structure you have developed.

Schedule of Courses

HQ keeps a schedule of all 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 8.

```
                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 8. Schedule of Classes

1. Gather the data. Table 5 lists the data elements and two occurrences of the data aggregate.

Table 5. Class Schedule Data Elements

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

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 by shifting them to a lower level as shown in Figure 9 on page 24.

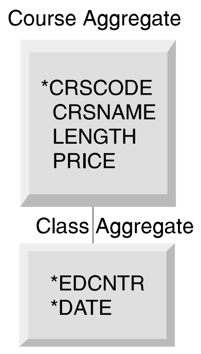


Figure 9. Class Schedule Step 1

b. Next, isolate duplicate values in two occurrences of the data aggregate by shifting the data elements to a higher level.

   This data aggregate does not contain duplicate values.

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.

   Controlling keys are as follows:

Course aggregate	CRSCODE
Class aggregate	CRSCODE, EDCNTR, DATE

3. When you have 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 Ed Center needs to print a report giving the courses that its instructors are qualified to teach. The report is to be in the format shown in Figure 10.

```
       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 10. Instructor Skills Report

1. Gather the data. Table 6 on page 25 lists the data elements and two occurrences of the data aggregate.

Table 6. Instructor Skills Data Elements

Data Elements	Occurrence 1	Occurrence 2
INSTR	REYNOLDS, P.W.	MORRIS, S. R.
CRSCODE	multiple	multiple
CRSNAME	multiple	multiple

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 by shifting to a higher level as shown in Figure 11.

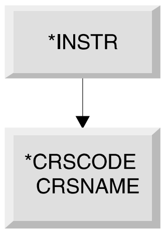


Figure 11. Instructor Skills Step 1

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

      No duplicate values are 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. There are no changes to this data structure.
3. 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 12 shows what the report is to look like.

```
                     INSTRUCTOR SCHEDULES

INSTRUCTOR           COURSE         CODE     ED CENTER     DATE
------------------------------------------------------------------
BENSON, R. J.        TRANS THEORY   41837    CHICAGO       1/14/96
MORRIS, S. R.        TRANS THEORY   41837    NEW YORK      3/10/96
                     LOGIC THEORY   41852    BOSTON        3/27/96
                     CIRCUIT DES    41850    CHICAGO       4/21/96
REYNOLDS, B. H.      MICRO PROG     41840    NEW YORK      2/25/96
                     CIRCUIT DES    41850    LOS ANGELES   3/10/96
```

Figure 12. Instructor Schedules

1. Gather the data. Table 7 lists the data elements and two occurrences of the data aggregate.

Table 7. Instructor Schedules Data Elements

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

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 by shifting data elements to a lower level as shown in Figure 13.

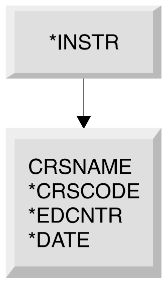


Figure 13. Instructor Schedules Step 1

   b. Isolate duplicate values in two occurrences of the data aggregate by shifting data elements to a higher level as shown in Figure 14.

      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.

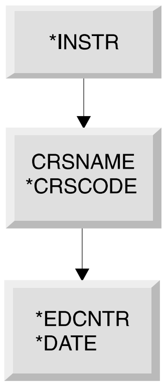


Figure 14. Instructor Schedules Step 2

   c. Group data elements with their controlling keys.

      All data elements are grouped with their controlling keys in the current data structure. No changes to the current data structure are required.

3. Determine the mappings for the data aggregates.

   The mappings for this local view are:

$$\begin{tikzcd}
\textbf{Instructor} \arrow[r, <->>] & \textbf{Course} \\
\textbf{Course} \arrow[r, <->>] & \textbf{Class}
\end{tikzcd}$$

Combining the requirements of the three examples presented in this chapter and designing a hierarchic structure for the database based on these requirements requires analysis of data requirements.

Related Reading: For more information on analyzing data requirements, see *IMS/ESA Administration Guide: Database Manager*.

Chapter 3. Analyzing IMS Application Processing Requirements

This chapter assumes you are writing application programs for IMS environments.

This chapter explains the kinds of application programs IMS supports and the requirements each satisfies.

In this Chapter:

Related Reading: For information on writing CICS application programs, see "Chapter 4. Analyzing CICS Application Processing Requirements" on page 51.

Your Application's Requirements

One of the steps of application design is to decide how the business processes, or tasks, that the end user wants performed can be best grouped into a set of programs that efficiently performs the required processing. To analyze processing requirements, consider:

When the task must be performed

– Will it be scheduled unpredictably (for example, on terminal demand) or periodically (for example, weekly)?

How the program that performs the task is executed

– Will it be executed 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 the action the program is to perform 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 action 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

Security	Should access to the program be restricted?
Recovery	Are there special recovery considerations in the program's processing?
Availability	Does your application require high data availability?
Integrity	Do other departments use the same data?

Answers to questions like these can help you decide on the number of application programs that the processing will require, and on the types of programs that perform the processing most efficiently. Although rules dealing with how many programs can most efficiently do the required processing do not exist, here are some suggestions:

- As you look at each programming 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, daily as opposed to different times throughout the month), that task might be more efficiently performed by several programs.
- As you define each program, it is a good idea for maintenance and recovery reasons to keep it as simple as possible. The simpler a program is—the less it does—the easier it is to maintain, and to restart after a program or system failure. The same is true with data availability—the less data that is accessed, the more likely the data is to be available. The more limited the access requested, the more likely the data is to be available.

  Similarly, if the data that the application requires is physically in one place, it might 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.
- Documenting each of the user tasks is helpful during the design process, and in the future when others will work with your application. Be sure you are aware of installation standards in this area. The kind of information that is typically kept is when the action is to be executed, a functional description, and requirements for maintenance, security, and recovery.

  Example: For the current roster process described in “Listing Data Elements” on page 11, you might record the information shown in Figure 15 on page 31. How frequently the program is run is determined by the number of classes (20) needed by the Ed Center each week.

```
                  USER TASK DESCRIPTION

NAME:  Current Roster

ENVIRONMENT:  Batch            FREQUENCY:  20 per week

INVOKING EVENT OR DOCUMENT:  Time period (one week)

REQUIRED RESPONSE TIME:  24 hours

FUNCTION DESCRIPTION:  Print weekly, a current student

roster, in student number sequence, for each class

offered at the Ed Center.

MAINTENANCE:  Included in Education DB maintenance.

SECURITY:  None

RECOVERY:  After a failure, the ability to start

printing a particular class roster starting from a

particular sequential student number.
```

Figure 15. Documenting User Task Descriptions: Current Roster Example

The Database Your Program Accesses

When designing your program, consider the type of database it must access. The type of database depends on the operating environment. The program types you can run and the different types of databases you can access in a DB batch, TM batch, DB/DC, DBCTL, or DCCTL environment are shown in Table 8 on page 31.

Table 8. Program and Database Options in IMS Environments

Environment	Type of Program You Can Run	Type of Database That Can Be Accessed
DB Batch	DB Batch	Full-function DB2 GSAM MVS Files
TM Batch	TM Batch	DB2 GSAM MVS files

Table 8. Program and Database Options in IMS Environments (continued)

Environment	Type of Program You Can Run	Type of Database That Can Be Accessed
DB/DC	MPPs	Full-function Fast Path DEDBs and MSDBs DB2
	IFPs	Full-function Fast Path DEDBs and MSDBs DB2
	BMPs	Full-function Fast Path DEDBs and MSDBs DB2 GSAM MVS files
DBCTL	BMPs (Batch-oriented)	Full-function Fast Path DEDBs DB2 GSAM MVS files
DCCTL	MPP	DB2
	IFP	DB2
	BMP	DB2 GSAM

The types of databases that can be accessed are:

IMS Databases

IMS databases are of two types: full-function and Fast Path.

– **Full-Function Databases**

Full-function databases are hierarchic databases that are accessed through Data Language I (DL/I) call interface and can be processed by all four types of application programs: IFPs, MPPs, BMPs, and DB batch. DL/I calls make it possible for IMS application programs to retrieve, replace, delete, and add segments to full-function databases.

If your installation uses data sharing, online programs and batch programs can access the same full-function database concurrently.

Full-function database types include: HDAM, HIDAM, HSAM, HISAM, SHSAM, and SHISAM.

– **Fast Path Databases**

Fast Path databases are of two types: MSDBs and DEDBs.

- **Main storage databases** (MSDBs) are root-segment-only databases that reside in virtual storage during execution.
- **Data entry databases** (DEDBs) are hierarchic databases that provide a high level of availability for, and efficient access to, large volumes of detailed data.

MPPs, BMPs, and IFPs can access Fast Path databases. In the DBCTL environment, BMPs can access DEDBs but not MSDBs.

DB2 Databases

DB2 databases are relational databases that can be processed by IMS batch, MPPs, BMPs, and IFPs. An IMS application program might access only DL/I databases, both DL/I and DB2 databases, or only DB2 databases. Relational databases are represented to application programs and users as tables, and are processed using a relational data language called Structured Query Language (SQL).

Related Reading: For information on processing DB2 databases, see *IBM DATABASE 2 Application Programming and SQL Guide*.

MVS Files

BMPs (in both the DB/DC and DBCTL environment) are the only type of online application program that can access MVS files for their input or output. Batch programs can also access MVS files.

GSAM Databases

Generalized Sequential Access Method (GSAM) is an access method that makes it possible for BMPs and batch programs to access a sequential MVS data set as a simple database. A GSAM database can be accessed by MVS or by IMS.

Accessing Data: The Types of Programs You Can Write

You must decide upon what type of program to use: batch programs, message processing programs (MPPs), IMS Fast Path programs (IFPs), or batch message processing programs (BMPs). As Table 8 on page 31 shows, the types of programs you can use depend on whether you are running in the batch, DB/DC, or DBCTL environment.

The following sections explain the types of databases that the programs access, when the programs are used, and how to recover the programs.

DB Batch Processing

The following description of DB batch processing can help you decide if this batch program is appropriate for your application.

Data That a DB Batch Program Can Access

A DB batch program can access full-function databases, DB2 databases, GSAM databases, and MVS files. A DB batch program cannot access DEDBs or MSDBs.

Using DB Batch Processing

Batch programs are typically longer-running programs than online programs. You use a batch program when you have a large number of database updates to do or a report to print. Because a batch program runs by itself—it does not compete with any other programs for resources like databases—it can run independently of the control region. If your installation uses data sharing, DB batch programs and online programs can access full-function databases concurrently. Batch programs:

- Typically produce a large amount of output, such as reports.
- Are not executed by another program or user. They are usually scheduled at specific time intervals (for example, weekly) and are started with JCL.
- Produce output that is not needed right away. The turnaround time for batch output is not crucial, as it usually is for online programs.

Recovering a DB Batch Program

You include checkpoints in your batch program to restart it in case of failure.

Issuing Checkpoints: You issue checkpoints in a batch program to commit database changes and provide places from which to restart your program. Issuing checkpoints in a batch program is important, because commit points do not occur automatically, as they do in MPPs, transaction-oriented BMPs, and IFPs.

Issuing checkpoints is particularly important in a batch program that participates in data sharing with your online system. Checkpoints free up resources for use by online programs. You should initially include checkpoints in all batch programs that you write. Even though the checkpoint support might not be needed then, it is easier to incorporate checkpoints initially than to try to fit them in later. And it is possible that you might want to convert your batch program to a BMP or participate in data sharing. For more information on issuing checkpoints, see "Checkpoints in Batch Programs" on page 46.

To issue checkpoints (or other system service calls), you must specify an I/O PCB for your program. To obtain an I/O PCB, use the compatibility option by specifying CMPAT=YES in the PSBGEN statement in your program's PSB.

Related Reading: For more information on obtaining an I/O PCB, see *IMS/ESA Application Programming: Database Manager*.

Recommendation: For PSBs used by DB batch programs, always specify CMPAT=YES.

Backing out Database Changes: The type of storage medium for the system log determines what happens when a DB batch program terminates abnormally. Installations can specify that the system log be stored on either DASD or tape.

System Log on DASD: If the system log is stored on DASD, the installation can specify, with the BKO execution parameter, that IMS is to dynamically back out the changes that the program has made to the database since its last commit point.

Related Reading: For information on using the BKO execution parameter, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

Dynamically backing out database changes has the following advantages:

- Data accessed by the program that failed is available to other programs immediately. Otherwise, if batch backout is used, other programs cannot access the data until the IMS Batch Backout utility has been run to back out the database changes.
- If data sharing is being used and two programs are deadlocked, one of the programs can continue processing. Otherwise, if batch backout is used, both programs fail.

IMS performs dynamic backout for a batch program when an IMS-detected failure occurs, for example, when a deadlock is detected. Logging to DASD makes it possible for batch programs to issue the SETS, ROLB, and ROLS system service calls. These calls cause IMS to dynamically back out changes that the program has made.

Related Reading: For information on the SETS, ROLB, and ROLS calls, see the chapter entitled "Recovering Databases and Maintaining Database Integrity" in either of the following books:

- *IMS/ESA Application Programming: Database Manager*
- *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*

System Log on Tape: If a batch application program terminates abnormally and the system log is stored on tape, you must use the IMS Batch Backout utility to back out the program's changes to the database.

TM Batch Processing

A TM batch program acts like a DB batch program with the following differences:

- It cannot access full-function databases, but it can access DB2 databases, GSAM databases, and MVS files.
- To issue checkpoints for recovery, you need not specify CMPAT=YES in your program's PSB. (The CMPAT parameter is ignored in TM batch.) The I/O PCB is always the first PCB in the list.
- You cannot dynamically back out a database because IMS does not own the databases.

Processing Messages: MPPs

The following description of a message processing program (MPP) can help you decide if this online program is appropriate for your application.

Data That an MPP Can Access

An MPP is an online program that can access full-function databases, DEDBs, MSDBs, and DB2 databases. Unlike BMPs and batch programs, MPPs cannot access GSAM databases. MPPs can only run in DB/DC and DCCTL environments.

Using an MPP

The primary purpose of an MPP is to process requests from users at terminals and from other application programs. Ideally, MPPs are very small, and the processing they perform is tailored to respond to requests quickly. They process messages as their input, and send messages as responses.

Definition: A *message* is data that is transmitted between any two terminals, application programs, or IMS systems. Each message has one or more segments.

MPPs are executed through transaction codes. When you define an MPP, you associate it with one or more transaction codes. Each transaction code represents a transaction the MPP is to process. To process a transaction, a user at a terminal enters a code for that transaction. IMS then schedules the MPP associated with that code, and the MPP processes the transaction. The MPP might need to access the database to do this. Generally, an MPP goes through these five steps to process a transaction:

1. Retrieve a message from IMS.
2. Process the message and access the database as necessary.
3. Respond to the message.
4. Repeat the process until no messages are forthcoming.
5. Terminate.

When an MPP is defined, a system administrator makes decisions about the program's scheduling and processing. For each MPP, a system administrator specifies:

- The transaction's priority
- 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

Defining priorities and processing limits gives system administration some control over load balancing and processing.

Although an MPP's primary purpose is to process and reply to messages quickly, it is flexible in how it processes a transaction and where it can send output messages. For example, an MPP can send output messages to other terminals and application programs. See "Chapter 5. Gathering Requirements for Database Options" on page 69 for a description of some of the options available to MPPs.

Processing Messages: IFPs

The following description of an IMS Fast Path (IFP) program can help you decide if this online program is appropriate for your application.

Data That an IFP Can Access

An IFP is similar to an MPP: Its main purpose is to quickly process and reply to messages from terminals.

Like an MPP, an IFP can access full-function databases, DEDBs, MSDBs, and DB2 databases. IFPs can only be run in DB/DC and DCCTL environments.

Using an IFP

You should use an IFP if you need quick processing and can accept the characteristics and constraints associated with IFPs.

The main differences between IFPs and MPPs are as follows:

- Messages processed by IFPs must consist of only one segment. Messages that are processed by MPPs can consist of several segments.
- IFPs bypass IMS queuing, allowing for more efficient processing. Transactions that are processed by Fast Path's expedited message handler are on a first-in, first-out basis.

IFPs also have the following characteristics:

- They run in **transaction response mode**. This means that they must respond to the terminal that sent the message before the terminal can enter any more requests.
- They process only **wait-for-input transactions**. When you define a program as processing wait-for-input transactions, the program remains in virtual storage, even when no additional messages are available for it to process.

Restrictions:

- An IMS program cannot send messages to an IFP transaction unless it is in another IMS system that is connected using Intersystem Communication (ISC).
- MPPs cannot pass conversations to an IFP transaction.

Recovering an IFP

IFPs must be defined as single mode. This means that a commit point occurs each time the program retrieves a message. Because of this, you do not need to issue checkpoint calls.

Batch Message Processing: BMPs

BMPs are application programs that can perform batch-type processing online and access the IMS message queues for their input and output. Because of this and because of the data available to them, BMPs are the most flexible of the IMS application programs.

The two types of BMPs are: **batch-oriented** and **transaction-oriented**. They are described in the following sections.

Batch Processing Online: Batch-Oriented BMPs

The following description of a batch message processing program can help you decide if this batch program is appropriate for your application.

Data a Batch-Oriented BMP Can Access: A batch-oriented BMP performs batch-type processing in any online environment. When run in the DB/DC or DCCTL environment, a batch-oriented BMP can send its output to the IMS message queue to be processed later by another application program. Unlike a transaction-oriented BMP, a batch-oriented BMP cannot access the IMS message queue for input.

In the DBCTL environment, a batch-oriented BMP can access full-function databases, DB2 databases, DEDBs, MVS files, and GSAM databases. In the DB/DC environment, a batch-oriented BMP can access all of these types of databases, as well as Fast Path MSDBs. In the DCCTL environment, this program can access DB2 databases, MVS files, and GSAM databases.

Using a Batch-Oriented BMP: A batch-oriented BMP can be simply a batch program that runs online. (Online requests are processed by the IMS DB/DC, DBCTL, or DCCTL system rather than by a batch system.) You can even run the same program as a BMP or as a batch program.

Recommendation: If the program performs a large number of database updates without issuing checkpoints, consider running it as a batch program so that it does not degrade the performance of the online system.

To use batch-oriented BMPs most efficiently, avoid a large amount of batch-type processing online. If you have a BMP that performs time-consuming processing such as report writing and database scanning, schedule it during non-peak hours of processing. This will prevent it from degrading the response time of MPPs.

Because BMPs can degrade response times, the response time requirements at your installation should be the main consideration in deciding the extent to which you will use batch message processing. Therefore, use BMPs accordingly.

Recovering a Batch-Oriented BMP: Issuing checkpoint calls is an important part of a batch-oriented BMP's processing, because commit points do not occur automatically, as they do in MPPs, transaction-oriented BMPs, and IFPs. Unlike most batch programs, a BMP shares resources with MPPs. In addition to committing database changes and providing places from which to restart (as for a batch program), checkpoints release resources that are locked for the program. For more information on issuing checkpoints, see "Checkpoints in Batch-Oriented BMPs" on page 45.

If a batch-oriented BMP fails, IMS and DB2 back out the database updates the program has made since the last commit point. You then restart the program with JCL. If the BMP processes MVS files, you must provide your own method of taking checkpoints and restarting.

Converting a Batch Program to a Batch-Oriented BMP: If you have IMS TM or are running in the DBCTL environment, you can convert a batch program to a batch-oriented BMP.

- If you have IMS TM, you might want to convert your programs for these reasons:
  - BMPs can send output to the message queues.
  - BMPs can access DEDBs and MSDBs.
  - BMPs simplify program recovery because logging goes to a single system log. If your installation uses DASD for the system log in batch, you can specify that you want dynamic backout for the program. In that case, batch recovery is similar to BMP recovery, except, of course, with batch you need to manage multiple logs.
  - Restart can be done automatically from the last checkpoint without changing the JCL.
- If you are using DBCTL, you might want to convert your programs for these reasons:
  - BMPs can access DEDBs.
  - BMPs simplify program recovery because logging goes to a single system log. If your installation uses DASD for the system log in batch, you can specify that you want dynamic backout for the program. In that case, batch recovery is similar to BMP recovery, except, of course, with batch you need to manage multiple logs.
- If you are running Sysplex data sharing and you either have IMS TM or are using DBCTL, you might want to convert your program. This is because using batch-oriented BMPs helps you stay within the Sysplex data-sharing limit of 32 connections for each OSAM or VSAM structure.

  If your installation uses data sharing, you can run batch programs concurrently with online programs. If your installation does not use data sharing, converting a batch program to a BMP makes it possible to run the program with BMPs and other online programs.

  Also, if you plan to run your batch programs offline, converting them to BMPs enables you to run them with the online system, instead of waiting until the online system is not running. Running a batch program as a BMP can also keep the data more current.
- If you have IMS TM or are using DBCTL, you can have a program that runs as either a batch program or a BMP.

  Recommendation: Code your checkpoints in a way that makes them easy to modify. Converting a batch program to a BMP or converting a batch program to use data sharing requires more frequent checkpoints. Also, if a program fails while running in a batch region, you must restart it in a batch region. If a program fails in a BMP region, you must restart it in a BMP region.

The requirements for converting a batch program to a BMP are:

- The program must have an I/O PCB. You can obtain an I/O PCB in batch by specifying the compatibility (CMPAT) option in the program specification block (PSB) for the program.

  Related Reading: For more information on the CMPAT option in the PSB, see *IMS/ESA Utilities Reference: System*.

- BMPs must issue checkpoint calls more frequently than batch programs.

Batch Message Processing: Transaction-Oriented BMPs

The following description of a transaction-oriented BMP can help you decide if this batch program is appropriate for your application.

Data a Transaction-Oriented BMP Can Access: Transaction-oriented BMPs can access MVS files, GSAM databases, DB2 databases, full-function databases, DEDBs, and MSDBs.

Unlike a batch-oriented BMP, a transaction-oriented BMP can access the IMS message queue for input and output, and it can only run in the DB/DC and DCCTL environments.

Using a Transaction-Oriented BMP: Unlike MPPs, transaction-oriented BMPs are not scheduled by IMS. You schedule them as needed and start them with JCL. For example, an MPP, as it processes each message, might send an output message giving details of the transaction to the message queue. A transaction-oriented BMP could then access the message queue to produce a daily activity report.

Typically, you use a transaction-oriented BMP to simulate direct update online: Instead of updating the database while processing its transactions, an MPP sends its updates to the message queue. A transaction-oriented BMP then performs the updates for the MPP. You can run the BMP as needed, depending on the number of updates. This improves response time for the MPP, and it keeps the data current. This can be more efficient than having the MPP process its transactions if the MPP's response time is very important. One disadvantage in doing this, however, is that it splits the transaction into two parts which is not necessary.

If you have a BMP perform an MPP's updates, design the BMP so that, if the BMP terminates abnormally, you can reenter the last message as input for the BMP when you restart it. For example, suppose an MPP gathers database updates for three BMPs to process, and one of the BMPs terminates abnormally. You would need to reenter the message that the terminating BMP was processing to one of the other BMPs for reprocessing.

BMPs can process transactions defined as wait-for-input (WFI). This means that IMS allows the BMP to remain in virtual storage after it has processed the available input messages. IMS returns a QC status code, indicating that the program should terminate when one of the following occurs:

- The program reaches its time limit.
- The master terminal operator enters a command to stop processing.
- IMS is terminated with a checkpoint shutdown.

You specify WFI for a transaction on the WFI parameter of the TRANSACT macro during IMS system definition.

A batch message processing region (BMP) scheduled against wait-for-input (WFI) transactions returns a QC status code (no more messages) only for the following commands: `/PSTOP REGION`, `/DBD`, `/DBR`, or `/STA`.

Like MPPs, BMPs can send output messages to several destinations, including other application programs. See "Identifying Output Message Destinations" on page 99 for more information.

Recovering a Transaction-Oriented BMP: Like MPPs, with transaction-oriented BMPs, you can choose where commit points occur in the program. You can specify that a transaction-oriented BMP be single or multiple mode, just as you can with an MPP. If the BMP is single mode, issuing checkpoint calls is not as critical as in a multiple mode BMP. In a single mode BMP, a commit point occurs each time the program retrieves a message. For more information on issuing checkpoints in a BMP, see “Checkpoints in MPPs and Transaction-Oriented BMPs” on page 44.

Programming Integrity and Recovery Considerations

This section explains how IMS protects data integrity, and how you can plan ahead for program recovery. These topics assume some knowledge of IMS application programming. You might want to read this section after reading the IMS application programming book that is applicable for your environment (as explained in the “Preface” on page xiii).

How IMS Protects Data Integrity: Commit Points

When an online program accesses the database, it is not necessarily the only program doing so. IMS and DB2 make it possible for more than one application program to access the data concurrently without endangering the integrity of the data. To do this, IMS and DB2 prevent other application programs from accessing segments that your program deletes, replaces, or inserts, until your program reaches a *commit point*, the place in the program’s processing at which it completes a unit of work. IMS and DB2 then make the changes your program has made to the database permanent and make the changed data available to other application programs.

What Happens at a Commit Point

When an application program finishes processing one distinct unit of work, IMS and DB2 consider that processing to be valid, even if the program later encounters problems. For example, an application program that is retrieving, processing, and responding to a message from a terminal constitutes a *unit of work*. If the program encounters problems while processing the next input message, the processing it has done on the first input message is not affected. These input messages are separate pieces of processing.

A commit point indicates to IMS that a program has finished a unit of work, and that the processing it has done is accurate. At that time:

- IMS releases segments it has locked for the program since the last commit point. Those segments are then available to other application programs.
- IMS and DB2 make the program’s changes to the database permanent.
- The current position in all databases except GSAM is reset to the start of the database.

If the program terminates abnormally before reaching the commit point:

- IMS and DB2 back out all of the changes the program has made to the database since the last commit point. (This does not apply to batch programs that write their log to tape.)
- IMS discards any output messages that the program has produced since the last commit point.

  Until the program reaches a commit point, IMS holds the program’s output messages so that, if the program terminates abnormally, users at terminals and other application programs do not receive inaccurate information from the abnormally terminating application program.

If the program is processing an input message and terminates abnormally, the input message is **not** discarded if both of the following conditions exist:

1. Your installation is not using the Non-Discardable Messages (NDM) exit routine.
2. IMS terminates the program with one of the following abend codes: U0777, U2478, U2479, U3303. The input message is saved and processed later.

   Exception: The input message is discarded if it is not terminated by one of the abend codes previously referenced. When the program is restarted, IMS gives the program the next message.

If the program is processing an input message when it terminates abnormally, and if your installation is using the NDM exit routine, the input message might be discarded from the system regardless of the abend. Whether the input message is discarded from the system depends on how for your installation has written the NDM exit routine.

Related Reading: See *IMS/ESA Customization Guide* for more information about the NDM exit routine.

- IMS notifies the MTO that the program terminated abnormally.
- IMS and DB2 release any locks that the program has held on data it has updated since the last commit point. This makes the data available to other application programs and users.

Where Commit Points Occur

A commit point can occur in a program for any of the following reasons:

- The program terminates normally. Except for a program that accesses Fast Path resources, normal program termination is always a commit point. A program that accesses Fast Path resources must reach a commit point before terminating.
- The program issues a Checkpoint call. Checkpoint calls are a program’s means of explicitly indicating to IMS that it has reached a commit point in its processing.
- If a program processes messages as its input, a commit point might occur when the program retrieves a new message. IMS considers this commit point the start of a new unit of work in the program. Retrieving a new message is not always a commit point. This depends on whether the program has been defined as **single mode** or **multiple mode**.
  - If you specify single mode, a commit point occurs each time the program issues a call to retrieve a new message. Specifying single mode can simplify recovery, because you can restart the program from the most recent call for a new message if the program terminates abnormally. When IMS restarts the program, the program begins by processing the next message.
  - If you specify multiple mode, a commit point occurs when the program issues a Checkpoint call or when it terminates normally. At those times, IMS sends the program’s output messages to their destinations. Because multiple-mode programs contain fewer commit points than do single-mode programs, multiple-mode programs might offer slightly better performance than single-mode programs. When a multiple-mode program terminates abnormally, IMS can only restart it from a checkpoint. Instead of reprocessing only the most recent message, a program might have several messages to reprocess, depending on when the program issued the last Checkpoint call.

Integrity and Recovery Considerations

Figure 16 shows the modes in which the programs can run. Because processing mode is not applicable to batch programs and batch-oriented BMPs, they are not listed in the figure.

Mode / Program Type	Single Mode Only	Multiple Mode Only	Either Mode
MPP			X
IFP	X		
Transaction-Oriented BMP			X

Figure 16. Processing Modes

You specify single or multiple mode on the MODE parameter of the TRANSACT macro.

Related Reading: For information on the TRANSACT macro, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

See Figure 17 on page 42, for an illustration of the difference between single-mode and multiple-mode programs.

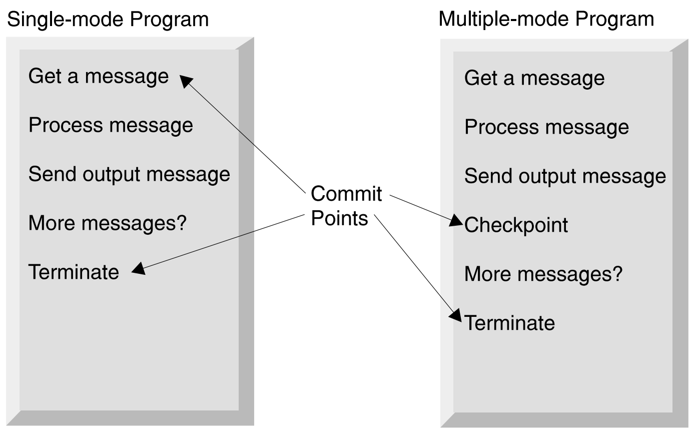


Figure 17. Single Mode and Multiple Mode

DB2 does some processing with multiple- and single-mode programs that IMS does not. When a multiple-mode program issues a call to retrieve a new message, DB2 performs an authorization check. If the authorization check is successful, DB2 closes any SQL cursors that are open. This affects the design of your program.

Related Reading: For more information on this topic, see *IMS/ESA Application Programming: Transaction Manager*.

DB2's SQL `COMMIT` statement causes DB2 to make permanent changes to the database. However, this statement is valid only in TSO application programs. If an IMS application program issues this statement, it receives a negative SQL return code.

Planning for Program Recovery: Checkpoint and Restart

Recovery in an IMS application program that accesses DB2 data is handled by both IMS and DB2. IMS coordinates the process, and DB2 handles recovery of DB2 data.

Introducing Checkpoint Calls

Checkpoint calls indicate to IMS that the program has reached a commit point. They also establish places in the program from which the program can be restarted. IMS has symbolic Checkpoint calls and basic Checkpoint calls.

A program might issue only one type of Checkpoint call.

- MPPs and IFPs must use basic Checkpoint calls.
- BMPs and batch programs can use either symbolic Checkpoint calls or basic Checkpoint calls.

Programs that issue symbolic Checkpoint calls can specify as many as seven data areas in the program to be checkpointed. When IMS restarts the program, the Restart call restores these areas to the condition they were in when the program issued the symbolic Checkpoint call. Because symbolic Checkpoint calls do not support MVS files, if your program accesses MVS files, you must supply your own method of establishing checkpoints.

You can use symbolic Checkpoint for either Normal Start or Extended Restart (XRST).

Example: Typical calls for a Normal start would be as follows:

- XRST (I/O area is blank)
- CHKP (I/O area has checkpoint ID)
- Database Calls (including checkpoints)
- CHKP (final checkpoint)

Example: Typical calls for an Extended Restart (XRST) would be as follows:

- XRST (I/O area has checkpoint ID)
- CHKP (I/O area has new checkpoint ID)
- Database Calls (including checkpoints)
- CHKP (final checkpoint)

Related Reading: For more information on Checkpoint calls, see *IMS/ESA Application Programming: Database Manager*.

The Restart call, which you must use with symbolic Checkpoint calls, provides a way of restarting a program after an abnormal termination. It restores the program's data areas to the way they were when the program issued the symbolic Checkpoint call. It also restarts the program from the last checkpoint the program established before terminating abnormally.

All programs can use basic Checkpoint calls. Because you cannot use the restart call with the basic Checkpoint call, you must provide program restart. Basic Checkpoint calls do not support either MVS or GSAM files. IMS programs cannot use MVS Checkpoint and Restart. If you access MVS files, you must supply your own method of establishing checkpoints and restarting.

Integrity and Recovery Considerations

In addition to the actions that occur at a commit point, issuing a Checkpoint call causes IMS to:

- Inform DB2 that the changes your program has made to the database can be made permanent. DB2 makes the changes to DB2 data permanent, and IMS makes the changes to IMS data permanent.
- Write a log record containing the checkpoint identification given in the call to the system log, but only if the PSB contains a DB PCB. You can print checkpoint log records by using the IMS File Select and Formatting Print Program (DFSERA10). With this utility, you can select and print log records based on their type, the data they contain, or their sequential positions in the data set. Checkpoint records are X'18' log records.

  Related Reading: *IMS/ESA Utilities Reference: System* describes the DFSERA10 program.
- Send a message containing the checkpoint identification that was given in the call to the system console operator and to the IMS master terminal operator.
- Return the next input message to the program's I/O area, if the program processes input messages. In MPPs and transaction-oriented BMPs, a Checkpoint call acts like a call for a new message.

Restriction: Do not specify CHKPT=EOV on any DD statement in order to take an IMS checkpoint because of unpredictable results.

When to Use Checkpoint Calls

Issuing Checkpoint calls is most important in programs that do not have built-in commit points. The decision about whether your program should issue checkpoints, and if so, how often, depends on your program. Generally, these programs should issue Checkpoint calls:

- Multiple-mode programs
- Batch-oriented BMPs (which can issue either SYNC or CHKP calls)
- Most batch programs
- Programs that run in a data sharing environment

You do not need to issue Checkpoint calls in:

- Single-mode BMP or MPP programs
- Database load programs
- Programs that access the database in read-only mode, as defined with the PROCOPT=GO option (during a PSBGEN), and are short enough to restart from the beginning
- Programs that have exclusive use of the database

Checkpoints in MPPs and Transaction-Oriented BMPs: The mode type of the program is specified on the MODE keyword of the TRANSACT macro during IMS system generation. The modes are single and multiple.

- **In single-mode programs**

  In single mode programs (MODE=SNGL was specified on the TRANSACT macro during IMS system definition), a Get Unique to the message queue causes an implicit commit to be performed.
- **In multiple-mode programs**

  In multiple-mode BMPs and MPPs, the only commit points are those that result from the Checkpoint calls that the program issues and from normal program termination. If the program terminates abnormally and it has not issued Checkpoint calls, IMS backs out the program's database updates and cancels the messages it created since the beginning of the program. If the program has issued Checkpoint calls, IMS backs out the program’s changes and cancels the output messages it has created since the most recent checkpoint.

Consider the following when issuing Checkpoint calls in multiple-mode programs:

– How long it would take to back out and recover that unit of processing. The program should issue checkpoints frequently enough to make the program easy to back out and recover.

– How you want the output messages grouped. Checkpoint calls establish how a multiple-mode program’s output messages are grouped. Programs should issue Checkpoint calls frequently enough to avoid building up too many output messages.

Depending on the database organization, issuing a Checkpoint call might reset your position in the database.

Related Reading: For more information about losing your position when a checkpoint is issued, see *IMS/ESA Application Programming: Database Manager*.

Checkpoints in Batch-Oriented BMPs: Issuing Checkpoint calls in a batch-oriented BMP is important for several reasons:

- In addition to committing changes to the database and establishing places from which the program can be restarted, Checkpoint calls release resources that IMS has locked for the program.
- A batch-oriented BMP that uses DEDBs or MSDBs might terminate with abend U1008 if a `SYNC` or `CHKP` call is not issued before the application program terminates.
- If a batch-oriented BMP does not issue checkpoints frequently enough, it can be abnormally terminated, or it can cause another application program to be abnormally terminated by IMS for any of these reasons:
  - – If a BMP retrieves and updates many database records between Checkpoint calls, it can tie up large portions of the databases and cause long waits for other programs needing those segments.

    Exception: For a BMP with a processing option of GO or exclusive, IMS does not lock segments for programs. Issuing Checkpoint calls releases the segments that the BMP has locked and makes them available to other programs.
  - – The space needed to maintain lock information about the segments that the program has read and updated exceeds what has been defined for the IMS system. If a BMP locks too many segments, the amount of storage needed for the locked segments can exceed the amount of available storage. If this happens, IMS terminates the program abnormally. You must increase the program’s checkpoint frequency before rerunning the program. The available storage is specified during IMS system definition.

    Related Reading: See *IMS/ESA Installation Volume 2: System Definition and Tailoring* for more information on specifying storage.

    You can limit the number of locks for the BMP by using the LOCKMAX=*n* parameter on the PSBGEN statement. For example, a specification of LOCKMAX=5 means the application cannot obtain more than 5000 locks at any time. The value of *n* must be between 0 and 255. When a maximum lock limit does not exist, 0 is the default. If the BMP tries to acquire more than the specified number of locks, IMS terminates the application with abend U3301.

    Related Reading: For more information about this abend, see *IMS/ESA Messages and Codes*.

Integrity and Recovery Considerations

Checkpoints in Batch Programs: Batch programs that update databases should issue Checkpoint calls. The main consideration in deciding how often to take checkpoints in 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 10 or 15 minutes.

If you might need to back out the entire batch program, the program should issue the Checkpoint call at the very beginning of the program. IMS backs out the program to the checkpoint you specify, or to the most recent checkpoint, if you do not specify a checkpoint.

For a batch program to issue Checkpoint calls, it must specify the compatibility option in its PSB (CMPAT=YES). This generates an I/O PCB for the program, which IMS uses as an I/O PCB in the Checkpoint call.

Another important reason for issuing Checkpoint calls in batch programs is that, although they may currently run in an IMS batch region, they might later need to access online databases. This would require converting them to BMPs. Issuing Checkpoint calls in a BMP is important for reasons other than recovery—for example, to release database resources for other programs. So, you should initially include checkpoints in all batch programs that you write. Although the checkpoint support might not be needed then, it is easier to incorporate Checkpoint calls initially than to try to fit them in later.

To free database resources for other programs, batch programs that run in a data-sharing environment should issue Checkpoint calls more frequently than those that do not run in a data-sharing environment.

Specifying Checkpoint Frequency

You should specify checkpoint frequency in your program so that you can easily modify it when the frequency needs to be adjusted. You can do this by:

- Using a counter in your program to keep track of elapsed time, and issuing a Checkpoint call after a certain time interval.
- Using a counter to keep track of the number of root segments your program accesses, and issuing a Checkpoint call after a certain number of root segments.
- Using a counter to keep track of the number of updates your program performs, and issuing a Checkpoint call after a certain number of updates.

Data Availability Considerations

Your program might be unable to access data in a full-function database. This section describes the conditions for an unavailable database and the program calls that allow your program to manage data under these conditions.

Dealing with Unavailable Data

The conditions that make the database totally unavailable for both read and update are:

- The `/LOCK` command for a database was issued.
- The `/STOP` command for a database was issued.
- The `/DBRECOVERY` command was issued.
- Authorization for a database failed.

The conditions that make the database available only for read and not for update are:

- The /DBDUMP command has been issued.
- Database ACCESS value is RD (read).

In addition to unavailability of an entire database, other situations involving unavailability of a limited amount of data can also inhibit program access. One such example would be a failure situation involving data sharing. The active IMS system knows which locks were held by a sharing IMS system at the time the sharing IMS system failed. Although the active IMS system continues to use the database, it must reject access to the data which the failed IMS system locked upon failure. This situation occurs for both full-function and DEDB databases.

The two situations where the program might encounter unavailable data are:

- The program makes a call requiring access to a database that was unavailable at the time the program was scheduled.
- The database was available when the program was scheduled, but limited amounts of data are unavailable. The current call has attempted to access the unavailable data.

Regardless of the condition causing the data to be unavailable, the program has two possible approaches when dealing with unavailable data. The program can be **insensitive** or **sensitive** to data unavailability.

- When the program is insensitive, IMS takes appropriate action when the program attempts to access unavailable data.
- When the program is sensitive, IMS informs the program that the data it is attempting to access is not available.

If the program is insensitive to data unavailability, and attempts to access unavailable data, IMS aborts the program (3303 pseudoabend), and backs out any updates the program has made. The input message that the program was processing is suspended, and the program is scheduled to process the input message when the data becomes available. However, if the database is unavailable because dynamic allocation failed, a call results in an AI (unable to open) status code.

If the program is sensitive to data unavailability and attempts to access unavailable data, IMS returns a status code indicating that it could not process the call. The program then takes the appropriate action. A facility exists for the program to initiate the same action that IMS would have taken if the program had been insensitive to unavailable data.

IMS does not schedule batch programs if the data that the program can access is unavailable. If the batch program is using block-level data sharing, it might encounter unavailable data if the sharing system fails and the batch system attempts to access data that was updated but not committed by the failed system.

Scheduling and Accessing Unavailable Databases

By using the INIT, INQY, SETS, SETU, and ROLS calls, the program can manage a data environment where the program is scheduled with unavailable databases.

The INIT call informs IMS that the program is sensitive to unavailable data and can accept the status codes that are issued when the program attempts to access such data. The INIT call can also be used to determine the data availability for each PCB.

Integrity and Recovery Considerations

The INQY call is operable in both batch and online IMS environments. IMS application programs can use the INQY call to request information regarding output destination, session status, the current execution environment, the availability of databases, and the PCB address based on the PCBNAME. The INQY call is only supported via the AIB interface (AIBTDLI or CEETDLI using the AIB rather than the PCB address).

The SETS, SETU, and ROLS calls enable the application to define multiple points at which to preserve the state of full-function (except HSAM) databases and message activity. The application can then return to these points at a later time. By issuing a SETS or SETU call before initiating a set of DL/I calls to perform a function, the program can later issue the ROLS call if it cannot complete a function due to data unavailability.

The ROLS call allows the program to roll back its IMS full-function database activity to the state that it was in prior to a SETS or SETU call being issued. If the PSB contains an MSDB or a DEDB, the SETS and ROLS (with token) calls are invalid. Use the SETU call instead of the SETS call if the PSB contains a DEDB, MSDB, or GSAM PCB.

Related Reading: For more information on using the SETS and SETU calls with the ROLS call, see *IMS/ESA Application Programming: Database Manager*.

The ROLS call can also be used to undo all update activity (database and messages) since the last commit point and to place the current input message on the suspend queue for later processing. This action is initiated by issuing the ROLS call without a token or I/O area.

Restriction: With DB2, you cannot use ROLS (with a token) or SETS.

Use of STAE or ESTAE and SPIE in IMS Programs

IMS uses STAE or ESTAE routines in the control region, the dependent (MPP, IFP, BMP) regions, and the batch regions. In the control region, STAE or ESTAE routines ensure that database logging and various resource cleanup functions are complete. In the dependent region, STAE or ESTAE routines are used to notify the control region of any abnormal termination of the application program or the dependent region itself. If the control region is not notified of the dependent region termination, resources are not properly released and normal checkpoint shutdown might be prevented.

In the batch region, STAE or ESTAE routines ensure that database logging and various resource cleanup functions are complete. If the batch region is not notified of the application program termination, resources might not be properly released.

Two important aspects of the STAE or ESTAE facility are that:

- IMS relies on its STAE or ESTAE facility to ensure database integrity and resource control.
- The STAE or ESTAE facility is also available to the application program.

Because of these two factors, be sure you clearly understand the relationship between the program and the STAE or ESTAE facility.

Generally, do not use the STAE or ESTAE facility in your application program. However, if you believe that the STAE or ESTAE facility is required, you must observe the following basic rules:

- When the environment supports STAE or ESTAE processing, the application program STAE or ESTAE routines always get control before the IMS STAE or ESTAE routines. Therefore, you must ensure that the IMS STAE or ESTAE exit routines receive control by observing the following procedures in your application program:
  - Establish the STAE or ESTAE routine only once and always before the first DL/I call.
  - When using the STAE or ESTAE facility, the application program should not alter the IMS abend code.
  - Do not use the RETRY option when exiting from the STAE or ESTAE routine. Instead, return a CONTINUE-WITH-TERMINATION indicator at the end of the STAE or ESTAE processing. If your application program specifies the RETRY option, be aware that IMS STAE or ESTAE exit routines will not get control to perform cleanup. Therefore, system and database integrity might be compromised.
  - For PL/I use of STAE and SPIE, see the description of IMS considerations in *OS PL/I Version 2 Programming Guide*.
  - For PL/I, COBOL, and C/MVS, if you are using the AIBTDLI interface in a non-Language Environment enabled system, you must specify NOSTAE and NOSPIE. However, in Language Environment Version 1.2 or later enabled environment, the NOSTAE and NOSPIE restriction is removed.
- The application program STAE or ESTAE exit routine must not issue DL/I calls (DB or TM) because the original abend might have been caused by a problem between the application and IMS. A problem between the application and IMS could result in recursive entry to STAE or ESTAE with potential loss of database integrity, or in problems taking a checkpoint. This also could result in a hang condition or an ABENDU0069 during termination.

Dynamic Allocation for IMS Databases

Use the dynamic allocation function to specify the JCL information for IMS databases in a library instead of in the JCL of each batch or online job.

Related Reading: For additional information on the definitions for dynamic allocation, see the description of the DFSMDA macro in *IMS/ESA Utilities Reference: System*.

If your installation uses dynamic allocation, do not include JCL DD statements for any database data sets that have been defined for dynamic allocation. Check with the DBA or comparable specialist at your installation to determine which databases have been defined for dynamic allocation.

Chapter 4. Analyzing CICS Application Processing Requirements

This chapter provides information for writing application programs in a CICS environment. See "Chapter 3. Analyzing IMS Application Processing Requirements" on page 29 for the corresponding information on IMS application programming.

The following sections explain the kinds of programs CICS supports and the requirements that each satisfies.

In this Chapter:

Your Application's Requirements

One of the steps of application design is to decide how the business processes, or tasks 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 performed

– Will it be scheduled unpredictably (for example on terminal demand) or periodically (for example, weekly)?

How the program that performs the task is executed

– Will it be executed online, where response time is more important, or by batch job submission, where a slower response time is acceptable?

The consistency of the processing components

– Does this action the program is to perform 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 action 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

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?

Answers to questions like these can help you decide on the number of application programs that the processing will require, and on the types of programs that perform the processing most efficiently. Although rules dealing with how many programs can most efficiently do the required processing do not exist, here are some suggestions:

- As you look at each programming 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 is 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 to restart after a program or system failure. The same is true with data availability—the less data that is accessed, the more likely the data is to be available; the more limited the data accessed, the more likely the data is to be available.

  Similarly, if the data that the application requires is physically in one place, it might 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.
- Documenting each of the user tasks is helpful during the design process, and in the future when others will work with your application. Be sure you are aware of installation standards in this area. The kind of information that is typically kept is when the task is to be executed, a functional description, and requirements for maintenance, security, and recovery.

  Example: For the Current Roster process described under “Listing Data Elements” on page 11, you might record the information shown in Figure 18 on page 53. How frequently the program is run is determined by the number of classes (20) for which the Ed Center will print current rosters each week.

```
                 USER TASK DESCRIPTION

NAME:  Current Roster

ENVIRONMENT:  Batch             FREQUENCY:  20 per week

INVOKING EVENT OR DOCUMENT:  Time period (one week)

REQUIRED RESPONSE TIME:  24 hours

FUNCTION DESCRIPTION:  Print weekly, a current student

roster, in student number sequence, for each class

offered at the Ed Center.

MAINTENANCE:  Included in Education DB maintenance.

SECURITY:  None

RECOVERY:  After a failure, the ability to start

printing a particular class roster starting from a

particular sequential student number.
```

Figure 18. Current Roster Task Description

The Database Your Program Accesses

When designing your program, consider the type of data it must access. The type of data depends on the operating environment. The data available to CICS online and IMS batch programs is shown in Table 9.

Table 9. The Data Your Program Can Access

Type of Program	DL/I Databases	DB2 Databases	MVS Files
CICS Online	Yes	Yes	Yes
DB Batch	Yes	Yes	Yes

Notes:
[1] Except for Generalized Sequential Access Method (GSAM) databases. GSAM enables batch programs to access a sequential MVS data set as a simple database.
[2] IMS does not participate in the call process.
[3] Access through CICS file control or transient data services.

Also, consider the type of database your program must access. As shown in Table 10 on page 54, the type of database that can be accessed depends on the operating environment.

The Database Your Program Accesses

Table 10. Program and Database Options in the CICS Environments

Environment	Type of Program You Can Write	Type of Database That Can Be Accessed
DB Batch	DB Batch	DL/I Full-function DB2 GSAM MVS Files
DBCTL	BMP	DL/I Full-function DL/I DEDBs DB2 GSAM MVS Files
	CICS Online	DL/I Full-function DL/I DEDBs DB2 MVS Files (access through CICS file control or transient data services)

Notes:
¹A third CICS environment, referred to as remote DL/I, also exists. In this environment, a CICS system supports applications that issue DL/I calls but does not service the requests itself. It “function ship” the DL/I calls to another CICS system that is using DBCTL. For more information on remote DL/I, see *CICS/ESA Database Control Guide*.
²IMS does not participate in the call process.

The types of databases that can be accessed are:

Full-Function Databases

Full-function databases are hierarchic databases that are accessed through Data Language I (DL/I). DL/I calls enable application programs to retrieve, replace, delete, and add segments to full-function databases. CICS online and BMP programs can access the same database concurrently (if participating in IMS data sharing); an IMS batch program must have exclusive access to the database (if not participating in IMS data sharing). See “Using Data Sharing” on page 59 for more details about when to use this environment.

All types of programs (batch, BMPs, and online) can access full-function databases.

Fast Path DEDBs

Data entry databases (DEDBs) are hierarchic databases for, and efficient access to, large volumes of detailed data. In the DBCTL environment, CICS online and BMP programs can access DEDBs.

DB2 Databases

DB2 databases are relational databases. Relational databases are represented to application programs and users as tables and are processed using a relational data language called Structured Query Language (SQL). DB2 databases can be processed by CICS online transactions, and by IMS batch and BMP programs.

Related Reading: For information on processing DB2 databases, see *IBM DATABASE 2 Version 3 Application Programming and SQL Guide*.

GSAM Databases

Generalized Sequential Access Method (GSAM) is an access method that enables BMPs and batch programs to access a ″flat″ sequential MVS data set as a simple database. A GSAM database can be accessed by MVS or CICS.

MVS Files

CICS online and IMS batch programs can access MVS files for their input, processing, or output. Batch programs can access MVS files directly; online programs must access them through CICS file control or transient data services.

Writing a Program to Access IMS Databases: The Types of Programs

This section explains the following kinds of application programs that CICS users can write to process IMS databases:

- CICS online programs
- IMS batch programs
- IMS batch message processing (BMP) programs that are batch-oriented

As shown in Table 10 on page 54, the types of programs you can use depend on whether you are running in the DBCTL environment. Within the different environments, the type of program you write depends on the processing your application requires. Each type of program answers different application requirements.

Writing a CICS Online Program

The following description of a CICS online program can help you decide if an online program is appropriate for your application.

Data That a CICS Online Program Can Access

CICS online programs run in the DBCTL environment and can access IMS full-function databases, Fast Path DEDBs, DB2 databases, and MVS files.

Online programs that access IMS databases are executed in the same way as other CICS programs.

Using a CICS Online Program

An online program runs under the control of CICS, and it accesses resources concurrently with other online programs. Some of the application requirements online programs can answer are:

- Information in the database must be available to many users.
- Program needs to communicate with terminals and other programs.
- Programs must be available to users at remote terminals.
- Response time is important.

The structure of an online program, and the way it receives status information, depend on whether it is a call- or command-level program. However, both command- and call-level online programs:

- Schedule a PSB (for CICS online programs). A PSB is automatically scheduled for batch or BMP programs.
- Issue either commands or calls to access the database. Online programs cannot mix commands and calls in one logical unit of work (LUW).
- Optionally, terminate a PSB for CICS online programs.
- Issue an EXEC CICS RETURN statement when they have finished their processing. This statement returns control to the linking program. When the highest-level program issues the RETURN statement, CICS regains control and terminates the PSB if it has not yet been terminated.

Programs to Access IMS Databases

Because an online application program can be used concurrently by several tasks, it must be quasi-reentrant.

An online program in the DBCTL environment can use many IMS system service requests.

Related Reading:

- For more information on writing these types of programs, see *IMS/ESA Application Programming: Database Manager*or *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*.
- For more details about programming techniques and restrictions, see *CICS/ESA Application Programming Reference*.
- For a summary of the calls and commands an online program can issue, see *IMS/ESA Application Programming: Database Manager* or *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*.

DL/I database or system service requests must refer to one of the program communication blocks (PCBs) from the list of PCBs passed to your program by IMS. The PCB that must be used for making system service requests is called the I/O PCB. When present, it is the first PCB in the list of PCBs.

For an online program in the DBCTL environment, the I/O PCB is optional. To use the I/O PCB, you must indicate this in the application program when it schedules the PSB.

Before you run your program, the program specification blocks (PSBs) and database descriptions (DBDs) the program uses must be converted to internal control block format using the IMS ACBGEN utility. (PSBs tell IMS an application program's characteristics and use of data and terminals. DBDs tell IMS a database's physical and logical characteristics.)

Related Reading: For more information on performing an ACBGEN and a PSBGEN, see *IMS/ESA Utilities Reference: System*.

Because an online program shares a database with other online programs, it may affect the performance of your online system. For more information on what you can do to minimize the effect your program has on performance, see "Maximizing the Performance of Your CICS System" on page 60.

Writing an IMS Batch Program

The following description of a batch program can help you decide if this program is appropriate for your application.

Data That a Batch Program Can Access

A batch program can access DL/I full-function, DB2, and GSAM databases, and MVS files. A batch program cannot access DEDBs or MSDBs, and it can run in the DBCTL environment.

Using a Batch Program

Batch programs typically run longer than online programs. If it is not participating in IMS data sharing, a batch program runs by itself and does not compete with other programs for database resources. Use a batch program to do a large number of database updates or when you want to print a report. Batch programs:

- Typically produce a large amount of output—for example, reports.

- Are not executed by another program or user. They are usually scheduled at specific time intervals (for example, weekly) and are started with JCL.
- Produce output that is not needed right away. The response time for batch output is not as important as it usually is for online programs.

The structure of a batch program and the way it receives status information depend on whether it is a command- or call-level program.

Related Reading: For more information on this topic, see *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS* or *IMS/ESA Application Programming: Database Manager*.

Unlike online programs, batch programs do not schedule or terminate PSBs. This is done automatically.

Batch programs can issue system service requests (such as checkpoint, restart, and rollback) to perform functions such as dynamically backing out database changes made by your program.

Related Reading: For a summary of the commands and calls, you can use in a batch program, see *IMS/ESA Application Programming: Database Manager* or *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*.

When performing a PSBGEN, your installation must define the language of the program that will schedule the PSB. For your program to be able to successfully issue certain system service requests, such as a checkpoint or a rollback request, an I/O PCB must be available for your program. To obtain an I/O PCB, specify CMPAT=YES in the PSBGEN statement. Make all batch programs sensitive to the I/O PCB so that checkpoints are easily introduced. Design all batch programs with checkpoint and restart in mind. Although the checkpoint support may not be needed initially, it is easier to incorporate checkpoints initially than to try to fit them in later. With checkpoints, it will be easier to convert batch programs to BMP programs or to batch programs that use data sharing.

Related Reading: For more information about obtaining an I/O PCB, see “Requesting an I/O PCB in Batch Programs” on page 62. For information on how to perform a PSBGEN, see *IMS/ESA Utilities Reference: System*.

Converting a Batch Program to a Batch-Oriented BMP

If you are running in the DBCTL environment, you can convert a batch program to a batch-oriented BMP. Conversion to a BMP can be advantageous for these reasons:

- Logging is to the IMS log, which means that multiple logs are unnecessary.
- Automatic backout is available.
- Restart can be done automatically from the last checkpoint without changing the JCL.
- Concurrent access to databases is possible. If you are needing to run your batch programs offline, converting them to BMPs enables you to run them with the online system, instead of having to wait until the online system is not running. Running a batch program as a BMP can also keep the data more current.
- BMPs can access DEDBs.
- You can have a program that runs as either a batch or BMP program. However, because batch programs require fewer checkpoint calls than BMPs (except when data sharing), code Checkpoint calls in a way that makes them easy to modify.

Also, if a program fails while running in a batch region, you must restart it in a batch region. If a program fails in a BMP region, you must restart it in a BMP region.

- If you are running Sysplex data sharing, use of batch-oriented BMPs helps you stay within the Sysplex data sharing limit of 32 connections for each OSAM or VSAM structure.

Requirements for converting a batch program to a BMP are:

- A BMP must have an I/O PCB. You can obtain an I/O PCB in batch by specifying the compatibility option in the program specification block (PSB) for the program.

  Related Reading: For more information on the compatibility option in the PSB, see *IMS/ESA Utilities Reference: System*.
- BMPs should issue Checkpoint calls more frequently than batch programs. However, batch programs in a data-sharing environment must also issue Checkpoint calls frequently.

Writing a Batch-Oriented BMP Program

The following description of a batch-oriented BMP program can help you decide if this program is appropriate for your application.

Data That a Batch-Oriented BMP Can Access

Batch-oriented batch message processing (BMP) programs can access full-function, DEDB, DB2, and GSAM databases and MVS files. Batch-oriented BMPs can be run only in a DBCTL environment.

Using a Batch-Oriented BMP

A batch-oriented BMP performs batch processing online. A batch-oriented BMP can be simply a batch program that runs online. You can even run the same program as a BMP or as a batch program.

Recommendations: If the program performs a large number of database updates without issuing checkpoint calls, it may be more efficient to run it as a batch program so that it does not degrade the performance of the online system.

To use batch-oriented BMPs most efficiently, avoid a large amount of batch-type processing online. If you have a BMP that performs time-consuming processing such as report writing and database scanning, schedule it during non-peak hours of processing.

Because BMPs can degrade response times, carefully consider the response time requirements at your installation to decide on the extent to which you will use batch message processing. You should look at the tradeoffs in using BMPs and use them accordingly.

Recovering a Batch-Oriented BMP

Issuing Checkpoint calls is an important part of a batch-oriented BMP's processing, because commit points do not occur automatically as they do in some other types of programs.

Unlike most batch programs, a BMP can share resources with CICS online programs using DBCTL. In addition to committing database changes and providing places from which to restart (as for a batch program), Checkpoint calls release resources locked for the program. For more information on issuing Checkpoint calls, see “Checkpoints in Batch-Oriented BMPs” on page 45.

If a batch-oriented BMP fails, IMS backs out the database updates the program has made since the last commit point. You must restart the program with JCL. If the BMP processes MVS files, you must provide your own method of taking checkpoints and restarting.

Using Data Sharing

If your installation uses data sharing, your programs can participate in IMS data sharing. Under data sharing, CICS online and BMP programs can access the same DL/I database concurrently.

Batch programs in a data-sharing environment can access databases used by other batch programs, and by CICS and IMS online programs. With data sharing, you can share data directly and your program's requests need not go through a mirror transaction.

Related Reading: For more information on sharing a database with an IMS system, see *IMS/ESA Administration Guide: System*.

Scheduling and Terminating a PSB (CICS Online Programs Only)

Before your online program issues any DL/I calls, it must indicate to IMS its intent to use a particular PSB by issuing either a `PCB` call or a `SCHD` command. In addition to indicating which PSB your program will use, the `PCB` call obtains the address of the PCBs in the PSB. When you no longer need a PSB, you can terminate it using the `TERM` request. The rest of this section describes the use of the `TERM` request and how it can affect your system.

In a CICS online program, you use a `PCB` call or `SCHD` command (for command-level programs) to obtain the PSB for your program. Because CICS releases the PSB your program uses when the transaction ends, your program need not explicitly terminate the PSB. You should use a terminate request only if you want to:

- Use a different PSB
- Commit all the database updates and establish a logical unit of work for backing out updates
- Free IMS resources for use by other CICS tasks

A terminate request causes a CICS sync point, and a CICS sync point terminates the PSB. For more information about CICS recovery concepts, see the appropriate CICS publication.

Only use the terminate request for the above reasons. Do not use terminate requests for other reasons because:

- A terminate request forces a CICS sync point. This sync point releases all recoverable resources and IMS database resources that were enqueued for this task.

  If the program continues to update other CICS resources after the terminate request and then terminates abnormally, only those resources that were updated after the terminate request are backed out. Any IMS changes made by the program are not backed out.
- IMS lock management detects deadlocks that occur if two transactions are waiting for segments held by the other.

When a deadlock is detected, one transaction is abnormally terminated. Database changes are backed out to the last TERM request. If a TERM request or CICS sync point was issued prior to the deadlock, CICS does not restart the transaction.

Related Reading: For a complete description of transaction restart considerations, see *CICS/ESA Recovery and Restart Guide*.

- Issuing a terminate request causes additional logging.
- If the terminal output requests are issued after a terminate request and the transaction fails at this point, the terminal operator does not receive the message.

  The terminal operator may assume that the entire transaction failed, and reenter the input, thus repeating the updates that were made before the terminate request. These updates were not backed out.

Linking and Passing Control to Other Programs (CICS Online Programs Only)

Use CICS to link your program to other programs without losing access to the facilities acquired in the linking program, as in the following examples:

- You could schedule a PSB and then link to another program using a LINK command. On return from that program, the PSB is still scheduled.
- Similarly, you could pass control to another program using the XCTL command, and the PSB remains scheduled until that program issues an EXEC CICS RETURN statement. However, when you pass control to another program using XCTL, the working storage of the program passing control is lost. If you want to retain the working storage for use by the program being linked to, you must pass the information in the COMMAREA.

Recommendation: To simplify your work, instead of linking to another program, you can issue all DL/I requests from one program module. This helps to keep the programming simple and easy to maintain.

Terminating a PSB or issuing a sync point affects the linking program. For example, a terminate request or sync point that is issued in the program that was linked causes the release of CICS resources enqueued in the linking program.

Maximizing the Performance of Your CICS System

When you write programs that share data with other programs (for example, a program that will participate in IMS data sharing or a BMP), be aware of how your program affects the performance of the online system. This section explains some things you can do to minimize the effect your program has on that performance.

A BMP program, in particular, can affect the performance of the CICS online transactions. This is because BMP programs usually make a larger number of database updates than CICS online transactions, and a BMP program is more likely to hold segments that CICS online programs need. Limit the number of segments held by a BMP program, so CICS online programs need not wait to acquire them.

One way to limit the number of segments held by a BMP or batch program that participates in IMS data sharing is to issue checkpoint requests in your program to commit database changes and release segments held by the program. When deciding how often to issue Checkpoint requests, you can use one or more of the following techniques:

- Divide the program into small logical units of work, and issue a Checkpoint call at the end of each unit.
- Issue a Checkpoint call after a certain number of DL/I requests have been issued, or after a certain number of transactions are processed.

In CICS online programs, release segments for use by other transactions to maximize the performance of your online system. (Ordinarily, database changes are committed and segments are released only when control is returned to CICS.) To more quickly free resources for use by other transactions, you can issue a `TERM` request to terminate the PSB. However, less processing overhead generally occurs if the PSB is terminated when control is returned to CICS.

Programming Integrity and Database Recovery Considerations

This section explains how IMS and CICS protect data integrity for CICS online programs, and how you can plan ahead for recovering batch and BMP programs.

How IMS Protects Data Integrity for Your Program (CICS Online Programs)

IMS protects the integrity of the database for programs that share data by:

- Preventing other application programs with update capability from accessing any segments in the database record your program is processing, until your program finishes with that record and moves to a new database record in the same database.
- Preventing other application programs from accessing segments that your program deletes, replaces, or inserts, until your program reaches a sync point. When your program reaches a sync point, the changes your program has made to the database become permanent, and the changed data becomes available to other application programs.

  Exception: If PROCOPT=GO has been defined during PSBGEN for your program, your program can access segments that have been updated but not committed by another program.
- Backing out database updates made by an application program that terminates abnormally.

You may also want to protect the data your program accesses by retaining segments for the sole use of your program until your program reaches a sync point—even if you do not update the segments. (Ordinarily, if you do not update the segments, IMS releases them when your program moves to a new database record.) You can use the Q command code to reserve segments for the exclusive use of your program. You should use this option only when necessary because it makes data unavailable to other programs and can have an impact on performance.

Recovering Databases Accessed by Batch and BMP Programs

This section describes the planning you must do for recovering databases accessed by batch or BMP programs. CICS recovers databases accessed by CICS online programs in the same way it handles other recoverable CICS resources. For example, if an IMS transaction terminates abnormally, CICS and IMS back out all database updates to the last sync point.

Integrity and Recovery Considerations

For batch or BMP programs, do the following:

- Take checkpoints in your program to commit database changes and provide places from which your program can be restarted.
- Provide the code for or issue a request to restart your program.

You may also want to back out the database changes that have been made by a batch program that has not yet committed these changes.

To perform these tasks, you use system service calls, described in more detail in the appropriate application programming book for your environment.

Requesting an I/O PCB in Batch Programs

For your program to successfully issue any system service request, an I/O PCB must have been previously requested. See *IMS/ESA Application Programming: Database Manager* for details on how to request an I/O PCB in your program.

Taking Checkpoints in Batch and BMP Programs

Taking checkpoints in batch and BMP programs is important for two reasons:

Recovery

Checkpoints establish places in your program from which your program could be restarted, in the event of a program or system failure. If your program abnormally terminates after issuing a Checkpoint request, database changes will be backed out to the point at which the Checkpoint request was issued.

Integrity

Checkpoints also commit the changes your program has made to the database.

In addition to providing places from which to restart your program and committing database changes, issuing Checkpoint calls in a BMP program or in a program participating in IMS data sharing releases database segments for use by other programs.

When a batch or BMP program issues a Checkpoint request, IMS writes a record containing a checkpoint ID to the IMS/ESA system log.

When your application program reaches a point during its execution where you want to make sure that all changes made to that point have been physically entered in the database, issue a Checkpoint request. If some condition causes your program to fail before its execution is complete, the database must be restored to its original state. The changes made to the database must be backed out so that the database is not left in a partially updated condition for access by other application programs.

If your program runs a long time, you can reduce the number of changes that must be backed out by taking checkpoints in your program. Then, if your program terminates abnormally, only the database updates that occurred after the checkpoint must be backed out. You can also restart the program from the point at which you issued the Checkpoint request, instead of having to restart it from the beginning.

Issuing a Checkpoint call cancels your position in the database.

Issue a Checkpoint call just before issuing a Get Unique call, which reestablishes your position in the database record after the checkpoint is taken.

The Kinds of Checkpoints You Can Use: The two kinds of checkpoint calls are: **basic** and **symbolic**. Both kinds commit your program's changes to the database and establish places from which your program can be restarted:

Batch and BMP programs can issue basic Checkpoint calls using the `CHKP` call. When you use basic Checkpoint calls, you must provide the code for restarting the program after an abnormal termination.

Batch and BMP programs can also issue symbolic Checkpoint calls. You can issue a symbolic Checkpoint call by using the `CHKP` call. Like the basic Checkpoint call, the symbolic Checkpoint call commits changes to the database and establishes places from which the program can be restarted. In addition, the symbolic Checkpoint call:

- Works with the Extended Restart call to simplify program restart and recovery.
- Lets you specify as many as seven data areas in the program to be checkpointed. When you restart the program, the restart call restores these areas to the way they were when the program terminated abnormally.

Specifying a Checkpoint ID: Each Checkpoint call your program issues must have an identification, or ID. Checkpoint IDs must be 8 bytes in length and should contain printable EBCDIC characters.

When you want to restart your program, you can 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 then searches for checkpoint information with an ID matching the one you have supplied. The first matching ID that IMS 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 cannot be sure that IMS will restart your program from the checkpoint you specified.

One way to make sure that checkpoint IDs are unique within and among programs is to construct IDs in the following order:

- Three bytes of information that uniquely identifies your program.
- Five bytes of information that serves as the ID within the program, for example, a value that is increased by 1 for each Checkpoint command or call, or a portion of the system time obtained at program start by issuing the TIME macro.

Specifying Checkpoint Frequency: To determine the frequency of Checkpoint requests, you must consider the type of program and its performance characteristics.

In Batch Programs: When deciding how often to issue Checkpoint requests in a batch program, you should consider the time required to back out and reprocess the program after a failure. For example, if you anticipate that the processing your program performs will take a long time to back out, you should establish checkpoints more frequently.

If you might back out of the entire program, issue the Checkpoint request at the very beginning of the program. IMS backs out the database updates to the checkpoint you specify.

Integrity and Recovery Considerations

In a data-sharing environment, also consider the impact of sharing resources with other programs on your online system. You should issue Checkpoint calls more frequently in a batch program that shares data with online programs, to minimize resource contention.

It is a good idea to design all batch programs with checkpoint and restart in mind. Although the checkpoint support may not be needed initially, it is easier to incorporate Checkpoint calls initially than to try to fit them in later. If the Checkpoint calls are incorporated, it is easier to convert batch programs to BMP programs or to batch programs that use data sharing.

In BMP Programs: When deciding how often to issue Checkpoint requests in a BMP program, consider the performance of your CICS online system. Because these programs share resources with CICS online transactions, issue Checkpoint requests to release segments so CICS online programs need not wait to acquire them. “Maximizing the Performance of Your CICS System” on page 60 explains this in more detail.

Printing Checkpoint Log Records: You can print checkpoint log records by using the IMS File Select and Formatting Print Program (DFSERA10). With this utility, you can select and print log records based on their type, the data they contain, or their sequential positions in the data set. Checkpoint records are type 18 log records. *IMS/ESA Utilities Reference: System* describes this program.

Backing Out Database Changes

If your program terminates abnormally, the database must be restored to its previous state and uncommitted changes must be backed out. Changes made by a BMP or CICS online program are automatically backed out. Database changes made by a batch program might or might not be backed out, depending on whether your system log is on DASD.

For a Batch Program: What happens when a batch program terminates abnormally and how you recover the database depend on the storage medium for the system log. Installations can specify that the system log is to be stored on either DASD or on tape.

When the system log is on DASD

The installation can specify that IMS is to dynamically back out the changes that a batch program has made to the database since its last commit point by coding BKO=Y in the JCL. IMS performs dynamic backout for a batch program when an IMS-detected failure occurs, such as when a deadlock is detected (for batch programs that share data).

DASD logging also makes it possible for batch programs to issue the rollback (`ROLB`) system service request, in addition to `ROLL`. The `ROLB` request causes IMS to dynamically back out the changes the program has made to the database since its last commit point, and then to return control to the application program.

Dynamically backing out database changes has the following advantages:

– Data accessed by the program that failed is immediately available to other programs. Otherwise, if batch backout is not used, data is not available to other programs until the IMS Batch Backout utility has been run to back out the database changes.

– If two programs are deadlocked, one of the programs can continue processing. Otherwise, if batch backout is not used, both programs will fail. (This applies only to batch programs that share data.)

Instead of using dynamic backout, you can run the IMS Batch Backout utility to back out changes.

When the system log is on tape

If a batch application program terminates abnormally and the system log is stored on tape, you must use the IMS Batch Backout utility to back out the program's changes to the database.

Related Reading: For more information, see *IMS/ESA Utilities Reference: Database Manager*.

For BMP Programs: If your program terminates abnormally, the changes the program has made since the last commit point are backed out. If a system failure occurs, or if the CICS control region or DBCTL terminates abnormally, DBCTL emergency restart backs out all changes made by the program since the last commit point. You need not use the IMS Batch Backout utility because DBCTL backs out the changes. If you need to back out all changes, you can use the ROLL system service call to dynamically back out database changes.

Restarting Your Program

If you issue symbolic Checkpoint calls (for batch and BMP programs), you can use the Extended Restart system service request (XRST) to restart your program after an abnormal termination. The XRST call restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last Checkpoint request the program issued before terminating abnormally.

If you use basic Checkpoint calls (for batch and BMP programs), you must provide the necessary code to restart the program from the latest checkpoint in the event that it terminates abnormally.

One way to restart the program from the latest checkpoint is to store repositioning data in an HDAM database. Your program writes a database record containing repositioning information to the HDAM database. It updates this record at intervals. When the program terminates, the database record is deleted. At the completion of the XRST call, the I/O area always contains a checkpoint ID used by the restart. Normally, XRST will return the 8-byte symbolic checkpoint ID, followed by 4 blanks. If the 8-byte ID consists of all blanks, then XRST will return the 14-byte timestamp ID. Also, check the status code in the PCB. The only successful status code for an XRST call is a row of blanks.

Data Availability Considerations

Unfortunately, the data that a program needs to access may sometimes be unavailable. This section describes the situations where data is unavailable, whether a program is scheduled in these situations, and the functions your program might need to use when data is not available.

Unavailability of a Database

The conditions that make an entire database unavailable for both read and update are the following:

- A STOP command has been issued for the database.
- A DBRECOVERY (DBR) command has been issued for the database.
- DBRC authorization for the database has failed.

The conditions that make a database available for read but not for update are:

- A DBDUMP command has been issued for the database.
- The database access value is RD (read).

In a data-sharing environment, the command or error that created any of the above conditions may have originated on the other system which is sharing data.

Whether a program is scheduled or whether an executing program can schedule a PSB when the database is unavailable depends on the type of program and the environment:

- A batch program

  IMS does not schedule a batch program when one of the databases that the program can access is not available.

  In a non-data sharing environment, DBRC authorization for a database may fail because the database is currently authorized to a DB/DC environment. In a data-sharing environment, a CICS or a DBCTL master terminal global command to recover a database or to dump a database may make the database unavailable to a batch program.
- An online or BMP program in the DBCTL environment.

  When a program executing in this environment attempts to schedule with a PSB containing one or more full-function databases that are unavailable, the scheduling is allowed. If the program does not attempt to access the unavailable database, it can function normally. If it does attempt to access the database, the result is the same as when the database is available but some of the data in it is not available.

Unavailability of Some Data in a Database

In addition to the situation where the entire database is unavailable, there are other situations where a limited amount of data is unavailable. One example is a failure situation involving data sharing where the IMS system knows which locks were held by a sharing IMS at the time the sharing IMS system failed. This IMS system continues to use the database but rejects access to the data that the failed IMS system held locked at the time of failure.

Let's say that a batch program, an online program, or a BMP program are operating in the DBCTL environment. If so, the online or BMP programs may have been scheduled when an entire database was not available. The following options apply to these programs when they attempt to access data and either the entire database is unavailable or only some of the data in the database is unavailable.

Programs executing in these environments have an option of being sensitive or insensitive to data unavailability.

- When the program is insensitive to data unavailability and attempts to access unavailable data, the program fails with a 3303 abend. For online programs, this is a pseudoabend. For batch programs, it is a real abend. However, if the database is unavailable because dynamic allocation failed, a call results in an AI (unable to open) status code.
- When the program is sensitive to data unavailability and attempts to access unavailable data, IMS returns a status code indicating that it could not process the request. The program can then take the appropriate action. A facility exists for the program to then initiate the same action that IMS would have taken if the program had been insensitive to unavailable data.

The program issues the `INIT` call or `ACCEPT STATUS GROUP A` command to inform IMS that it is sensitive to unavailable data and can accept the status codes issued when the program attempts to access such data. The `INIT` request can also be used to determine data availability for each PCB in the PSB.

The SETS or SETU and ROLS Functions

The `SETS` or `SETU` and `ROLS` requests allow an application to define multiple points at which to preserve the state of full-function databases. The application can then return to these points at a later time. By issuing a `SETS` or `SETU` request before initiating a set of DL/I requests to perform a function, the program can later issue the `ROLS` request if it cannot complete the function due possibly to data unavailability.

`ROLS` allows the program to roll back its IMS activity to the state prior to the `SETS` or `SETU` call.

Restriction:`SETS` or `SETU` and `ROLS` only roll back the IMS updates. They do not roll back the updates made using CICS file control or transient data.

Additionally, you can use the `ROLS` call or command to undo all database update activity since the last checkpoint.

Use of STAE or ESTAE and SPIE in IMS Batch Programs

This section describes using STAE or ESTAE and SPIE in an IMS batch program. For information on using these routines in CICS online programs, refer to CICS manuals.

IMS uses STAE or ESTAE routines in the IMS batch regions to ensure that database logging and various resource cleanup functions are completed. Two important aspects of the STAE or ESTAE facility are that:

- IMS relies on its STAE or ESTAE facility to ensure database integrity and resource control.
- The STAE or ESTAE facility is also available to the application program.

Because of these two factors, be sure you clearly understand the relationship between the program and the STAE or ESTAE facility.

Generally, do not use the STAE or ESTAE facility in your batch application program. However, if you believe that the STAE or ESTAE facility is required, you must observe the following basic rules:

- When the environment supports STAE or ESTAE processing, the application program STAE or ESTAE routines always get control before the IMS STAE or ESTAE routines. Therefore, you must ensure that the IMS STAE or ESTAE exit routines receive control by observing the following procedures in your application program:
  - Establish the STAE or ESTAE routine only once and always before the first DL/I call.
  - When using the STAE or ESTAE facility, the application program must not alter the IMS abend code.
  - Do not use the RETRY option when exiting from the STAE or ESTAE routine. Instead, return a CONTINUE-WITH-TERMINATION indicator at the end of the STAE or ESTAE processing. If your application program does specify the

RETRY option, be aware that IMS STAE or ESTAE exit routines will not get control to perform cleanup. Therefore, system and database integrity may be compromised.

  - For PL/I use of STAE and SPIE, see the description of IMS considerations in *OS PL/I Version 2 Programming Guide*.
  - For PL/I, COBOL, and C/MVS, if you are using the AIBTDLI interface in a non-Language Environment enabled environment, you must specify NOSTAE or NOSPIE. However, in a Language Environment Version 1.2 or later enabled environment, the NOSTAE and NOSPIE restriction is removed.
- The application program STAE/ESTAE exit routine must not issue DL/I calls because the original abend may have been caused by a problem between the application and IMS. This would result in recursive entry to STAE/ESTAE with potential loss of database integrity or in problems taking a checkpoint.

Dynamic Allocation for IMS Databases

Use the dynamic allocation function to specify the JCL information for IMS databases in a library instead of in the JCL of each batch job or in the JCL for DBCTL. If your installation uses dynamic allocation, do not include JCL DD statements for any database data sets that have been defined for dynamic allocation. Check with the database administrator (DBA) or comparable specialist at your installation to determine which databases have been defined for dynamic allocation.

Related Reading: For additional information on the definitions for dynamic allocation, see the DFSMDA macro in *IMS/ESA Utilities Reference: System*.

Chapter 5. Gathering Requirements for Database Options

This chapter guides you in gathering information that the database administrator (DBA) can use in designing a database and implementing that design. After designing hierarchies for the databases that your application will access, the DBA evaluates database options in terms of which options will best meet application requirements. Whether these options are used depends on the collected requirements of the applications. To design an efficient database, the DBA needs information about the individual applications. This chapter describes the type of information that can be helpful to the DBA, how the information you are gathering relates to different database options, and the different aspects of your application that you need to examine.

In this Chapter:

Analyzing Data Access

The DBA chooses a type of database, based on how the majority of programs that use the database will access the data. IMS databases are categorized according to the access method used. The following is a list of the types of databases that can be defined:

HDAM (Hierarchical Direct Access Method)

HIDAM (Hierarchical Indexed Direct Access Method)

MSDB (Main Storage Database)

DEDB (Data Entry Database)

HSAM (Hierarchical Sequential Access Method)

HISAM (Hierarchical Indexed Sequential Access Method)

GSAM (Generalized Sequential Access Method)

SHSAM (Simple Hierarchical Sequential Access Method)

SHISAM (Simple Hierarchical Indexed Sequential Access Method)

Some of the information that you can gather to help the DBA with this decision answers questions like the following:

- To access a database 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 database record are the dependents of the root segment. How will each program access the segments within each database record?

  Directly

  Sequentially

  Both

It is important to note the distinction between accessing a database record and accessing segments within the record. A program could access database records sequentially, but after the program is within a record, the program might access the segments directly. These are different, and can influence the choice of access method.

- To what extent will the program update the database?

  By adding new database records?

  By adding new segments to existing database records?

  By deleting segments or database records?

Again, note the difference between updating a database record and updating a segment within the database record.

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, IMS can find any database record that you want, regardless of the sequence of database records in the database.

IMS full function has two direct access methods.

- HDAM processes data directly by using a randomizing routine to store and locate root segments.
- HIDAM uses 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 database record. By following pointers, IMS can access a path of segments without passing through all the segments in the preceding paths.

Some of the requirements that direct access satisfies are:

- Fast direct processing of roots using an index or a randomizing routine
- Sequential processing of database records with HIDAM using the index
- Fast access to a path of segments using pointers

In addition, when you delete data from a direct-access database, the new space is available almost immediately. This gives you efficient space utilization; therefore, reorganization of the database is often unnecessary. Direct access methods internally maintain their own pointers and addresses.

A disadvantage of direct access is that you have a larger IMS overhead because of the pointers. But if direct access fulfills your data access requirements, it is more efficient than using a sequential access method.

Primarily Direct Processing: HDAM

HDAM is efficient for a database that is usually accessed directly but sometimes sequentially. HDAM uses a randomizing routine to locate its root segments and then chains dependent segments together according to the pointer options chosen. The MVS access methods that HDAM can use are Virtual Storage Access Method (VSAM) and Overflow Storage Access Method (OSAM).

The requirements that HDAM satisfies are:

- Direct access of roots by root keys because HDAM uses a randomizing routine to locate root segments
- Direct access of paths of dependents

- Adding new database records and new segments because the new data goes into the nearest available space
- Deleting database records and segments because the space created by a deletion can be used by any new segment

HDAM Characteristics: An HDAM database:

- Can store root segments anywhere. Root segments do not need to be in sequence because the randomizing routine locates them.
- Uses a randomizing routine to locate the relative block number and root anchor point (RAP) within the block that points to the root segment.
- Accesses the RAPs from which the roots are chained in physical sequence. Then the root segments that are chained from the root anchors are returned. Therefore, sequential retrieval of root segments from HDAM is not based on the results of the randomizing routine and is not in key sequence unless the randomizing routine put them into key sequence.
- May not give the desired result for some calls unless the randomizing module causes the physical sequence of root segments to be in the key sequence. For example, a `GU` call for a root segment that is qualified as less than or equal to a root key value would scan in physical sequence for the first RAP of the first block. This may result in a not-found condition, even though segments meeting the qualification do exist.

For dependent segments, an HDAM database:

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

An Overview of How HDAM Works: This section contains diagnosis, modification, or tuning information.

When a database record is stored in an HDAM database, 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 database, 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 specified RAP number gives HDAM the location of the root within a physical block.

Although HDAM can place roots and dependents anywhere in the database, it is 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, but it also depends on other 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 database record in two ways:
  - In hierarchic sequence, starting with the root
  - In parent-to-dependent sequence, with parents having pointers to each of their paths of dependents

To use HDAM for sequential access of database records by root key, you need to use a secondary index or a randomizing routine that stores roots in physical key sequence.

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 MVS access methods it can use are VSAM and OSAM. The specific requirements that HIDAM satisfies are:

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

HIDAM can satisfy most processing requirements that involve an even mixture of direct and sequential processing. However, HIDAM is not very efficient with sequential access of dependents.

HIDAM Characteristics: For root segments, a HIDAM database:

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

For dependent segments, a HIDAM database:

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

An Overview of How HIDAM Works: This section contains diagnosis, modification, or tuning information.

HIDAM uses two databases. The primary database holds the data. An index database contains entries for all of the root segments in order by their key fields. For each key entry, the index database contains the address of that root segment in the primary database.

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 and then goes to the primary database to find the root.

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

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

If you plan to process database records sequentially, you can specify special pointers in the DBD for the database so that IMS does not need to go to the index to locate the next root segment. These pointers chain the roots together. If you do not chain roots together, HIDAM always goes to the index to locate a root segment. When you process database 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 in other root segments, to find the root segment you have requested.

Main Storage Database: MSDB

Use MSDBs to store an installation's most frequently-accessed data. MSDBs are suitable for applications such as general ledger applications in the banking industry.

MSDB Characteristics: MSDBs reside in virtual storage, enabling application programs to avoid the I/O activity that is required to access them. The two kinds of MSDBs are terminal-related and non-terminal-related.

In a terminal-related MSDB, each segment is owned by one terminal, and each terminal owns only one segment. One use for this type of MSDB is an application in which each segment contains data associated with a logical terminal. In this type of application, the program can read the data (perhaps for reporting purposes), but cannot update it. A non-terminal-related MSDB stores data that is needed by many users during the same time period. It can be updated and read from all terminals (for example, a real time inventory control application, where reduction of inventory can be noted from many cash registers).

An Overview of How MSDBs Work:

Diagnosis, Modification or Tuning Information

MSDB segments are stored as root segments only. Only one type of pointer, the forward chain pointer, is used. This pointer connects the segment records in the database.

End of Diagnosis, Modification or Tuning Information

Data Entry Database: DEDB

DEDBs are designed to provide access to and efficient storage for large volumes of data. The primary requirement a DEDB satisfies is a high level of data availability.

DEDB Characteristics: DEDBs are hierarchic databases that can have as many as 15 hierarchic levels, and as many as 127 segment types. They can contain both direct and sequential dependent segments. Because the sequential dependent segments are stored in chronological order as they are committed to the database, they are useful in journaling applications.

DEDBs support a subset of functions and options that are available for a HIDAM or HDAM database. For example, a DEDB does not support indexed access (neither primary index nor secondary index), or logically related segments.

An Overview of How DEDBs Work:

Diagnosis, Modification or Tuning Information

A DEDB can be partitioned into multiple areas, with each area containing a different collection of database records. The data in a DEDB area is stored in a VSAM data set. Root segments are stored in the root-addressable part of an area, with direct dependents stored close to the roots for fast access. Direct dependents that cannot be stored close to their roots are stored in the independent overflow portion of the area. Sequential dependents are stored in the sequential dependent portion at the end of the area so that they can be quickly inserted. Each area data set can have up to seven copies, making the data easily available to application programs.

End of Diagnosis, Modification or Tuning Information

Sequential Access

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

IMS full-function has two sequential access methods. Like the direct access methods, one has an index and the other does not:

- HSAM only processes root segments and dependent segments sequentially.
- 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 database records.

Some of the general requirements that sequential access satisfies are:

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

The three disadvantages of using sequential access methods are:

- Sequential access methods give slower access to the right-most segments in the hierarchy, because HSAM and HISAM must read through all other segments to get to them.
- HISAM requires frequent reorganization to reclaim space from deleted segments and to keep the logical records of a database record physically adjoined.
- You cannot update HSAM databases. You must create a new database to change any of the data.

Sequential Processing Only: HSAM

HSAM is a hierarchic access method that can handle only sequential processing. You can retrieve data from HSAM databases, but you cannot update any of the data. The MVS access methods that HSAM can use are QSAM and BSAM.

HSAM is ideal for the following situations:

- You are using the database to collect (but not update) data or statistics.
- You only plan to process the data sequentially.

HSAM Characteristics: HSAM stores database records in the sequence in which you submit them. You can only process records and dependent segments sequentially, which means the order in which you have loaded them. HSAM stores dependent segments in hierarchic sequence.

An Overview of How HSAM Works:

Diagnosis, Modification or Tuning Information

HSAM databases are very simple databases. The data is stored in hierarchic sequence, one segment after the other, and no pointers or indexes are used.

End of Diagnosis, Modification or Tuning Information

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. 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 database record, you store the remaining segments in an overflow data set. The access methods that HISAM can use are VSAM and OSAM.

HISAM is well-suited for:

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

The situations in which your processing has some of the characteristics above but where HISAM is not necessarily a good choice, occur when:

- You must access dependents directly.
- You have a high number of inserts and deletes.
- Many of the database records exceed average size and must use the overflow data set. The segments that overflow into the overflow data set require additional I/O.

HISAM Characteristics: For database records, HISAM databases:

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

For dependent segments, HISAM databases:

- Start each HISAM database 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 database record does not fit in the primary data set

An Overview of How HISAM Works:

Diagnosis, Modification or Tuning Information

HISAM does not immediately reuse space. When you insert a new segment, HISAM databases shift data to make room for the new segment, and this leaves unused space after deletions. HISAM space is reclaimed when you reorganize a HISAM database.

End of Diagnosis, Modification or Tuning Information

Accessing MVS Files through IMS: GSAM

GSAM enables IMS batch application programs and BMPs to access a sequential MVS data set as a simple database. The MVS access methods that GSAM can use are BSAM and VSAM. A GSAM database is an MVS data set record that is defined as a database record. The record is handled as one unit; it contains no segments or fields and the structure is not hierarchic. GSAM databases can be accessed by MVS, IMS, and CICS.

Analyzing Data Access

In a CICS environment, an application program can access a GSAM database from either a Call DL/I (or EXEC DLI) batch or batch-oriented BMP program. A CICS application cannot, however, use EXEC DLI to process GSAM databases; it must use IMS calls.

You commonly use GSAM to send input to and receive output from batch-oriented BMPs or batch programs. To process a GSAM database, an application program issues calls similar to the ones it issues to process a full-function database. The program can read data sequentially from a GSAM database, and it can send output to a GSAM database.

GSAM is a sequential access method. You can only add records to an output database sequentially.

Accessing IMS Data through MVS: SHSAM and SHISAM

Two database access methods give you simple hierarchic databases that MVS can use as data sets, SHSAM and SHISAM.

These access methods can be particularly helpful when you are converting data from MVS files to an IMS database. SHISAM is indexed and SHSAM is not.

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

SHSAM and SHISAM databases can be accessed by MVS access methods without IMS, which is useful during transitions.

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 application. When the DBA finds a conflict in the way that two or more programs need to access the data, three options are available to solve these problems. Each of the following options solves a different kind of conflict.

- When an application program does not need access to all 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 database.
- 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 applies 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 the fields in the segment but in a different order. If a segment contains fields that the application program does not need to process, using field-level sensitivity enables the program not to process them.

An Example of Field-Level Sensitivity

For example, suppose that a segment containing data about an employee contains the fields shown in Figure 19. These fields are:

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

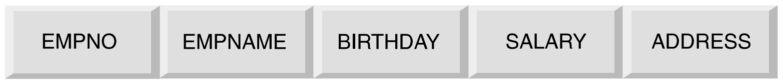


Figure 19. Physical Employee Segment

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 sensitivity for that application, the program would receive only the fields it needed in its I/O area. Figure 20 shows what the program's I/O area would contain.

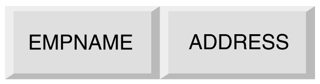


Figure 20. Employee Segment with Field-Level Sensitivity

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 sensitivity is very useful is when new uses of the database 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 both 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 database hierarchy does not 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

To understand these conflicts and how secondary indexing can resolve them, consider the examples of two application programs that process the patient hierarchy, shown in Figure 21. Three segment types in this hierarchy are:

- PATIENT contains three fields: the patient's identification number, name, and 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.

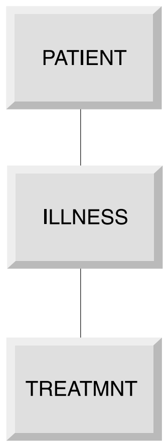


Figure 21. Patient Hierarchy

Using a Different Key

When an application program retrieves a segment from the database, 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.

Example: Suppose you have an online application program that processes requests about whether an individual has ever been to the clinic. If you are not sure whether the person has ever been to the clinic, you will not 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 database 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, IMS must search through all of the PATIENT segments to find the PATIENT segment you have requested. IMS does not 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 database. The separate database is called a *secondary index database*.

Then, if you indicate to IMS that it is to process the PATIENT segments in the patient hierarchy in the sequence of the index entries in the secondary index database, IMS can locate a PATIENT segment if you supply the patient’s name. IMS goes directly to the secondary index and locates the PATIENT index entry with the name you have supplied; the PATIENT index entries are in alphabetical order of the patient names. The index entry is a pointer to the PATIENT segment in the patient hierarchy. IMS can determine whether a PATIENT segment for the name you have supplied exists, and then it can return the segment to the application program if the segment exists. If the requested segment does not exist, IMS indicates this to the application program by returning a not-found status code.

Definitions: Three terms involved in secondary indexing are:

- The *pointer segment* is the index entry in the secondary database that IMS uses to find the segment you have requested. In the example above, the pointer segment is the index entry in the secondary index database that points to the PATIENT segment in the patient hierarchy.
- The *source segment* is the segment that contains the field that you are indexing. In the example above, the source segment is the PATIENT segment in the patient hierarchy, because you are indexing on the name field in the PATIENT segment.
- The *target segment* is the segment in the database that you are processing to which the secondary index points; it is 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.

The PATIENT segment that IMS returns to the application program’s I/O area looks the same as it would if secondary indexing had not been used.

The key feedback area is different. When IMS retrieves a segment without using a secondary index, IMS places the concatenated key of the retrieved segment in the 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 database, the contents of the key feedback area after the request 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 there. Note that the term “key of the pointer segment,” as used here, refers to the key as perceived by the application program—that is, the key does not include subsequence fields.

Example: Suppose index segment A shown in Figure 22 is indexed on a field in segment C. Segment A is the target segment, and segment C is the source segment.

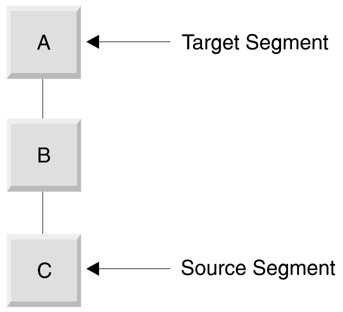


Figure 22. Indexing a Root Segment

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.

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 23 shows the physical database structure and the structure that is created by the secondary index.

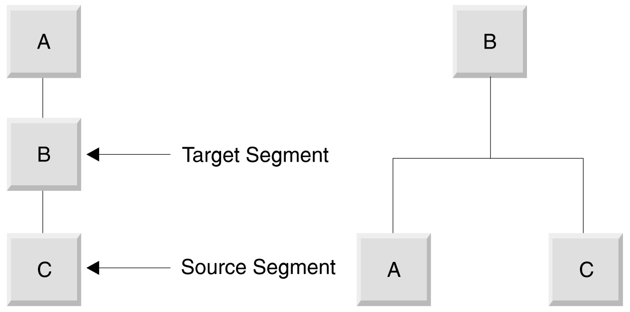


Figure 23. Indexing a Dependent Segment

When you retrieve the segments in the secondary index data structure on the right, IMS returns the following to the key feedback area:

- If you retrieve segment B, the key feedback area contains the key of the pointer segment in the secondary index database.
- 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 meet a certain qualification.

Example: Suppose that the medical clinic 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 does not use a secondary index, the program has to retrieve each PATIENT segment, and then retrieve the ILLNESS segment for each PATIENT segment. The program tests the date in the ILLNESS segment to determine whether 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 of the program 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 IMS the name of the field on which you are indexing the PATIENT segment, and the name of the segment that contains the index field. The application program can then request a PATIENT segment and qualify the request with the date in the ILLNESS segment. The PATIENT segment that is returned 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 is 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 database 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 that possible. Logical relationships can solve the following conflicts:

- 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 is arranged in the hierarchy.

Example: An application program that processes data in a purchasing database also requires access to a segment in a patient database:

- Program A processes information in the patient database 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 database 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 is given.

Understanding Data Structure Conflicts

Figure 24 shows the hierarchies that Program A and Program B require for their processing. Their processing requirements conflict: they both need to have access to the information that is contained in the TREATMNT segment in the patient database. 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 is not information about a patient's treatment; it is information about the disbursement of a medication. To the purchasing database, this is the disbursement segment (DISBURSE).

Figure 24 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.

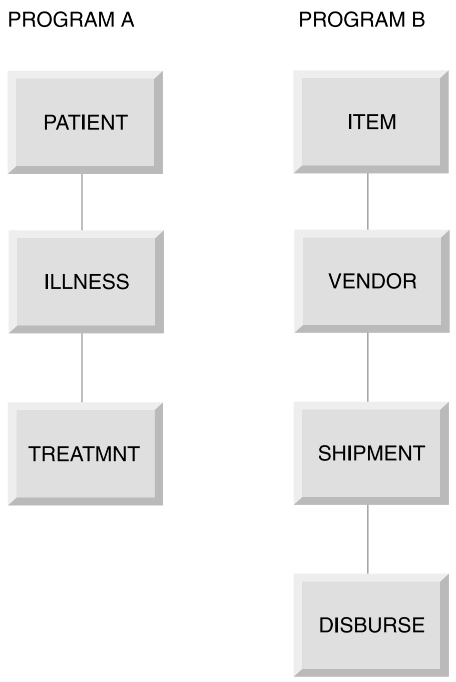


Figure 24. Patient and Inventory Hierarchies

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 database. When IMS receives a request for information in a DISBURSE segment in the purchasing database, IMS goes to the TREATMNT segment in the medical database that is pointed to by the DISBURSE segment. Figure 25 on page 83 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.

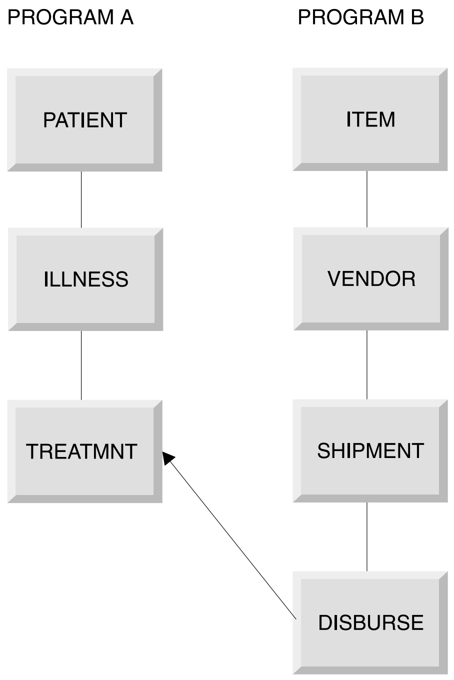


Figure 25. Logical Relationships Example

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 25 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 26 shows the hierarchies for each of these application programs.

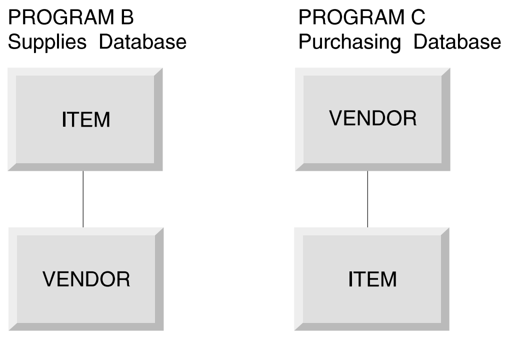


Figure 26. Supplies and Purchasing Hierarchies

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

Figure 27 shows the hierarchies of these two programs.

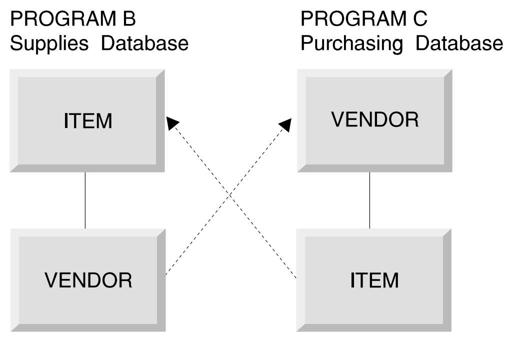


Figure 27. Program B and Program C Hierarchies

If you did not use logical relationships in this situation, you would:

- Keep the same data in both paths, which means that you would be keeping redundant data.
- Have the same disadvantages as separate files of data:
  – You would need to update multiple segments each time one piece of data changed.
  – You would need more storage.

Providing Data Security

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

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

Providing Data Availability

Specifying segment sensitivity and processing options also affects data availability. You should set the specifications so that the PCBs request the fewest SENSEGS and limit the possible processing options. With data availability, a program can continue to access and update segments in the database successfully, even though some parts of the database are unavailable.

The SENSEG statement defines a segment type in the database to which the application program is sensitive. A separate SENSEG statement must exist for each segment type. The segments can physically exist in one database or they can be derived from several physical databases. If an application program is sensitive to a segment that is below the root segment, it must also be sensitive to all segments in the path from the root segment to the sensitive segment.

Related Reading: For more information on using field-level sensitivity for data security and using the SENSEG statement to limit the scope of the PCBs, see *IMS/ESA Administration Guide: Database Manager*.

Keeping a Program from Accessing the Data: Data Sensitivity

An IMS 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 the segments in a particular hierarchy. Segment sensitivity tells IMS which segments in a hierarchy the program is allowed to access.
- **Field-level sensitivity** can keep a program from accessing all the fields that make up a particular segment. Field-level sensitivity tells IMS 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. IMS returns 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 indicate to IMS 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.

Example: Suppose that the patient hierarchy shown in Figure 21 on page 78 belongs to the medical database shown in Figure 28. The patient hierarchy is like a subset of the medical database.

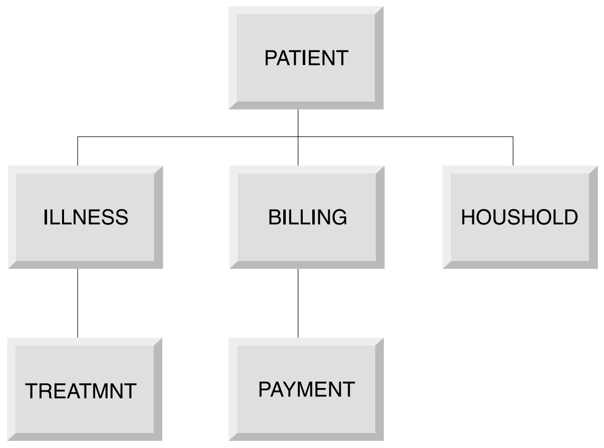


Figure 28. Medical Database Hierarchy

Providing Data Security

To make it possible for an application program to view only the segments PATIENT, ILLNESS, and TREATMNT from the medical database, you specify in the DB PCB that the hierarchy you are defining has these three segment types, and that they are from the medical database. You define the database hierarchy in the DBD; you define the application program's view of the database 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 does not 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 to which you want to restrict access.

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 29, a program must be sensitive to segment B in order to access segment C.

Example: 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), the program is not able to access that segment. Using key sensitivity lets you withhold segment B from the application program while giving the program access to segment B's dependents.

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, IMS does not return that segment; IMS returns only the segment's key with the keys of the other segments that are accessed.

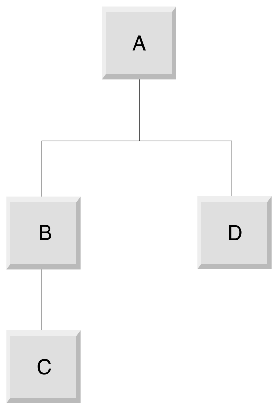


Figure 29. Sample Hierarchy

Preventing a Program from Updating Data: Processing Options

During PCB generation, you can use five options of the PROCOPT parameter (in the DATABASE macro) to indicate to IMS whether your program can read segments in the hierarchy, or whether it can also update segments. From most restrictive to least restrictive, these options are:

G	Your program can read segments.
R	Your program can read and replace segments.
I	Your program can insert segments.
D	Your program can read and delete segments.
A	Your program needs all the processing options. It is equivalent to specifying G, R, I, and D.

Related Reading: See *IMS/ESA Utilities Reference: System* for a thorough description of the processing options.

Processing options provide data security because they limit what a program can do to the hierarchy or to a particular segment. Specifying only the processing options the program requires ensures that the program cannot update any data it is not supposed to. For example, if a program does not need to delete segments from a database, the D option need not be specified.

When an application program retrieves a segment and has any of the just-described processing options, IMS locks the database record for that application. If PROCOPT=G is specified, other programs with the option can concurrently access the database record. If an update processing option (R, I, D, or A) is specified, no other program can concurrently access the same database record. If no updates are performed, the lock is released when the application moves to another database record or, in the case of HDAM, to another anchor point.

E and GO Processing Options

When using block-level or database-level data sharing for online and batch programs, you can use two additional processing options: E and GO.

Related Reading: For detailed information on database and block-level data sharing, refer to *IMS/ESA Administration Guide: System*.

E option: With the E option, your program has exclusive access to the hierarchy or to the segment you use it with. The E option is used in conjunction with the options G, I, D, R, and A. While the E program is running, other programs cannot access that data, but may be able to access segments that are not in the E program’s PCB.

GO option: When your program retrieves a segment with the GO option, IMS does not lock the segment. While the read without integrity program reads the segment, it remains available to other programs. This is because your program can only read the data (termed *read-only*); it is not allowed to update the database. Serialization between the program with PROCOPT=GO and any other update program does not occur; updates to the same data occur simultaneously.

If a segment has been deleted and another segment of the same type has been inserted in the same location, the segment data and all subsequent data that is returned to the application may be from a different database record.

A read-without-integrity program can also retrieve a segment even if another program is updating the segment. This means that the program need not wait for segments that other programs are accessing. If a read-without-integrity program reads data that is being updated by another program, and that program terminates abnormally before reaching the next commit point, the updated segments might contain invalid pointers. If an invalid pointer is detected, the read-without-integrity program terminates abnormally, unless the N or T options were specified with GO.

N option: When you use the N option with GO to access a full-function database or a DEDB, and the segment you are retrieving contains an invalid pointer, IMS returns a GG status code to your program. Your program can then terminate processing, continue processing by reading a different segment, or access the data using a different path. The N option must be specified as PROCOPT=GON, GON, or GONP.

T option: When you use the T option with GO and the segment you are retrieving contains an invalid pointer, the response from an application program depends on whether the program is accessing a full-function or Fast Path database.

For calls to full-function databases, the T option causes DL/I to automatically retry the operation. You can retrieve the updated segment, but only if the updating program has reached a commit point or has had its updates backed out since you last tried to retrieve the segment. If the retry fails, a GG status code is returned to your program.

For calls to Fast Path DEDBs, option T does not cause DL/I to retry the operation. A GG status code is returned. The T option must be specified as PROCOPT=GOT, GOT, or GOTP.

GOx and data integrity: For a very small set of applications and data, PROCOPT=GOx offers some performance and parallelism benefits. However, it does not offer application data integrity. For example, using PROCOPT=GOT in an online environment on a full-function database can cause performance degradation. The T option forces a re-read from DASD, negating the advantage of very large buffer pools and VSAM hiperspace for all currently running applications and shared data. For more information on the GOx processing option for DEDBs, see *IMS/ESA Utilities Reference: System*.

Read without Integrity — What It Means

Database-level sharing of IMS databases provides for sharing of databases between a single update-capable batch or online IMS system and any number of other IMS systems that are reading data that are without integrity.

Read without Integrity

In IMS, programs that use database-level sharing include PROCOPT=GOx in their DBPCBs for that data. For batch jobs, the DBPCB PROCOPTs establish the batch job’s access level for the database. That is, a batch job uses the highest declared intent for a database as the access level for DBRC database authorization. In an online IMS environment, database ACCESS is specified on the DATABASE macro during IMS system definition, and it can be changed using the `/START DB ACCESS=RO` command. Online IMS systems schedule programs with data availability determined by the PROCOPTs within those program PSBs being scheduled. That data availability is therefore limited by the online system’s database access.

The PROCOPT=GON and GOT options (described in “E and GO Processing Options” on page 87) provide certain limited PCB status code retry for some recognizable pointer errors, within the data that is being read without integrity. In some cases, dependent segment updates, occurring asynchronously to the read-without-integrity IMS instance, do not interfere with the program that is reading that data without integrity. However, update activity to an average database does not always allow a read-without-integrity IMS system to recognize a data problem.

What It Means

Each IMS batch or online instance has OSAM and VSAM buffer pools defined for it. Without locking to serialize concurrent updates that are occurring in another IMS instance, a read without integrity from a database data set fetches a copy of a block or CI into the buffer pool in storage. Blocks or CIs in the buffer pool can remain there a long time. Subsequent read without integrity of other blocks or CIs can then fetch more recent data. Data hierarchies and other data relationships between these different blocks or CIs can be inconsistent.

For example, consider an index database (VSAM KSDS), which has an index component and a data component. The index component contains only hierarchic control information, relating to the data component CI where a given keyed record is located. Think of this as the index component CI's way of maintaining the high key in each data component CI. Inserting a keyed record into a KSDS data component CI that is already full causes a CI split. That is, some portion of the records in the existing CI are moved to a new CI, and the index component is adjusted to point to the new CI.

Example: Suppose the index CI shows the high key in the first data CI as KEY100, and a split occurs. The split moves keys KEY051 through KEY100 to a new CI; the index CI now shows the high key in the first data CI as KEY050, and another entry shows the high key in the new CI as KEY100.

A program that is reading is without integrity, which already read the "old" index component CI into its buffer pool (high key KEY100), does not point to the newly created data CI and does not attempt to access it. More specifically, keyed records that exist in a KSDS at the time a read-without-integrity program starts might never be seen. In this example, KEY051 through KEY100 are no longer in the first data CI even though the "old" copy of the index CI in the buffer pool still indicates that any existing keys up to KEY100 are in the first data CI.

Hypothetical cases also exist where the deletion of a dependent segment and the insertion of that same segment type under a different root, placed in the same physical location as the deleted segment, can cause simple Get Next processing to give the appearance of only one root in the database. For example, accessing the segments under the first root in the database down to a level-06 segment (which had been deleted from the first root and is now logically under the last root) would then reflect data from the other root. The next and subsequent Get Next calls retrieve segments from the other root.

Read-only (PROCOPT=GO) processing does not provide data integrity.

Data Set Extensions

IMS instances with database-level sharing can open a database for read without integrity. After the database is opened, another program that is updating that database can make changes to the data. These changes might result in logical and physical extensions to the database data set. Because the read-without-integrity program is not aware of these extensions, problems with the RBA (beyond end-of-data) can occur.

Chapter 6. Gathering Requirements for Message Processing Options

One of the tasks of application design is providing information about your application's requirements to the people in charge of designing and administering the IMS system at your installation. This chapter describes the information you should provide, and why this information is important.

Restriction: This chapter applies to DB/DC and DCCTL environments only.

In this Chapter:

Identifying Online Security Requirements

Security in an online system means protecting the data from unauthorized use via terminals. It also means preventing unauthorized use of both the IMS system and the application programs that access the database. For example, you do not want a program that processes paychecks to be available to everyone who can access the system.

The security mechanisms that IMS provides are signon, terminal, and password security.

Related Reading: For an explanation of how to establish these types of security, see *IMS/ESA Administration Guide: System*.

Limiting Access to Specific Individuals: Signon Security

Signon security is available through Resource Access Control Facility (RACF) or a user-written security exit routine. With signon security, individuals who want to use IMS must be defined to RACF or its equivalent before they are allowed access.

When a person signs on to IMS, RACF or installation security exits verify that the person is authorized to use IMS before access to IMS-controlled resources is allowed. This signon security is provided by the `/SIGN ON` command. You can also limit the transaction codes and commands that individuals are allowed to enter. You do this by associating an individual's user identification (USERID) with the transaction codes and commands.

LU 6.2 transactions contain the USERID.

Related Reading: See *IMS/ESA Administration Guide: Transaction Manager* for additional information on security.

Limiting Access for Specific Terminals: Terminal Security

Use terminal security 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.

Online Security Requirements

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 you wanted to restrict access to this application even more, you could associate the paycheck transaction code with only one logical terminal. To enter that transaction code, a user needs to be at a physical terminal that is associated with that logical terminal.

Restriction: If you are using the shared-queues option, static control blocks representing the resources needed for the security check need to be available in the IMS system where the security check is being made. Otherwise, the security check is bypassed.

Related Reading: For more information on shared queues, see *IMS/ESA Administration Guide: Transaction Manager*.

Limiting Access to the Program: Password Security

Another way you can protect the application program is to require a password when a person enters the transaction code that is associated with the application program you want to protect. If you use only password security, the person entering a particular transaction code must also enter the password of the transaction before IMS processes the transaction.

If you use password security with terminal security, you can 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.

Related Reading: Password security for transactions is only supported if the transactions that are needed for the security check are defined in the IMS system where the security check is being made. Otherwise, the security check is bypassed.

Allowing Access to Security Data: Authorization Security

RACF has a data set that installations can use to store user-unique information. The AUTH call gives application programs access to the RACF data set security data, and a way to control access to application-defined resources. Thus, application programs can obtain the security information about a particular user.

How IMS Security Relates to DB2 Security

An important part of DB2 security is the authorization ID. The authorization ID that IMS uses for a program or a user at a terminal depends on the kind of security that is used and the kind of program that is running. For MPPs, IFPs, and transaction-oriented BMPs, the authorization ID depends on the type of IMS security:

- If the installation requires signon, IMS passes the USERID and group name that are signed-on to DB2.
- If the installation does not use signon, DB2 uses the name of the originating logical terminal as the authorization ID.

For batch-oriented BMPs, the authorization ID is dependent on the value specified for the BMPUSID= keyword in the DFSDCxxx PROCLIB member:

- If BMPUSID=USERID is specified, the value from the USER= keyword on the JOB statement is used.
- If USER= is not specified on the JOB statement, the program's PSB name is used.
- If BMPUSID=PSBNAME is specified, or if BMPUSID= is not specified at all, the program's PSB name is used.

Supplying Security Information

When you evaluate your application in terms of its security requirements, you need to look at each program individually. When you have done this, you can supply the following information to security personnel at your installation.

- For programs that require signon security:
  - List the individuals who should be able to access IMS.
- 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 are 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 require passwords.
- For commands that require security:
  - List the commands that require signon or password security.

Analyzing Screen and Message Formats

When an application program communicates with a terminal, an editing procedure 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 will edit your program's messages are based on how your data 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. (The I/O area contains the segments being processed by the application program.)

To supply information that will be helpful in these decisions, you should be familiar with how IMS edits messages. IMS has two editing procedures:

- **Message Format Service (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 application programs. Basic edit removes control characters from input messages and inserts the control characters you specify in output messages to the terminal.

Related Reading: For information on defining IMS editing procedures and on other design considerations for IMS networks, see *IMS/ESA Administration Guide: Transaction Manager*.

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.

The two control blocks that describe input messages to IMS are:

- The device input format (DIF) describes to IMS what the input message is to look like when it is entered at the terminal.
- The message input descriptor (MID) tells IMS how the application program expects to receive the input message in its I/O area.

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

The two control blocks that describe output messages to IMS are:

- The message output descriptor (MOD) tells IMS what the output message is to look like in the program's I/O area.
- The device output format (DOF) tells IMS how the message should appear on the terminal.

To define the MFS control blocks for an application program, you need 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.

Related Reading: *IMS/ESA Application Programming: Transaction Manager*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 are using.

If your application will use basic edit, you should describe how you want the data to be presented at the terminal, and what it is to 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.

Provide the following information about your program's editing requirements:

- 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 airline agent wants to reserve seats on a particular flight, the screen that asks for this information might 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 are processing is only one consideration when you analyze how you want the data presented at the terminal. In addition, you should weigh the needs of the person at the terminal (the human factors aspects in your application) against the effect of the screen design on the efficiency of the application program (the performance factors in the application program). Unfortunately, sometimes a trade-off between human factors and performance factors exists. A screen design that is easily understood and used by the person at the terminal may not be the design that gives the application program its best performance. Your first concern should be that you are 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 is one that makes it as easy as possible for that person to interact with IMS. 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 do not 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 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 does not 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 are working with.

Gathering Requirements for Conversational Processing

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. Only MPPs can be conversational programs; Fast Path programs and BMPs cannot be conversational.

Definition: Conversational processing means that the person at the terminal can communicate with the application program.

What Happens in a Conversation

During a *conversation*, the person at the terminal enters a request, receives the information from IMS, and enters another request. Although it is not apparent to the person at the terminal, a conversation can be processed by several application programs or by one application program.

Definition: A conversation is a dialog between a person at a terminal and IMS through one or more application programs.

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

In the preceding airline example, the first program might save the flight number and the names of the people traveling, and then pass control to another application program to reserve seats for those people on that flight. The first program saves this information in the SPA. If the second application program did not have the flight number and names of the people traveling, it would not be able to do its processing.

Designing a Conversation

The first part of designing a conversation is to design the flow of the conversation. If the requests from the person at the terminal are to be processed by only one application program, you need only to design that program. If the conversation should be processed by several application programs, you need to decide which steps of the conversation each program is to process, and what each program is to do when it has finished processing its step of the conversation.

When a person at a terminal enters a transaction code that has been defined as conversational, IMS schedules the conversational program (for example, Program A) associated with that transaction code. When Program A issues its first call to the message queue, IMS returns the SPA that is defined for that transaction code to Program A's I/O area. The person at the terminal must enter the transaction code (and password, if one exists) only on the first input screen; the transaction code need not be entered during each step of the conversation. IMS treats data in subsequent screens as a continuation of the conversation started on the first screen.

After the program has retrieved the SPA, Program A can retrieve the input message from the terminal. After it has processed the message, Program A can either continue the conversation, or end it.

To continue the conversation, Program A can do any of the following:

- 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*.

  Definition:A deferred program switch means that Program A responds to the terminal and then passes control to another conversational program, Program B. After passing control to Program B, Program A is no longer part of the conversation. The next input message that the person at the terminal enters goes to Program B, although the person at the terminal is unaware that this message is being sent to a second program.

  Restriction: A deferred program switch is disallowed if the application is involved in an inbound protected conversation. The application will receive an X6 status code if it attempts to perform a deferred program switch in this environment.
- Pass control of the conversation to another conversational program without first responding to the originating terminal. This is called an *immediate program switch*.

  Definition: An immediate program switch lets you pass control directly to another conversational program without having to respond to the originating terminal. When you do this, the program that you pass the conversation to must respond 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.

  Restriction: An immediate program switch is disallowed if the application is involved in an inbound protected conversation. The application will be abended with a U711 if it attempts to perform an immediate program switch in this environment.

To end the conversation, Program A can do either of the following:

- Move a blank to the first byte of the transaction code area of the SPA and then return the SPA to IMS.
- Respond to the terminal and pass control to a nonconversational program. This is also called a deferred program 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.

Important Points 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 gives the SPA for the transaction to program B when program B issues its first message call.

The SPA is kept with the message. When the truncated data option is on, the size of the retained SPA is the largest SPA of any transaction in the conversation.

Example: If the conversation starts with TRANA (SPA=100), and the program switches to a TRANB (SPA=50), the input message for TRANB will contain a SPA segment of 100 bytes. IMS adjusts the size of the SPA so that TRANB receives only the first 50 bytes.

However, the IMS support that adjusts the size of the SPA does not exist in either IMS Version 5 or earlier systems. If TRANB is to execute on a remote MSC system without this support, it will be passed a SPA of 100 bytes when it is only expecting 50 bytes. There are two ways to prevent this larger sized SPA from being sent to an IMS Version 5 or earlier system:

1. The IMS installation could define TRANB on the local IMS system to have the RTRUNC parameter on its TRANSACT macro, thus forcing the SPA to a size of 50 bytes when it is inserted by TRANA.
2. For an installation that never uses truncated data nor wants to change the TRANSACT macros for remote transactions to specify RTRUNC, a new specification is available to set the system-wide default for the truncated data option. The new specification is TRUNC=Y|N in the DFSDCxxx proclib member. The installation could set the system default to not save truncated data, and the SPA would be automatically truncated to a size of 50 bytes when it is inserted by TRANA.

Related Reading: For more information on how to structure a conversational program, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

Recovery Considerations in Conversations

Because a conversation involves several steps and can involve several application programs, consider the following items:

- One way you can make recovery easier is to design the conversation so that all the database updates are done in the last step of the conversation. This way, if the conversation terminates abnormally, IMS can back out all the updates because they were all made during the same step of the conversation. Updating the database during the last step of the conversation is also a good idea, because the input from each step of the conversation is available.
- Although a conversation can terminate abnormally during any step of the conversation, IMS backs out only the database updates and output messages resulting during the last step of the conversation. IMS does not back out database updates or cancel output messages for previous steps, even though some of that processing might be inaccurate as a result of the abnormal termination.
- Certain IMS system service calls can be helpful if the program determines that some of its processing was invalid. These calls include ROLB, SETS, SETU, and ROLS. The Roll Back call (ROLB) backs out all of the changes that the program has made to the database. ROLB also cancels the output messages that the program has created (except those sent with an express PCB, as explained below) since the program's last commit point.

  The SETS, or SETU, and ROLS (with a *token*) calls work together to allow the application program to set intermediate backout points within the call processing of the program. The application program can set up to nine intermediate backout points. Your program needs to use the SETS or SETU call to specify a token for each point. A subsequent ROLS call, using the same token, can back out all database changes and discard all nonexpress messages processed since that SETS or SETU call.

  Definition:A token is a 4-byte identifier.

- 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 application program inserts messages using an express PCB, IMS waits until it has the complete message, rather than for the occurrence of a commit point, to transmit the message to its destination. (In this context, "insert" refers to a situation in which the application program sends the message and it is received by IMS; "transmit" refers to a situation in which IMS begins sending the message to its destination.) Therefore, when IMS has the complete message, it will be transmitted even if the program abnormally terminates. Messages sent with an express PCB are sent to their final destinations even if the program terminates abnormally or issues a ROLB call. For more information about the express PCB, refer to "To Other Programs and Terminals".
- To verify the accuracy of the previous processing, and to correct the processing that is determined to be inaccurate, you can use the Conversational Abnormal termination routine, DFSCONE0.

  Related Reading: For more information on DFSCONE0, see *IMS/ESA Customization Guide*.
- 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 database calls, and use a user-written or system-provided exit routine to schedule it.

Identifying Output Message Destinations

An application program can send messages to another application program or to IMS terminals. To send output messages, the program issues a call and references the I/O PCB or an alternate PCB. The I/O PCB and alternate PCBs represent logical terminals and other application programs with which the application program communicates.

Definition: An *alternate PCB* is a data communication program communication block (DCPCB) that you define to describe output message destinations other than the terminal that originated the input message.

The Originating Terminal

To send a message to the logical terminal that sent the input message, the program uses an I/O PCB. IMS puts the name of the logical terminal that sent the message in the I/O PCB when the program receives the message. As a result, the program need not do anything to the I/O PCB before sending the message. If a program receives a message from a batch-oriented BMP or CPI Communications driven program, no logical terminal name is available to put into the I/O PCB. In these cases, the logical terminal name field contains blanks.

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 running, so you can send output messages to several alternate destinations.

The application program might need to respond to the originating terminal before the person at the originating terminal can send any more messages. This might occur when a terminal is in **response mode** or in **conversational mode**:

- **Response mode** can apply to a communication line, a terminal, or a transaction. When response mode is in effect, IMS does not 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.

  If a response-mode transaction is processed, including Fast Path transactions, and the application does not insert a response back to the terminal through either the I/O PCB or alternate I/O PCB, but inserts a message to an alternate PCB (program-to-program switch), the second or subsequent application program must respond to the originating terminal and satisfy the response. IMS will not take the terminal out of response mode.

  If an application program terminates normally and does not issue an `ISRT` call to the I/O PCB, alternate I/O PCB, or alternate PCB, IMS sends system message DFS2082I to the originating terminal to satisfy the response for all response-mode transactions, including Fast Path transactions.

  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 is been defined as response mode. You specify response mode for communication lines and terminals on the TYPE and TERMINAL macros, respectively, at IMS system definition. You can define any transaction as a response-mode transaction; you do this on the TRANSACT macro at IMS 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 must respond to the originating terminal. But sometimes the originating terminal is a physical terminal that is made up of two components—for example, a printer and a display. If the physical terminal is made up of two components, each component has a different logical terminal name. To send an output message to the printer part of the terminal, the program must use a different logical terminal name than the one associated with the input message; it must send the output message to an alternate destination. A special kind of alternate PCB is available to programs in these situations; it is called an *alternate response PCB*.

Definition: An alternate response PCB lets you send messages when exclusive, response, or conversational mode is in effect. See the next section for more information.

Alternate Response PCB

The destination of an alternate response PCB must be a logical terminal—you cannot use an alternate response PCB to represent another application program. When you use an alternate response PCB during response mode or conversational mode, the logical terminal represented by the alternate response PCB must represent the same physical terminal as the originating logical terminal.

In these processing modes, after receiving the message, the application program must respond by issuing an `ISRT` call to one of the following:

- The I/O PCB.
- An alternate response PCB.
- An alternate PCB whose destination is another application program, that is, a program-to-program switch.
- An alternate PCB whose destination is an ISC link. This is allowed only for front-end switch messages.

  Related Reading: For more information on front-end switch messages, see *IMS/ESA Customization Guide*.

If one of these criteria is not met, message DFS2082I is sent to the terminal.

Express PCB

Consider specifying an alternate PCB as an *express PCB*. The express designation relates to whether a message that the application program inserted is actually transmitted to the destination if the program abnormally terminates or issues a `ROLL`, `ROLB`, or `ROLS` call. For all PCBs, when a program abnormally terminates or issues a `ROLL`, `ROLB`, or `ROLS` call, messages that were inserted but not made available for transmission are canceled while messages that were made available for transmission are never canceled.

Definition: An express PCB is an alternate response PCB that allows your program to transmit the message to the destination terminal earlier than when you use a nonexpress PCB.

For a nonexpress PCB, the message is not made available for transmission to its destination until the program reaches a commit point. The commit point occurs when the program terminates, issues a `CHKP` call, or requests the next input message and when the transaction has been defined with MODE=SNGL.

For an express PCB, when IMS has the complete message, it makes the message available for transmission to the destination. In addition to occurring at a commit point, it also occurs when the application program issues a `PURG` call using that PCB or when it requests the next input message.

You should provide the answers to the questions below to the data communications administrator to help in meeting your application’s message processing 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 must send output messages?
- Should the program be able to send an output message before it terminates abnormally?

Chapter 7. Application Design for APPC

Advanced Program-to-Program Communication (APPC) is IBM's preferred protocol for program-to-program communication. Application programs can be distributed throughout the network and communicate with each other in many hardware architectures and software environments.

This chapter describes the APPC function of IMS TM.

In this Chapter:

Related Reading: For more information on APPC, see:

- *IMS/ESA Application Programming: Transaction Manager*, which includes specific information on APPC such as the application programming interface (API) and descriptions of the `APSB` and `DPSB` calls.
- *IMS/ESA Administration Guide: Transaction Manager*, which includes an overview of APPC for LU 6.2 devices and CPI Communications concepts.

Overview of APPC and LU 6.2

APPC allows application programs using APPC protocols to enter IMS transactions from LU 6.2 devices. The LU 6.2 application program runs on an LU 6.2 device supporting APPC.

APPC creates an environment that allows:

- Remote LU 6.2 devices to enter IMS local and remote transactions
- IMS application programs to insert transaction output to LU 6.2 devices with no coding changes to existing application programs
- New application programs to make full use of LU 6.2 device facilities
- Data integrity provided by IMS and in LU 6.2 environments that do not have a distributed sync-point function

Application Program Types

APPC/IMS is part of IMS TM that uses the CPI communications interface to communicate with application programs. APPC/IMS supports the following types of application programs for LU 6.2 processing:

- Standard DL/I
- Modified standard DL/I
- CPI Communications driven

Standard DL/I Application Program

A standard DL/I application program does not issue any CPI Communications calls or establish any CPI-C conversations. This application program can communicate with LU 6.2 products that replace other LU-type terminals using the IMS API. A standard DL/I application program does not need to be modified, recompiled, or link-edited, and it executes as it currently does.

Modified Standard DL/I Application Program

A modified standard DL/I application program is a standard DL/I online IMS TM application program that uses both DL/I calls and CPI Communications calls. It can be an MPP, BMP, or IFP that can access full-function databases, DEDBs, MSDBs, and DB2 databases.

A modified standard DL/I application program uses CPI Communications (CPI-C) calls to provide support for an LU 6.2 and non-LU 6.2 mixed network. The same application program can be a standard DL/I on one execution, when the CPI Communications `ALLOCATE` verb is not issued, and a modified standard DL/I on a different execution when the CPI Communications `ALLOCATE` verb is issued.

A modified standard DL/I application program receives its messages using DL/I `GU` calls to the I/O PCB and issues output responses using DL/I `ISRT` calls. CPI Communications calls can also be used to allocate new conversations and to send and receive data for them.

Related Reading: For a list of the CPI Communications calls, see *Common Programming Interface Communications Reference*.

Use a modified standard DL/I application program when you want to use an existing standard DL/I application program to establish a conversation with another LU 6.2 device or the same network destination. The standard DL/I application program is optionally modified and uses new functions, new application and transaction definitions, and modified DL/I calls to initiate LU 6.2 application programs. Program calls and parameters are available to use the IMS-provided implicit API and the CPI Communications explicit API.

CPI Communications Driven Program

A CPI Communications driven application program uses `Commit` and `Backout` calls, and CPI Communications interface calls or LU 6.2 verbs for input and output message processing. This application program uses the CPI Communications explicit API, and can access full-function databases, DEDBs, MSDBs, and DB2 databases. An LU 6.2 device can activate a CPI Communications driven application program only by allocating a conversation.

Unlike a standard DL/I or modified standard DL/I application program, input and output message processing for a CPI Communications driven program uses APPC/MVS buffers and bypasses IMS message queueing. Because these application programs do not use the IMS message queue, they can control their own execution with the partner LU 6.2 system. An IMS `APSB` call enables you to allocate a PSB for accessing IMS databases and alternate PCBs.

The application program uses the Common Programming Interface Resource Recovery (CPI-RR) `SRRCMIT` verb to initiate an IMS sync point and the CPI-RR `SRRBACK`verb for backout. CPI Communications driven application programs use the CPI-RR calls to initiate IMS sync point processing prior to program termination.

A CPI Communications driven application program is able to:

- Access any type of database
- Receive and send large messages like the standard DL/I and modified standard DL/I application programs
- Control the flow of input and output with CPI Communications calls
- Allocate multiple conversations with partner LU 6.2 devices
- Cause synchronization with conversation partners
- Use the IMS implicit API (for example, IMS queue services)
- Use IMS services (for example, sync point at program termination) regardless of the API that is used

Application Objectives

Each application type has a different purpose, and its ease-of-use varies depending on whether the program is a standard DL/I, modified standard DL/I, or a CPI Communications driven application program. Table 11 on page 105 lists the purpose and ease-of-use for each application type. This information must be balanced with IMS resource use.

Table 11. Using Application Programs in APPC

	Ease of Use		
Purpose of Application Program	**Standard DL/I Program**	**Modified Standard DL/I Program**	**CPI-C Driven Program**
Inquiry	Easy	Neutral	Very Difficult
Data Entry	Easy	Easy	Difficult
Bulk Transfer	Easy	Easy	Neutral
Cooperative	Difficult	Difficult	Desirable
Distributed	Difficult	Neutral	Desirable
High Integrity	Neutral	Neutral	Desirable
Client Server	Easy	Neutral	Very Difficult

Choosing Conversation Attributes

The LU 6.2 transaction program indicates how the transaction is to be processed by IMS. Two processing modes are available: **synchronous** and **asynchronous**.

Synchronous Conversation

A conversation is synchronous if the partner waits for the response on the same conversation used to send the input data.

Synchronous processing is requested by issuing the `RECEIVE_AND_WAIT` verb after the `SEND_DATA` verb. Use this mode for IMS response-mode transactions and IMS conversational-mode transactions.

Example:

```
MC_ALLOCATE TPN(MYTXN)
MC_SEND_DATA  'THIS CAN BE A RESPONSE MODE'
MC_SEND_DATA  'OR CONVERSATIONAL MODE'
MC_SEND_DATA  'IMS TRANSACTION'
MC_RECEIVE_AND_WAIT
```

For examples of transaction flow, see “LU 6.2 Flow Diagrams” on page 109.

Asynchronous Conversation

A conversation is asynchronous if the partner program normally deallocates a conversation after sending the input data. Output is sent to the TP name of DFSASYNC.

Asynchronous processing is requested by issuing the `DEALLOCATE` verb after the `SEND_DATA` verb. Use asynchronous processing for IMS commands, message switches, and non-response, non-conversational transactions.

Example:

```
MC_ALLOCATE TPN(OTHERTXN)
MC_SEND_DATA  'THIS MUST BE A MESSAGE SWITCH, IMS COMMAND'
MC_SEND_DATA  'OR A NON-RESP NON-CONV TRANSACTION'
MC_DEALLOCATE
```

For examples of transaction flow, see “LU 6.2 Flow Diagrams” on page 109.

Asynchronous Output Delivery

Asynchronous output is held on the IMS message queue for delivery. When the output is enqueued, IMS attempts to allocate a conversation to send this output. If this fails, IMS holds the output for later delivery. This delivery can be initiated by an operator command (/ALLOC), or by the enqueue of a new message for this LU 6.2 destination.

MSC Synchronous and Asynchronous Conversation

MSC remote application messages from both synchronous and asynchronous APPC conversations can be queued on the multiple systems coupling (MSC) link. These messages can then be sent across the MSC link to a remote IMS for processing.

For examples of transaction flow, see “LU 6.2 Flow Diagrams” on page 109.

Conversation Type

The APPC conversation type defines how data is passed on and retrieved from APPC verbs. It is similar in concept to file blocking and affects both ends of the conversation.

APPC supports two types of conversations:

Basic conversation

This low-conversation allows programs to exchange data in a standardized format. This format is a stream of data containing 2-byte length fields (referred to as LLs) that specify the amount of data to follow before the next length field. The typical data pattern is:

```
LL, data, LL, data
```

Each grouping of `LL, data` is referred to as a logical record. A basic conversation is used to send multiple segments with one verb and to receive maximum data with one verb.

Mapped conversation

This high-conversation allows programs to exchange arbitrary data records in data formats approved by application programmers. One send verb results in one receive verb, and MVS and VTAM handle the buffering.

Related Reading: For more information on basic and mapped conversations, see *SNA LU 6.2 Reference: Peer Protocols* and *SNA Transaction Programmer's Reference Manual for LU Type 6.2*.

Conversation State

CPI Communications uses conversation state to determine what the next set of actions will be. Examples of conversation states are:

RESET	The initial state before communications begin.
SEND	The program can send or optionally receive.
RECEIVE	The program must receive or abort.
CONFIRM	The program must respond to a partner.

The basic rules for APPC verbs are:
- The program that initiates the conversation speaks first.
- Only one APPC verb can be outstanding at time.
- Programs take turns sending and receiving.
- The state of the conversation determines the verbs a program can issue.

Synchronization Level

The APPC synchronization level defines the protocol that is used when changing conversation states. APPC and IMS support the following `sync_level` values:

NONE	Specifies that the programs do not issue calls or recognize returned parameters relating to synchronization.
CONFIRM	Specifies that the programs can perform confirmation processing on the conversation.
SYNCPT	Specifies that the programs participate in coordinated commit processing on resources that are updated during the conversation under the RRS/MVS recovery platform. A conversation with this level is also called a protected conversation.

Allocating a conversation with SYNCLVL=SYNCPT requires the Resource Recovery Services (RRS/MVS) as the sync-point manager (SPM). RRS/MVS controls the commitment of protected resources by coordinating the commit or backout request with the participating owners of the updated resources, the resource managers. IMS is the resource manager for DL/I, Fast Path data, and the IMS message queues. The application program decides whether the data is to be committed or aborted and communicates this decision to the SPM. The SPM then coordinates the actions in support of this decision among the resource managers.

Related Reading: For more information on SYNCLVL=SYNCPT, see APPC in *IMS/ESA Administration Guide: Transaction Manager*.

Application Programming Interface for LU Type 6.2

IMS application programs can use the IMS implicit LU 6.2 API to access LU 6.2 devices. This API provides compatibility with non-LU 6.2 device types so that the same application program can be used from both LU 6.2 and non-LU 6.2 devices. The API adds to the APPC interface by supplying IMS-provided processing for the application program. You can use the explicit CPI Communications interface for APPC functions and facilities for new or rewritten IMS application programs.

Implicit API

The implicit API accesses an APPC conversation indirectly. This API uses the standard DL/I calls (GU, ISRT, PURG) to send and receive data. It allows application programs that are not specific to LU 6.2 protocols to use LU 6.2 devices. The API uses new and changed DL/I calls (CHNG, INQY, SETO) to utilize LU 6.2. Using the existing IMS application programming base, you can write LU 6.2-specific applications using this API and not using the CPI Communications calls. Although the implicit API uses only some of the LU 6.2 capabilities, it can be a useful simplification for many applications. The implicit API also provides function outside of LU 6.2, like message queueing and automatic asynchronous message delivery.

IMS generates all CPI Communications calls under the implicit API. The application interaction is strictly with the IMS message queue.

The remote LU 6.2 system must be able to handle the LU 6.2 flows. APPC/MVS generates these flows from the CPI Communications calls issued by the IMS application program using the implicit API. An IMS application program can use the explicit API to issue the CPI Communications directly. This is useful with remote LU 6.2 systems that have incomplete LU 6.2 implementations, or that are incompatible with the IMS implicit API support. See the LU 6.2 data flow examples under "LU 6.2 Partner Program Design" on page 109.

The existing API is extended so that:

- Asynchronous LU 6.2 output is created by using alternate PCBs that reference LU 6.2 destinations. The DL/I CHNG call can supply parameters to specify an LU 6.2 destination. Default values are used for omitted parameters.
- An application program can retrieve the current conversation attributes such as the conversation type (basic or mapped), the sync_level (NONE, CONFIRM, or SYNCPT), and asynchronous or synchronous conversation.
- A terminal message switch can be used to and from LU 6.2 devices. See "LU 6.2 Partner Program Design" on page 109 for a description of the message switch.

Explicit API

The explicit API (the CPI Communications API) can be used by any IMS application program to access an APPC conversation directly. IMS resources are available to the CPI Communications driven application program only if the application issues the APSB (Allocate_ PSB) call. The CPI Communications driven application program must use the CPI-RR SRRCMIT and SRRBACK verbs to initiate an IMS sync point or backout, or if SYNCLVL=SYNCPT is specified, to communicate the sync point decision to the RRS/MVS sync point manager.

Related Reading: For a description of the SRRCMIT and SRRBACK verbs, see *SAA CPI Resource Recovery Reference*.

LU 6.2 Partner Program Design

The flow of a transaction that is sent from an LU 6.2 device differs, depending on the conversation attributes and synchronization levels. Different results occur, and the partner system takes actions accordingly. The flow diagrams and the integrity tables in this section present these differences.

LU 6.2 Flow Diagrams

Figure 30 through Figure 38 on page 113 show the flow between a synchronous or asynchronous LU 6.2 application program and an IMS application program in a single (local) IMS system.

Figure 39 on page 114 through Figure 42 on page 117 show the flow between a synchronous or asynchronous LU 6.2 application program in a single (local) IMS system and an IMS application program in a remote IMS system across a multiple systems coupling (MSC) link.

Figure 43 on page 118 and Figure 44 on page 119 show commit scenarios with SYNC_LEVEL=SYNCPT. Figure 45 on page 120 shows a backout scenario with SYNC_LEVEL=SYNCPT.

Differences in buffering and encapsulation of control data with user data may cause variations in the flows. The control data are the 3 returned fields from the Receive APPC verb: Status_received, Data_received, and Request_to_send_received. Any variations based on these differences will not affect the function or use of the flows.

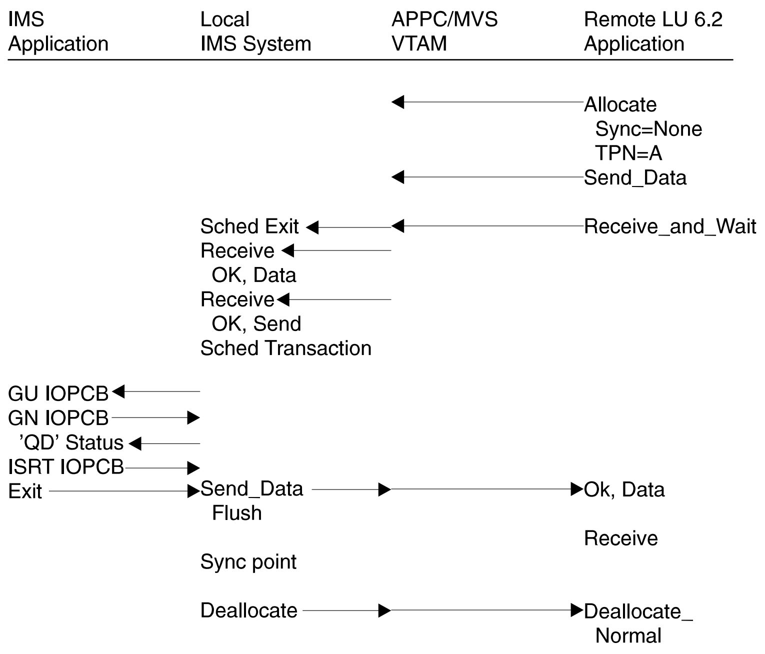


Figure 30. Flow of a Local IMS Synchronous Transaction with Sync_level=None

Figure 31 on page 110 shows the flow of a local synchronous transaction whose Sync_level is Confirm.

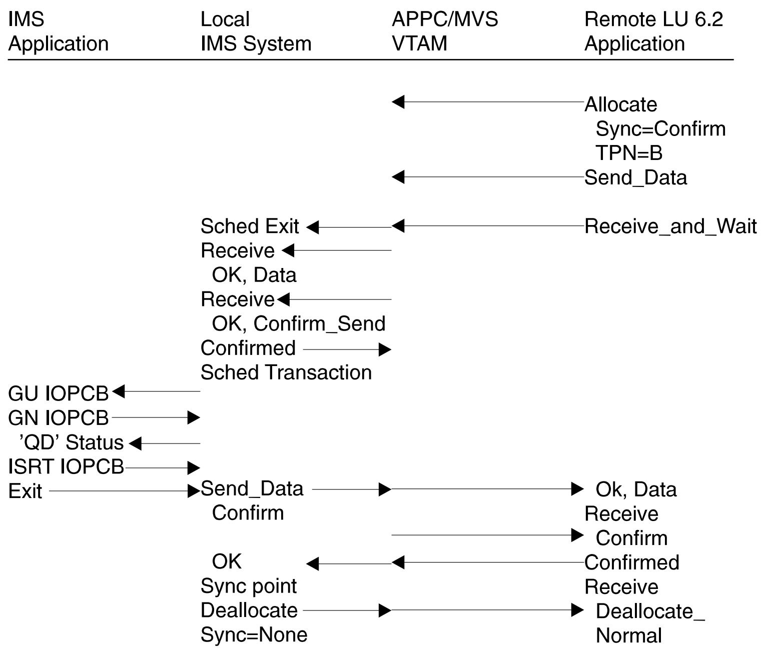


Figure 31. Flow of a Local IMS Synchronous Transaction with Sync_level=Confirm

Figure 32 shows the flow of a local asynchronous transaction whose Sync_level is None.

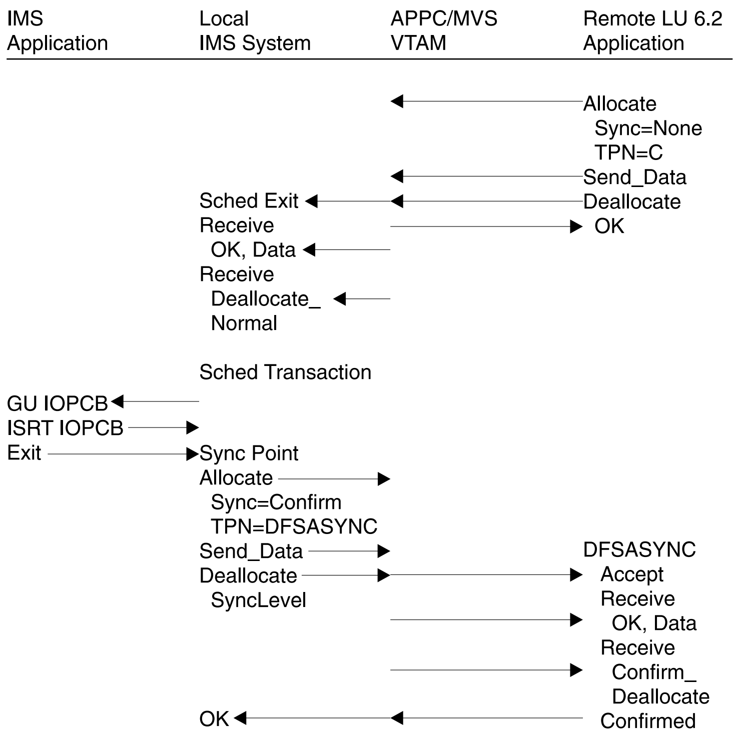


Figure 32. Flow of a Local IMS Asynchronous Transaction with Sync_level=None

Figure 33 on page 111 shows the flow of a local asynchronous transaction whose Sync_level is Confirm.

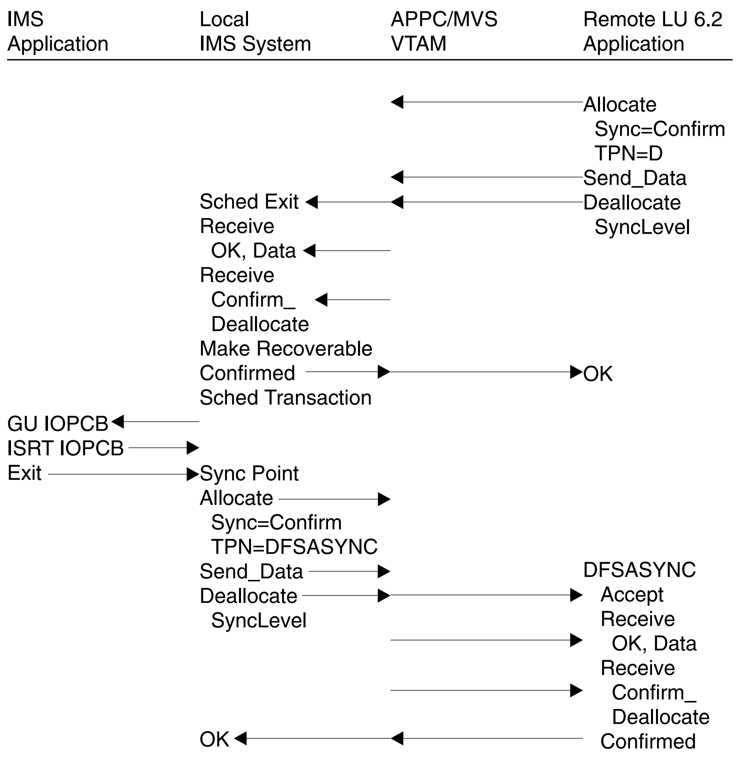


Figure 33. Flow of a Local IMS Asynchronous Transaction with Sync_level=Confirm

Figure 34 shows the flow of a local conversational transaction whose Sync_level is None.

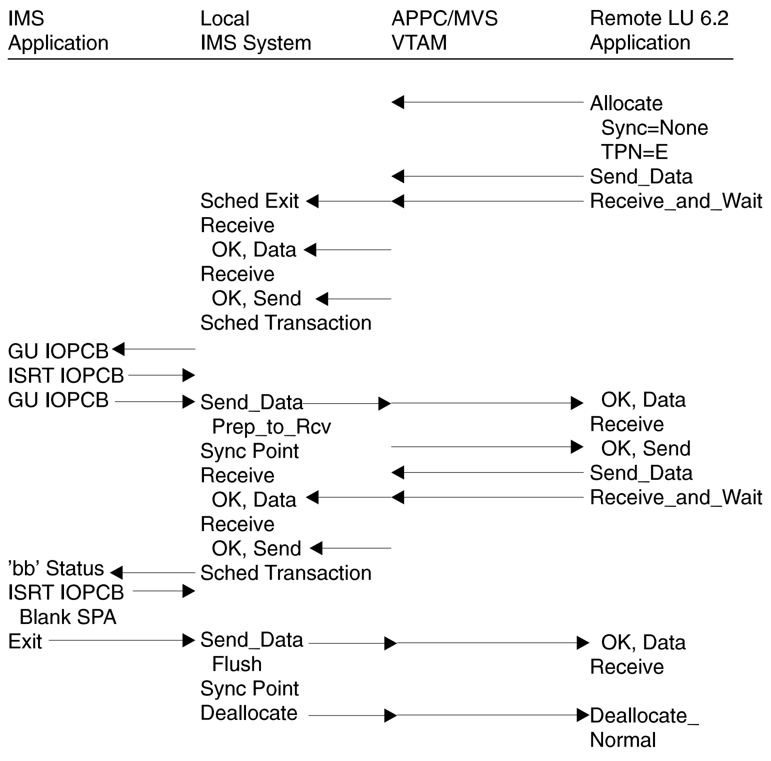


Figure 34. Flow of a Local IMS Conversational Transaction When Sync_level=None

LU 6.2 Partner Program Design

Figure 35 shows the flow of a local IMS command whose Sync_level is None.

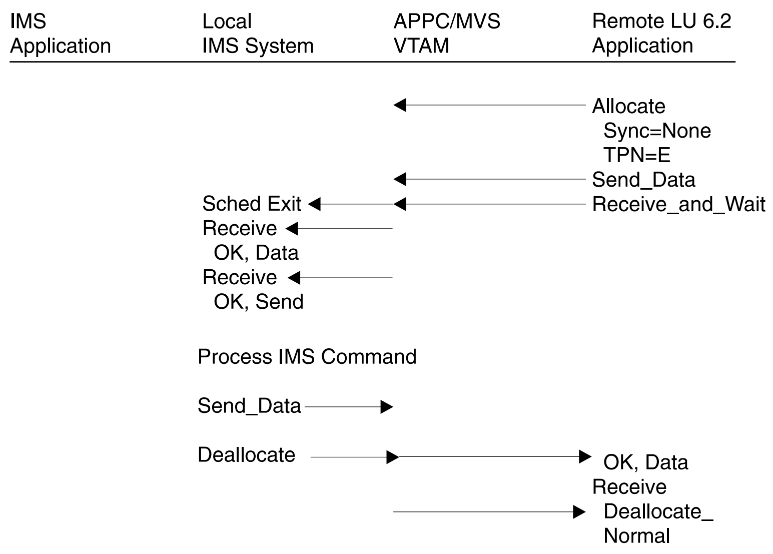


Figure 35. Flow of a Local IMS Command When Sync_level=None

Figure 36 shows the flow of a local asynchronous command whose Sync_level is Confirm.

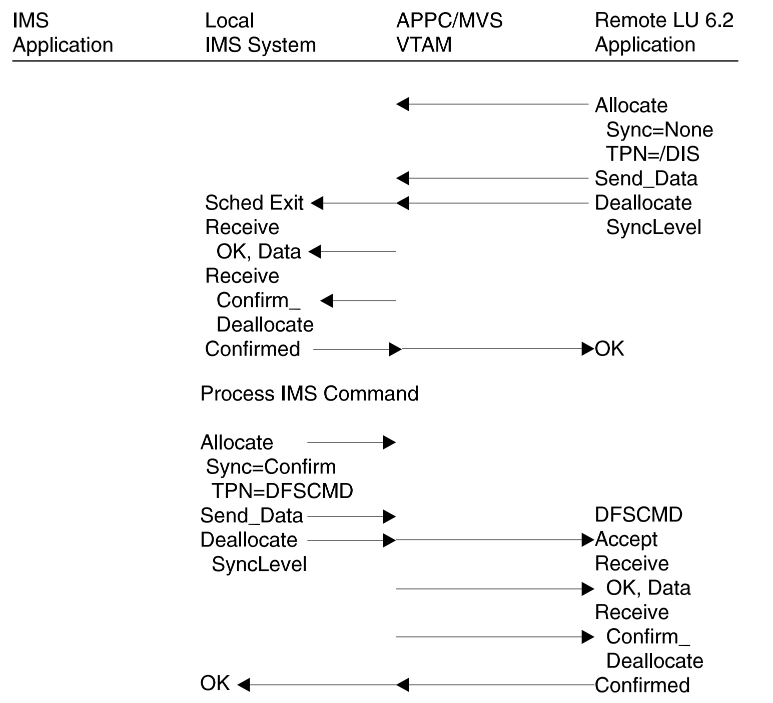


Figure 36. Flow of a Local IMS Asynchronous Command with Sync_level=Confirm

Figure 37 on page 113 shows the flow of a message switch whose Sync_level is None.

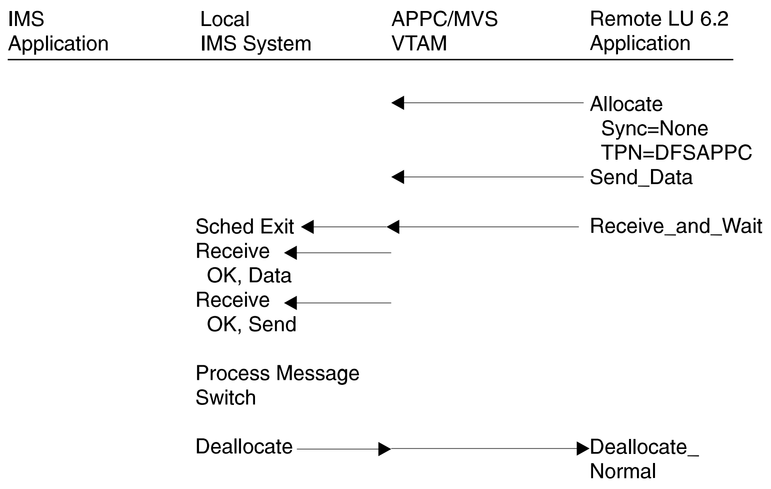


Figure 37. Flow of Message Switch with Sync_level=None

Synchronous is used to verify that no error has occurred while processing DFSAPPC. If an error occurred, the error message returns before DEALLOCATE.

Figure 38 shows the flow of a CPI-C driven program whose Sync_level is None.

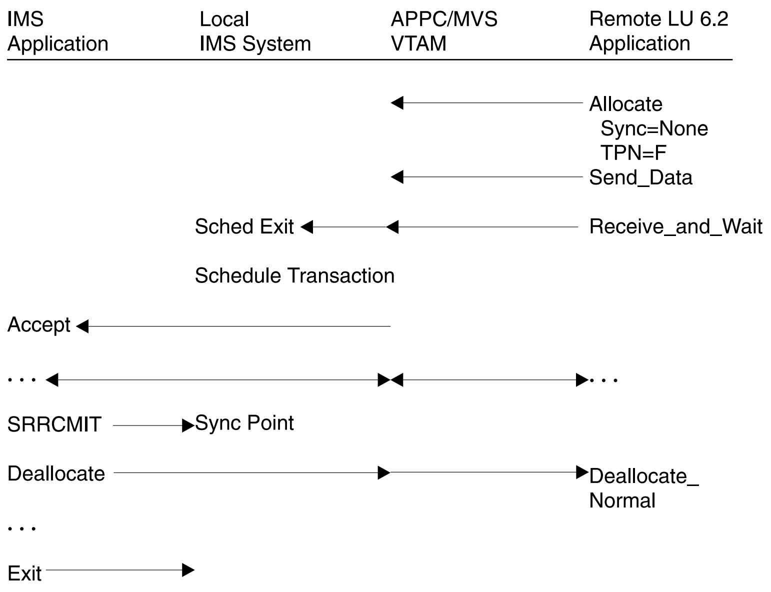


Figure 38. Flow of a Local CPI Communications Driven Program with Sync_level=None

Figure 39 on page 114 shows the flow of a remote synchronous transaction whose Sync_level is None.

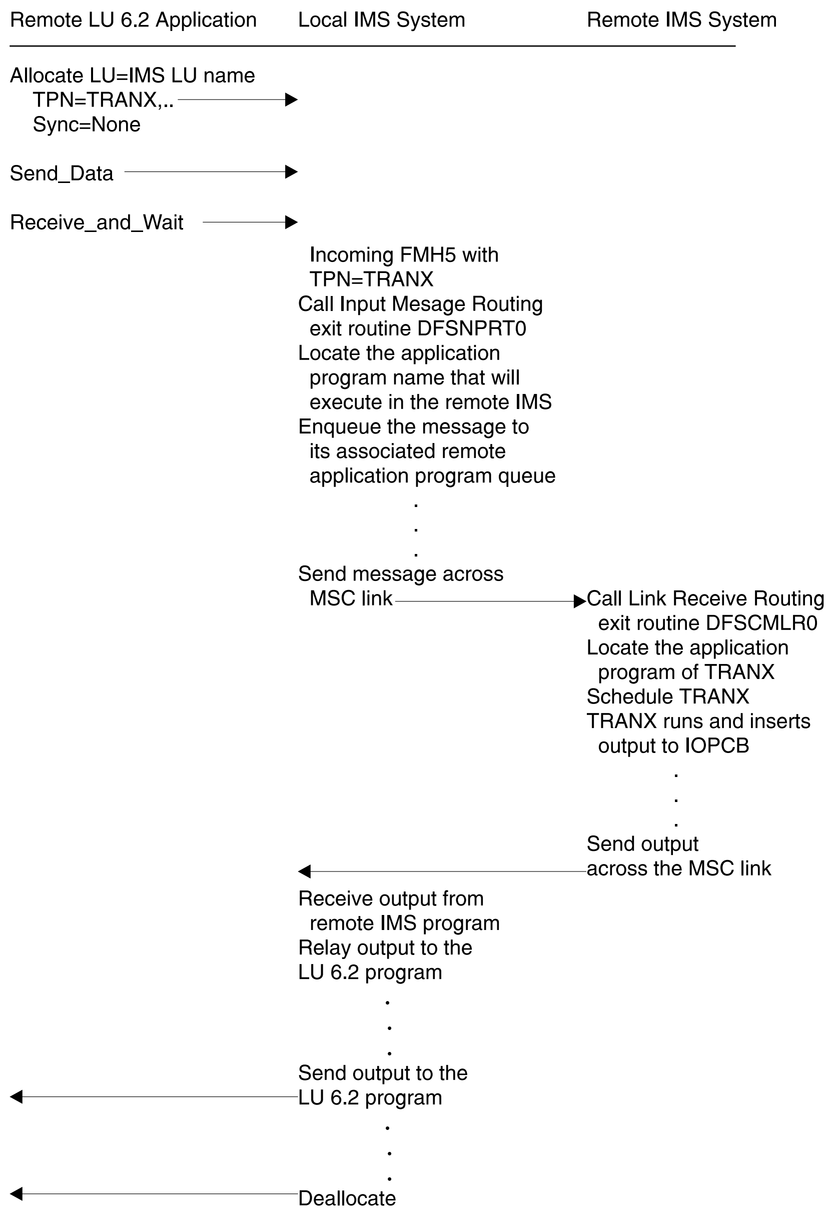


Figure 39. Flow of a Remote IMS Synchronous Transaction with Sync_level=None

Figure 40 on page 115 shows the flow of a remote asynchronous transaction whose Sync_level is None.

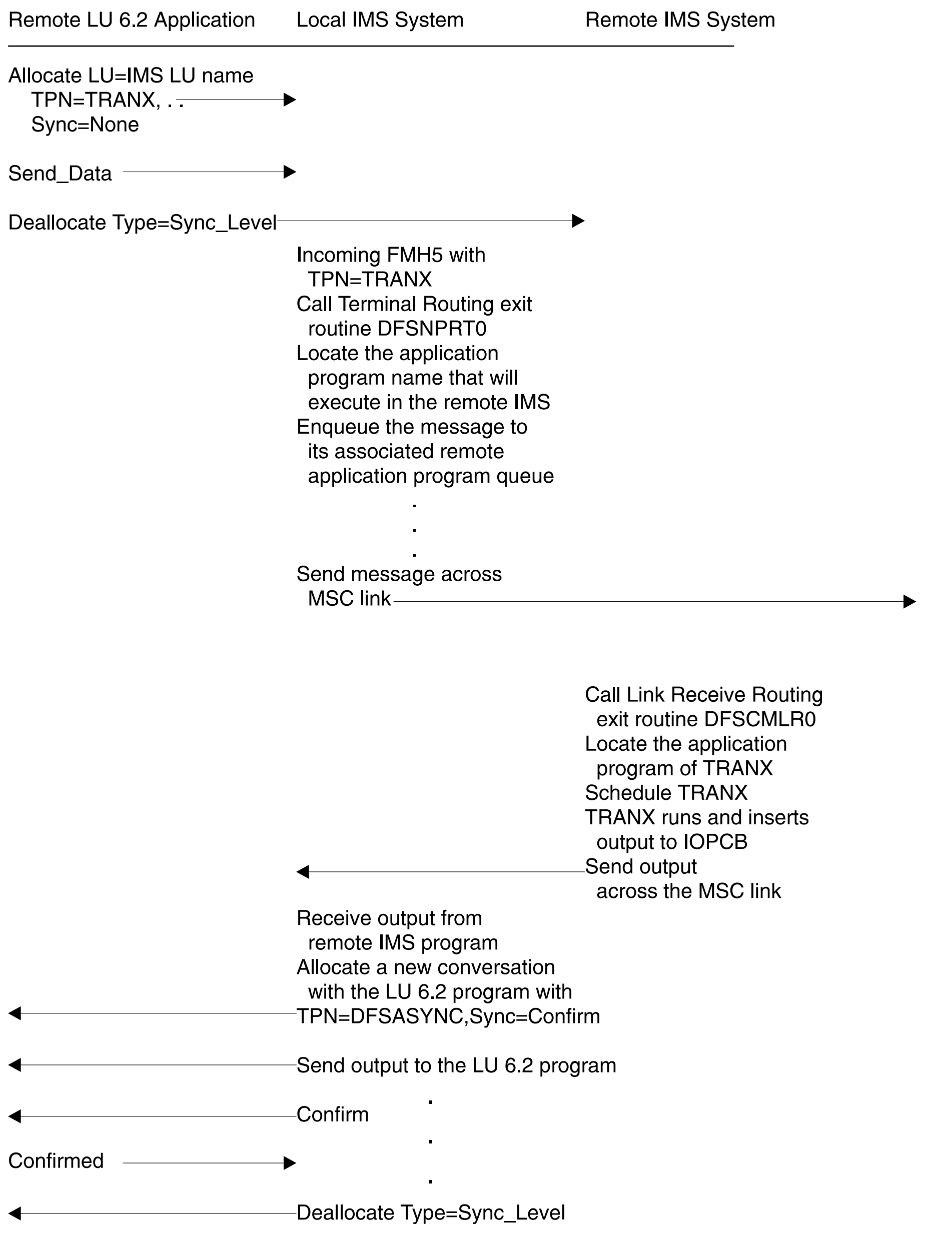


Figure 40. Flow of a Remote IMS Asynchronous Transaction with Sync_level=None

Figure 41 on page 116 shows the flow of a remote asynchronous transaction whose Sync_level is Confirm.

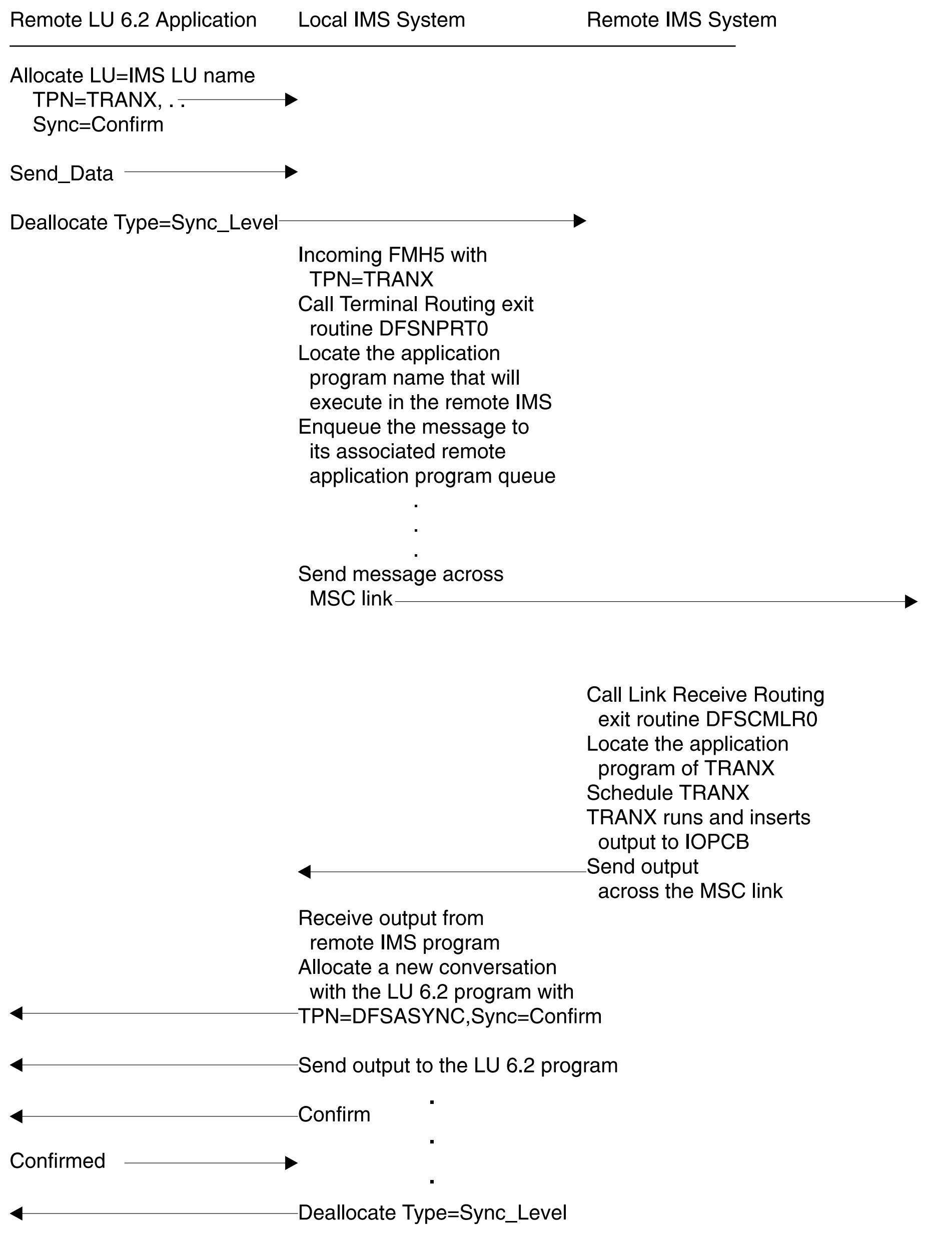

Remote LU 6.2 Application
Local IMS System
Remote IMS System
Allocate LU=IMS LU name
TPN=TRANX, . .
Sync=Confirm
Send_Data
Deallocate Type=Sync_Level
Incoming FMH5 with TPN=TRANX
Call Terminal Routing exit routine DFSNPRT0
Locate the application program name that will execute in the remote IMS
Enqueue the message to its associated remote application program queue
Send message across MSC link
Call Link Receive Routing exit routine DFSCMLR0
Locate the application program of TRANX
Schedule TRANX
TRANX runs and inserts output to IOPCB
Send output across the MSC link
Receive output from remote IMS program
Allocate a new conversation with the LU 6.2 program with TPN=DFSASYNC,Sync=Confirm
Send output to the LU 6.2 program
Confirm
Confirmed
Deallocate Type=Sync_Level

Figure 41. Flow of a Remote IMS Asynchronous Transaction with Sync_level=Confirm

Figure 42 on page 117 shows the flow of a remote synchronous transaction whose Sync_level is Confirm.

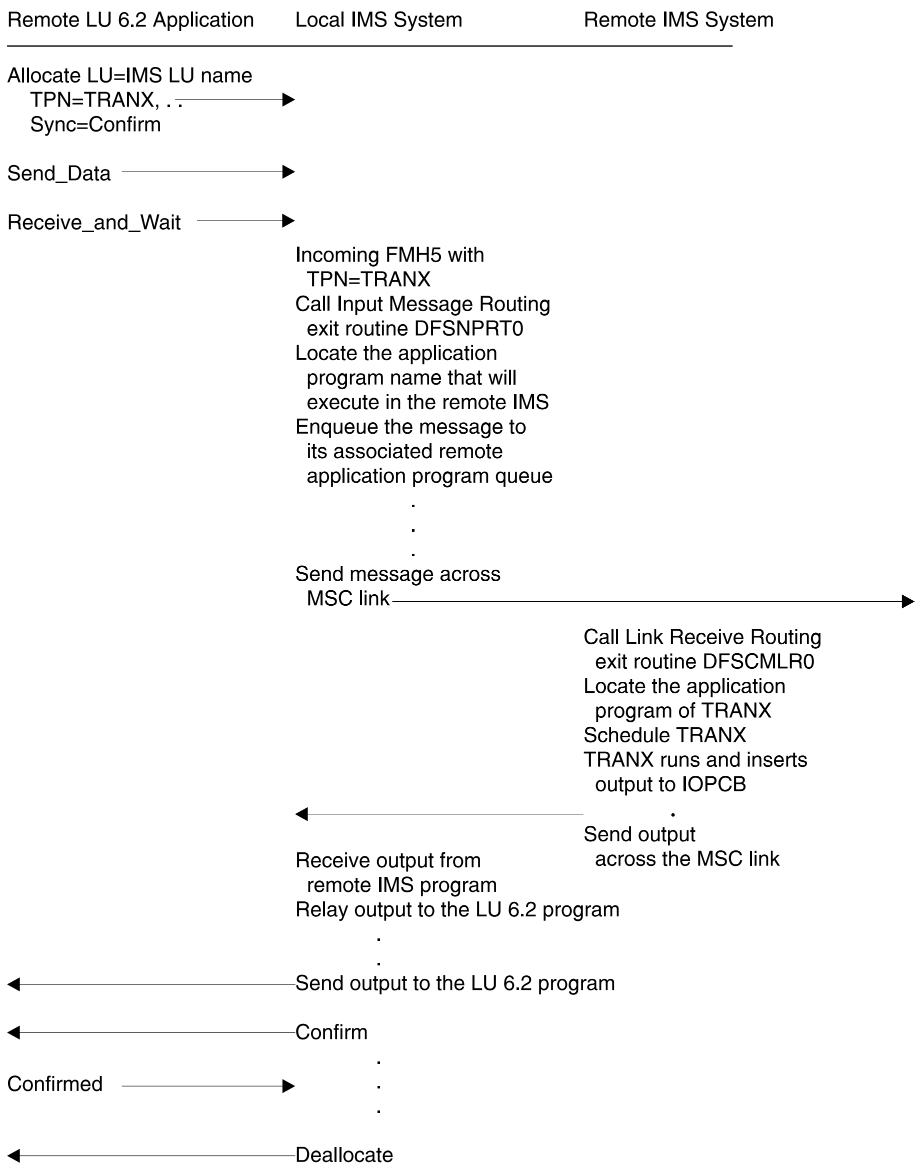

Remote LU 6.2 Application
Local IMS System
Remote IMS System
Allocate LU=IMS LU name TPN=TRANX, . . Sync=Confirm
Send_Data
Receive_and_Wait
Incoming FMH5 with TPN=TRANX
Call Input Message Routing exit routine DFSNPRT0
Locate the application program name that will execute in the remote IMS
Enqueue the message to its associated remote application program queue
Send message across MSC link
Call Link Receive Routing exit routine DFSCMLR0
Locate the application program of TRANX
Schedule TRANX
TRANX runs and inserts output to IOPCB
Send output across the MSC link
Receive output from remote IMS program
Relay output to the LU 6.2 program
Send output to the LU 6.2 program
Confirm
Confirmed
Deallocate

Figure 42. Flow of a Remote IMS Synchronous Transaction with Sync_Level=Confirm

The scenarios shown in the following figures provide examples of the two-phase process for the supported application program types. The LU 6.2 verbs are used to illustrate supported functions and interfaces between the components. Only parameters pertinent to the examples are included. This does not imply that other parameters are not supported.

Figure 43 on page 118 shows a standard DL/I program commit scenario whose Sync_Level is Syncpt.

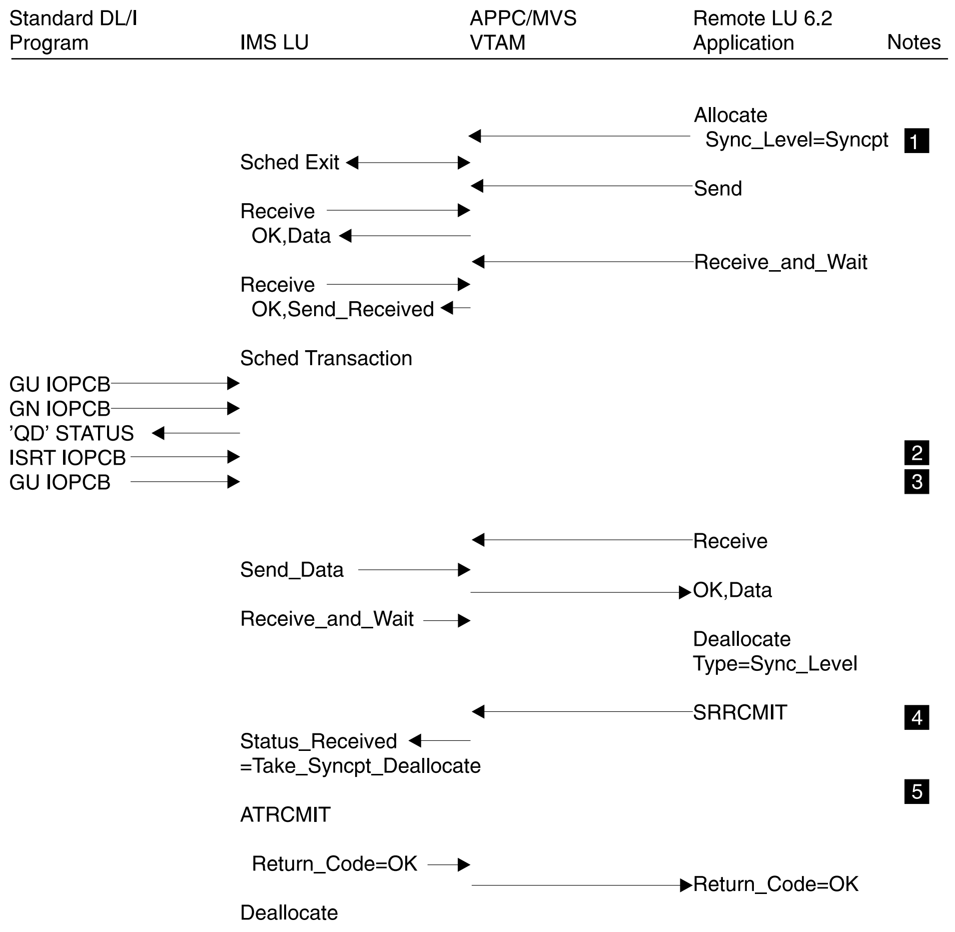


Figure 43. Standard DL/I Program Commit Scenario with Sync_Level=Syncpt

Notes:

1 `Sync_Level=Syncpt` triggers a protected resource update.

2 This application program inserts output for the remote application to the IMS message queue.

3 The `GU` initiates the transfer of the output.

4 The remote application sends a Confirmed after receiving data (output).

5 IMS issues `ATRCMIT` (equivalent to SRRCMIT) to start the two-phase process.

Figure 44 on page 119 shows a CPI-C driven commit scenario whose Sync_Level is Syncpt.

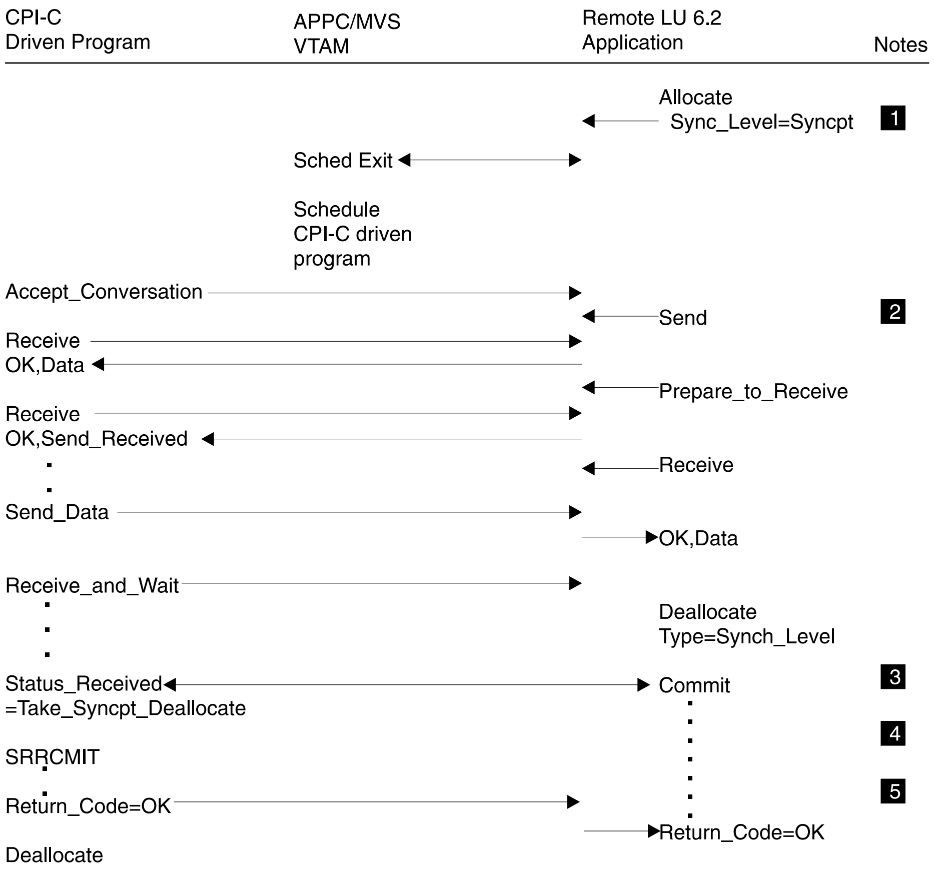


Figure 44. CPI-C Driven Commit Scenario with Sync_Level=Syncpt

Notes:

1 `Sync_Level=Syncpt` triggers a protected resource update.

2 The programs send and receive data.

3 The remote application decides to commit the updates.

4 The CPI-C program issues `SRRCMIT` to commit the changes.

5 The commit return code is returned to the remote application.

Figure 45 on page 120 shows a standard DL/I program backout scenario whose Sync_Level is Syncpt.

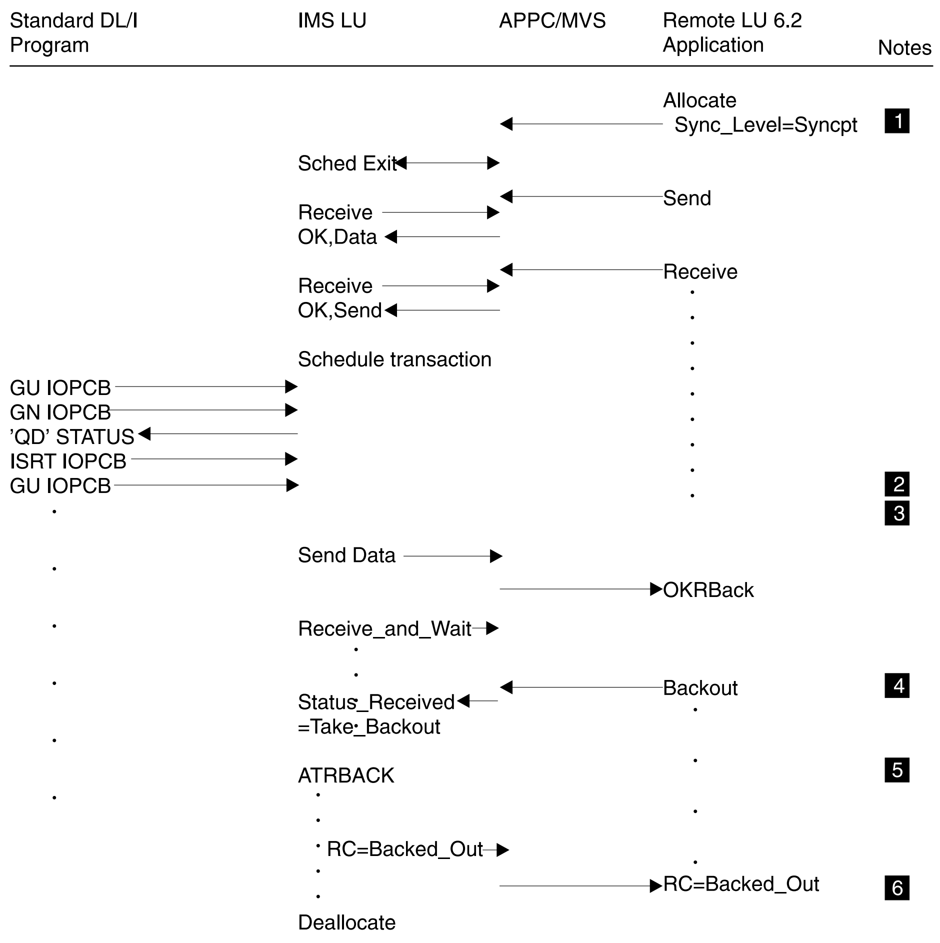


Figure 45. Standard DL/I Program Backout Scenario with Sync_Level=Syncpt

Notes:

1 `Sync_Level=Syncpt` triggers a protected-resource update.

2 This application program inserts output for the remote application to the IMS message queue.

3 The `GU` initiates the transfer of the output.

4 The remote application decides to back out any updates.

5 IMS abends the application with a U119 to back out the application.

6 The backout return code is returned to the remote application.

Figure 46 on page 121 shows a standard DL/I program backout scenario whose Sync_Level is Syncpt.

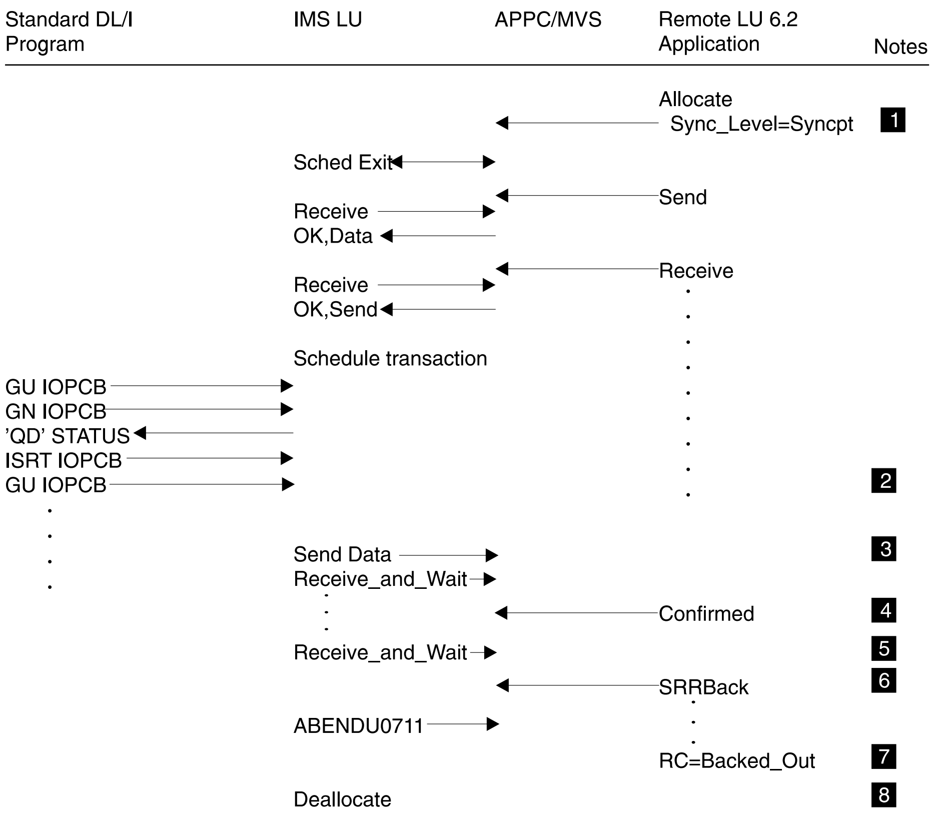


Figure 46. Standard DL/I Program Backout Scenario with Sync_Level=Syncpt

Notes:

1 `Sync_Level=Syncpt` triggers a protected-resource update.

2 This application program inserts output for the remote application to the IMS message queue.

3 The `GU` initiates the transfer of the output.

4 The remote application sends a Confirmed after receiving data (output).

5 IMS issues `ATBRCUW on behalf of the DL/I application to wait for a commit or backout.`

6 The remote application decides to back out any updates.

7 IMS abends the application with U0711 to back out the application.

8 The backout return code is returned to the remote application.

Figure 47 on page 122 shows a standard DL/I program ROLB scenario whose Sync_Level is Syncpt.

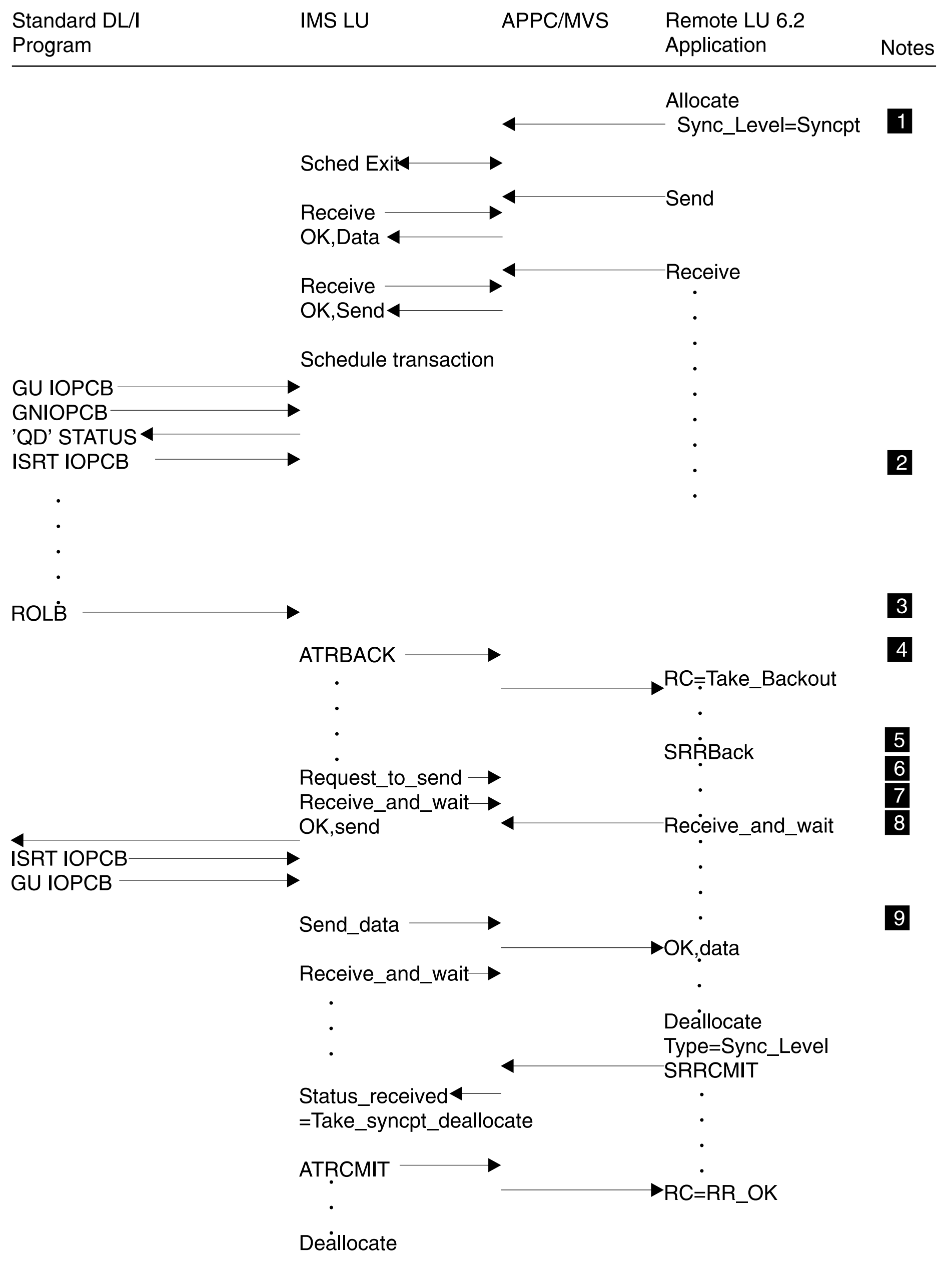


Figure 47. Standard DL/I Program ROLB Scenario with Sync_Level=Syncpt

Notes:

1 `Sync_Level=Syncpt` triggers a protected-resource update.

2 This application program inserts output for the remote application to the IMS message queue.

3 DL/I program issues a `ROLB`.

4 Standard DL/I program commit scenario. Refer to Figure 43 on page 118.

Figure 48 on page 123 shows multiple transactions in the same commit whose Sync_Level is Syncpt.

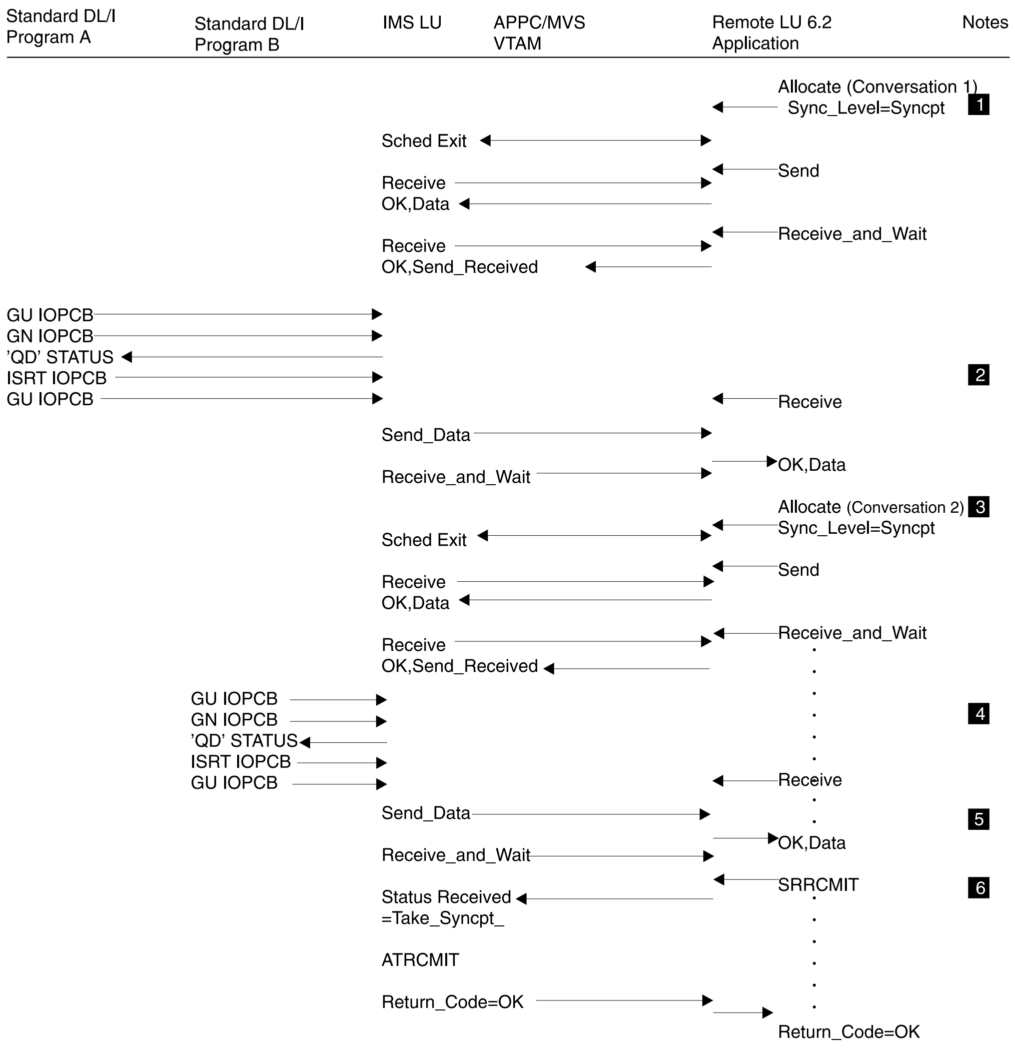


Figure 48. Multiple Transactions in Same Commit with Sync_Level=Syncpt

Notes:

1 An allocate with `Sync_Level=Syncpt` triggers a protected resource update with Conversation 1.

2 The first transaction provides the output for Conversation 1.

3 An allocate with `Sync_Level=Syncpt` triggers a protected resource update with Conversation 2.

4 The second transaction provides the output for Conversation 2.

5 The remote application issues SRRCMIT to commit both transactions.

6 IMS issues `ATRCMIT` to start the two-phase process on behalf of each DL/I application.

Integrity Tables

Table 12 on page 124 shows the results, from the viewpoint of the IMS partner system, of normal conversation completion, abnormal conversation completion due to a session failure, and abnormal conversation completion due to non-session failures. These results apply to asynchronous and synchronous conversations and both input and output. This figure also shows the outcome of the message, and the action that the partner system takes when it detects the failure. An example of an action, under “LU 6.2 Session Failure,” is a programmable work station (PWS) resend.

Table 12. Message Integrity of Conversations

Conversation Attributes	Normal	LU 6.2 Session Failure	Other Failure
Synchronous Sync_level=NONE	Input: Reliable Output: Reliable	Input: PWS resend Output: PWS resend	Input: Reliable Output: Reliable
Synchronous Sync_level=CONFIRM	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable
Synchronous Sync_level=SYNCPT	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable
Asynchronous Sync_level=NONE	Input: Ambiguous Output: Reliable	Input: Undetectable Output: Reliable	Input: Undetectable Output: Reliable
Asynchronous Sync_level=CONFIRM	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable
Asynchronous Sync_level=SYNCPT	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable

Notes:
[1] A *session failure* is a network-connectivity breakage.
[2] A *non-session failure* is any other kind of failure, such as invalid security authorization.
[3] IMS resends asynchronous output if CONFIRM is lost; therefore, the PWS must tolerate duplicate output.

Table 13 shows the specifics of the processing windows when integrity is compromised (the message is either lost or its state is ambiguous). The table indicates the relative probability of an occurrence of each window and whether output is lost or duplicated.

A Sync_level value of NONE does not apply to asynchronous output, because IMS always uses Sync_level=CONFIRM for such output.

Table 13. Results of Processing When Integrity is Compromised

Conversation Attributes	State of Window before Accepting Transaction	Probability of Window State	Possible Action while Sending Response	Probability of Action while Sending Response
Synchronous Sync_level=NONE	ALLOCATE to PREPARE_TO_ RECEIVE return	Medium	Can lose or send duplicate output.	Medium
Synchronous Sync_level=CONFIRM	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small	CONFIRM to IMS receipt. Can cause duplicate output.	Small
Synchronous Sync_level=SYNCPT	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small	CONFIRM to IMS receipt. Can cause duplicate output.	Small
Asynchronous Sync_level=NONE	Allocate to Deallocate	High	CONFIRMED to IMS receipt. Can cause duplicate output.	Small

Table 13. Results of Processing When Integrity is Compromised (continued)

Conversation Attributes	State of Window before Accepting Transaction	Probability of Window State	Possible Action while Sending Response	Probability of Action while Sending Response
Asynchronous Sync_level=CONFIRM	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small	CONFIRMED to IMS receipt. Can cause duplicate output.	Small
Asynchronous Sync_level=SYNCPT	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small	CONFIRMED to IMS receipt. Can cause duplicate output.	

Notes:
¹ The term *window* refers to a period of time when certain events can occur, such as the consequences described in this table.
² Can be recoverable.

Table 14 indicates how IMS recovers APPC transactions across IMS warm starts, XRF takeovers, APPC session failures, and MSC link failures.

Table 14. Recovering APPC Messages

Message Type	IMS Warm Start (NRE or ERE)	XRF Takeover	APPC (LU 6.2) Session Fail	MSC LINK Failure
Local Recoverable Tran., Non Resp., Non Conversation				
- APPC Sync. Conv. Mode	Discarded (2)	Discarded (4)	Discarded (6)	N/A (9)
- APPC Async. Conv. Mode	Recovered	Recovered	Recovered (1)	N/A (9)
Local Recoverable Tran., Conv. or Resp. mode				
- APPC Sync. Conv. Mode	Discarded (2)	Discarded (4)	Discarded (6)	N/A (9)
- APPC Async. Conv. Mode	N/A (8)	N/A (8)	N/A (8)	N/A (8,9)
Local Non Recoverable Tran.,				
- APPC Sync. Conv. Mode	Discarded (2)		Discarded (6)	N/A (9)
- APPC Async. Conv. Mode	Discarded (2)	Discarded (4)	Recovered (1)	N/A (9)
Remote Recoverable Tran., Non Resp., Non Conv.				
- APPC Sync. Conv. Mode	Discarded (2,5)	Discarded (3,5)	Recovered (1)	Recovered (7)
- APPC Async. Conv. Mode	Recovered	Recovered	Recovered (1)	Recovered (7)
Remote Recoverable Tran., Conv. or Resp. mode				
- APPC Sync. Conv. Mode	Discarded (2,5)	Discarded (3,5)	Recovered (1)	Recovered (7)
- APPC Async. Conv. Mode	N/A (8)	N/A (8)	N/A (8)	N/A (8)
Remote Non Recoverable Tran.,				
- APPC Sync. Conv. Mode	Discarded (2,5)	Discarded (3,5)	Recovered (1)	Recovered (7)
- APPC Async. Conv. Mode	Discarded (2,5)	Discarded (3,5)	Recovered (1)	Recovered (7)

Table 14. Recovering APPC Messages (continued)

Message Type	IMS Warm Start (NRE or ERE)	XRF Takeover	APPC (LU 6.2) Session Fail	MSC LINK Failure

Notes:

1. This recovery scenario assumes the message was enqueued before failure; otherwise, the message is discarded.
2. The message is discarded during IMS warm-start processing.
3. The message is discarded when the MSC link is restarted and when the message is taken off the queue (for sending across the link).
4. The message is discarded when the message region is started and when the message is taken off the queue (for processing by the application program).
5. For all remote MSC APPC transactions, if the message has already been sent across the MSC link to the remote system when the failure occurs in the local IMS, the message is processed. After the message is processed by the remote application program and a response message is sent back to the local system, it is enqueued to the DFSASYNC TP name of the LU 6.2 device or program that submitted the original transaction.
6. At sync point, the User Message Control Error exit routine (DFSCMUX0) can prevent the transaction from being aborted and the output message can be rerouted (recovered).

   For more information about this exit routine, see *IMS/ESA Customization Guide*.
7. The standard MSC Link recovery protocol recovers all messages that are queued or are in the process of being sent across the MSC link when the link fails.
8. IMS conversational-mode and response-mode transactions cannot be submitted from APPC asynchronous conversation sessions. APPC synchronous conversation-mode must be used.
9. MSC link failures do not affect local transactions.

DFSAPPC Message Switch

DFSAPPC is an LU 6.2 descriptor that provides an IMS system service. It allows LU 6.2 application programs to send messages to the following:

- Application programs (transactions)
- IMS-managed local or remote LTERMs (message switches)
- LU name and TP name

Messages sent with the LTERM= option are directed to IMS-managed local or remote LTERMs. Messages sent without the LTERM= option are sent to the appropriate LU 6.2 application or IMS application program.

Because the LTERM can be an LU 6.2 descriptor name, the message is sent to the LU 6.2 application program as if an LU 6.2 device had been explicitly selected.

With DFSAPPC, message delivery is asynchronous. If a message is allocated and the allocate fails, the message is held on the IMS message queue until it can be successfully delivered.

Example: In an LU 6.2 conversation shown below an IMS application issues a DFSAPPC message switch to its partner with the LU name FRED and TPN name REPORT. REPI is the user data.

```
DFSAPPC  (TPN=REPORT LU=FRED) REP1
```

You can use a 17-byte network-qualified name in the LU= field.

Restriction: LU 6.2 architecture prohibits the use of the ALTRESP PCB on a `CHNG` call in an LU 6.2 conversation. The LU 6.2 conversation can only be associated with the IOPCB. The application sends a message on the existing LU 6.2

conversation (synchronous) or has IMS create a new conversation (asynchronous) using the IOPCB. Since there is no LTERM associated with an LU 6.2 conversation, only the IOPCB represents the original LU 6.2 conversation.

Related Reading: For detailed information about DFSAPPC, see *IMS/ESA Administration Guide: Transaction Manager*.

Chapter 8. Writing ODBA Application Programs

This chapter describes the Open Database Access (ODBA) callable interface to IMS DB and presents information about how you write OS/390 application programs that use the interface. By using the ODBA interface, IMS DB databases can be accessed from environments that are outside the scope of IMS's control, such as DB2 stored procedures. The ODBA interface is not needed within IMS-controlled regions, such as MPRs, BMPs, or IFPs, for calls to locally-controlled databases.

The OS/390 application programs (hereafter called the ODBA application programs) run in a separate OS/390 address space that IMS regards as a separate region from the control region. The separate OS/390 address space will hereafter be called the OS/390 application region.

The ODBA interface gains access to IMS DB through the Database Resource Adapter (DRA). The ODBA application programs (which can access any address space within the MVS they are running in) gain access to IMS DB databases through the ODBA interface. See Figure 49, which illustrates this concept and shows the relationship between the components of this environment.

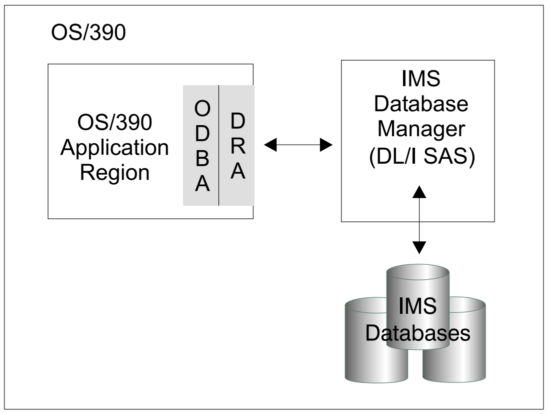


Figure 49. OS/390 Application Region's Connection to IMS DB

One OS/390 application region can connect to multiple IMS DBs and multiple OS/390 application regions can connect to a single IMS DB. The connection is similar to that of CICS to DBCTL.

Requirement: RRS/MVS is required; therefore, OS/390 Release 3 is the minimum level that can support the ODBA interface.

Related Reading: For a description of RRS and its uses, see Chapter 11, "Distributed Sync Point" in the *IMS Version 6 Release Planning Guide*.

General Application Program Flow

OS/390 application programs issue DL/I calls using an application interface block (AIB). No other interface is supported.

Restriction: The ODBA interface does not support calls into batch DL/I regions.

Several conditions must be met for the AIB call to succeed:

1. If an AIB is not passed in the call, a U261 abend is issued.
2. If the AIB that is passed is not valid, a U476 abend is issued.
3. If the AIB that is passed is not large enough (264 bytes), the AIB return and reason codes are set to X'104' and X'228'.
4. If the AIB that is passed is not on a fullword boundary, the MVS system will return an abend S201.
5. If there are other internal problems with the call, other return and reason codes are passed back to the OS/390 application program. See *IMS/ESA Application Programming: Database Manager* for a complete list of these return and reason codes.

The OS/390 must link edit with a language module (DFSCDLI0) or this module can be loaded into the OS/390 application region. The entry point for DFSCDLI0 is AERTDLI.

A simple example of the program flow of an OS/390 application program is:
1. Connect to IMS DB.
2. Allocate a PSB.
3. Perform DB calls.
4. Commit the changes.
5. Deallocate the PSB.
6. Terminate the connection.

The following sections discuss this flow.

Connect to IMS DB

The OS/390 application region must connect to an IMS DB before an OS/390 application program can begin to issue DL/I calls. The OS/390 application region can connect to multiple IMS DB subsystems concurrently and then identify the IMSID at connect time. The connection to a particular IMS DB is made if the CIMS INIT call refers to a specific IMSID.

The CIMS INIT call can also be used (without the IMSID) to initialize the environment and then the APSB call would make the connection.

The form of the connection call is:

```
CALL AERTDLI parmcount, CIMS, AIB
```

Where:

CIMS Is the required call function.

AIB Has the following fields:

AIBSFUNC
The subfunction is INIT. This field is mandatory.

AIBRSNM1
An optional field that provides an eyecatcher identifier of the application server that is associated with the AIB. This field is 8 bytes.

AIBRSNM2
Provides the optional 4-byte IMSID. The ID is optional if the call is issued as preconditioning. If the ID is given, the connection to the specified IMS DB is made.

Startup Table

The characteristics of the connection are determined from the DRA startup table. You should have one startup table for each IMS DB connection. The startup table name is DFSxxxx, where xxxx is the 4-byte subsystem ID.

For the details of building a DRA startup table, see *Installation Volume 2: System Definition and Tailoring*.

Allocate a PSB

The APSB call, introduced for CPIC-driven programs, is used with the ODBA interface to allocate a PSB for the OS/390 application region. Security is checked before the call can succeed. See *IMS/ESA Installation Volume 2: System Definition and Tailoring* for details. The APSB call is in the following form:

```
CALL AERTDLI parmcount, APSB, AIB
```

Where:

APSB Is the required call function.

AIB Is the name of the application interface block. The fields in the AIB must be filled in:

AIBRSNM1
Is the 8-character PSB name.

AIBRSNM2
Is the 4-byte IMSID.

Several conditions must be met for the allocation request to succeed.
1. The PSB must exist and security checking through RACF must succeed.
2. A CIMS INIT call must have been successful.
3. RRS/MVS must be active when the APSB call is made.

Multiple PSBs can be active at the same time, which is typical for server environments. No token is specifically provided to identify which PSB is to be used for a given call to a given IMS DB, so the same AIB *must* be used for all calls to the same PSB instance (APSB, DB calls, DPSB). This enables multiple instances of the same PSB to be in use for the same IMS DB at the same time. The parallelism is controlled by the thread count specified in the startup table. The maximum number of threads and dependent regions supported by an IMS DB instance is 999.

Perform DB Calls

All DL/I calls, with a few exceptions, are supported through the AIB. The unsupported calls entail message handling (the IOPCB is available only for system calls), CKPT, ROLL, ROLB, and INQY PROGRAM. Alternate destination PCBs cannot be used. Both full-function databases and DEDBs are available.

Commit Changes

Synchronization is performed by issuing the distributed commit calls, SRRCMIT or ATRCMIT, or possibly their rollback forms of SRRBACK or ATRBACK. See the

IMS/ESA Release Planning Guide for a discussion of distributed commit. IMS sync-point calls are not allowed. Commit is effective for all RRS/MVS controlled resources in the MVS task.

Deallocate the PSB

The DPSB call is used when the work unit is complete. A commit call must be issued before a DPSB call can be issued.

The DPSB call is in the following form:

```
CALL AERTDLI parmcount, DPSB, AIB
```

Where:

DPSB Is the required call function.

AIB Is the name of the application interface block. The following fields in the AIB must be filled in:

AIBRSNM1
Is the 8-character PSB name.

AIBSFUNC
Is an optional field that is used when you want to deallocate the PSB before the commit processing is initialized and when the commit processing is provided for the application, but outside it.

IMS performs phase 1 commit processing, returns control to the requestor, but holds the in-doubt work until RRS/MVS, as the commit manager, requests full commit processing. An example is in DB2 Stored Procedures, where DB2 initializes commit processing on behalf of the procedure. See “DB2 Stored Procedures Use of ODBA” on page 133 for a discussion of this scenario.

Terminate the Connection

The termination call is in the following form:

```
CALL AERTDLI parmcount, CIMS, AIB
```

Where:

CIMS Is the required call function.

AIB Is the name of the application interface block. The following fields in the AIB must be filled in:

AIBSFUNC
Is a mandatory field whose subfunction is TERM or TALL.

AIBRSNM1
Is an optional field that provides an eyecatcher identifier of the application server associated with the AIB. This field has an 8-byte value.

AIBRSNM2
Provides the 4-byte IMSID when the subfunction equals TERM and is not needed when the subfunction equals TALL.

Use the IMSID if a single IMS DB connection is to be served. TALL will terminate all connections for this OS/390 application region and remove the DRA from the address space.

Server Program Structure

The commit scope within the OS/390 application environment is all the work under the TCB from which the commit request is made to RRS/MVS. Server environments, therefore, need a separate TCB under which the individual client requests will be managed. Each TCB will map to a PST for thread handling.

Figure 50 shows an example TCB structure for a server environment.

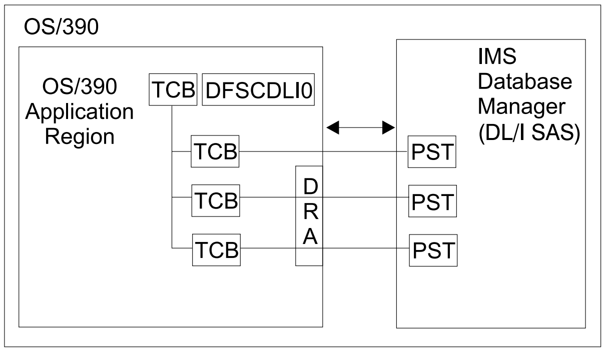


Figure 50. DRA Uses One TCB per Thread

Each connection to an IMS DB uses a thread under the TCB. When the APSB call is processed, a context is established and tied to the TCB. At commit time, all contexts for this TCB are committed or aborted by RRS/MVS.

Loading DFSCDLI0 rather than link editing is attractive when the OS/390 application region is a server supporting many clients with many instances of threads connected with the IMS DBs.

DB2 Stored Procedures Use of ODBA

DB2 Stored Procedures are a special case of the general server structure described in “Server Program Structure”. Stored Procedures connecting to ODBA require DB2 Version 5 or later and must run in a work load manager (WLM)-managed stored procedures address space.

DB2 establishes the ODBA environment by issuing the INIT subcall for the stored procedure address space. Connection to a specific IMS DB occurs when the APSB call is issued.

Each stored procedure running in the stored procedure address space runs under its own TCB that was established by DB2 when the stored procedure is initialized. DB2 issues the commit call on behalf of the stored procedure when control is returned to DB2. Only the PREP subfunction of the DPSB call should be issued by the stored procedures themselves.

Figure 51 on page 134 illustrates the connection from a DB2 Stored Procedures address space to an IMS DB subsystem. This connection allows DL/I data to be presented through an SQL interface, either locally to this DB2 or to DRDA connected DB2s

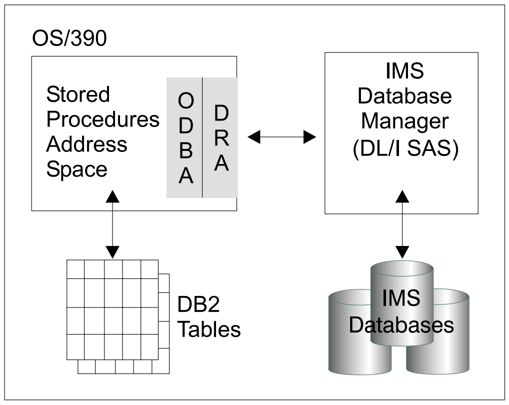

OS/390
Stored Procedures Address Space
ODBA
DRA
IMS Database Manager (DL/I SAS)
DB2 Tables
IMS Databases

Figure 51. DB2 Stored Procedures Connection to IMS DB

Figure 52 illustrates the general relationships involved with using DB2 Stored Procedures and IMS DB together.

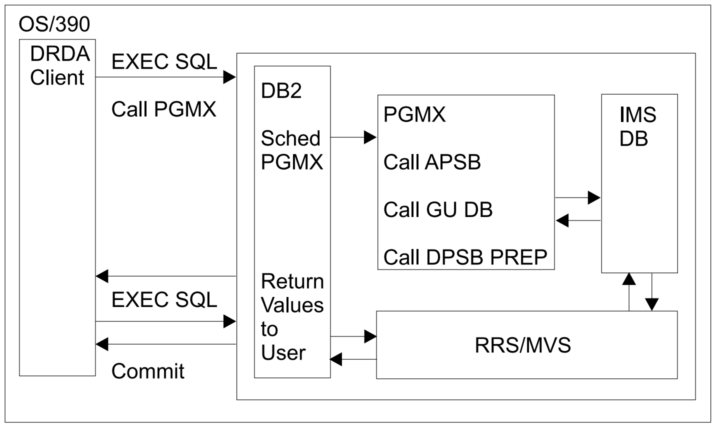


Figure 52. DB2 Stored Procedures Relationships

Chapter 9. Testing an IMS Application Program

This chapter tells you what is involved in testing an IMS application program (as a unit) and provides suggestions on how to do it. The purpose of this test, called a *program unit test*, is to ensure that the program correctly handles its input data, processing, and output test data.

The amount and type of testing you do depends on the individual program you are testing. Though no strict rules for testing are available, the guidelines offered in this chapter might be helpful.

In this Chapter:

What You Need to Test a Program

Before you start testing your program, be aware of the test procedures at your installation. To start testing, you need the following three items:

- Test JCL.
- A test database. Never test a program using a production database because the program, if faulty, might damage valid data.
- Test input data. The input data that you use need not be current, but it should be valid. You cannot 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 the situations that it might encounter. To thoroughly test the program, try to test as many of the paths that the program can take as possible.

Recommendations:

- Test each path in the program by using 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)

The DL/I test program, DFSDDLT0, is an IMS application program that executes the DL/I calls you specify against any database.

Restriction: DFSDDLT0 does not work if you are using a coordinator controller (CCTL).

Testing DL/I Call Sequences

An advantage of using DFSDDLT0 is that you can test the DL/I call sequence you will use prior to coding your program. 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 correct. When you test the program, and it does not execute correctly, you know that the DL/I calls are not part of the problem if you have 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 are 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 have defined for the segment, and the effect of the call. DFSDDLT0 is helpful in debugging because it can display IMS application control blocks.

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.

Comment statements let you choose whether 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.

Related Reading: For more details about using DFSDDLT0, and how to check call sequence performance, see *IMS/ESA Application Programming: Database Manager* and *IMS/ESA Application Programming: Transaction Manager*.

Using BTS II to Test Your Program

BTS II (Batch Terminal Simulator II) is a valuable tool for testing programs because you can use it to test call sequences. The documentation BTS II produces is helpful in debugging. You can also test online application programs without actually running them online.

Restriction: BTS II does not work if you are using a CCTL or running under DBCTL.

Related Reading: For information about how to use BTS II, refer to *BTS Program Reference/Operations Manual*.

Tracing DL/I Calls with Image Capture

The DL/I image capture program (DFSDLTR0) is a trace program that can trace and record DL/I calls issued by all types of IMS application programs.

Restriction: The image capture program does not trace calls to Fast Path databases.

You can run the image capture program in a DB/DC or a batch environment to:

Test your program

If the image capture program detects an error in a call it traces, it reproduces as much of the call as possible, although it cannot document where the error occurred, and cannot always reproduce the full SSA.

Produce input for DFSDDLT0

You can use the output produced by the image capture program as input to DFSDDLT0. The image capture program produces status statements, comment statements, call statements, and compare statements for DFSDDLT0.

Debug your program

When your program terminates abnormally, you can rerun the program using the image capture program, which can then reproduce and document the conditions that led to the program failure. You can use the information in the report produced by the image capture program to find and fix the problem.

Using Image Capture with DFSDDLT0

The image capture program produces the following control statements that you can use as input to DFSDDLT0:

Status statements

When you invoke the image capture program, it produces the status statement. The status statement it produces:

– Sets print options so that DFSDDLT0 prints all call trace comments, all DL/I calls, and the results of all comparisons.

– Determines the new relative PCB number each time a PCB change occurs while the application program is executing.

Comments statement

The image capture program also produces a comments statement when you invoke it. The comments statements give:

– The time and date IMS started the trace

– The name of the PSB being traced

The image capture program also produces a comments statement preceding any call in which IMS finds an error.

Call statements

The image capture program produces a call statement for each DL/I call the application program issues. It also generates a `CHKP` call when it starts the trace and after each commit point or `CHKP` request.

Compare statements

The image capture program produces data and PCB comparison statements if you specify COMP on the `TRACE` command (if you run the image capture program online), or on the DLITRACE control statement (if you run the image capture program as a batch job).

Restrictions on Using Image Capture Output

The status statement of the image capture call is based on relative PCB position. When the PCB parameter LIST=NO has been specified, the status statement may need to be changed to select the PCB as follows:

- If all PCBs have the parameter LIST=YES, the status statement does not need to be changed.
- If all PCBs have the parameter LIST=NO, the status statement needs to be changed from the relative PCB number to the correct PCB name.

- If some PCBs have the parameter LIST=YES and some have the parameter LIST=NO, the status statement needs to be changed as follows:
  – The PCB relative position is based on all PCBs as if LIST=YES.
  – For PCBs that have a PCB name, the status statement can be changed to use the PCB name based on a relative PCB number.
  – For PCBs that have LIST=YES and no PCB name, change the relative PCB number to refer to the relative PCB number in the user list by looking at the PCB list using LIST=YES and LIST=NO.

Running Image Capture Online

When you run the image capture program online, the trace output goes to the IMS log data set. To run the image capture program online, you issue the IMS TRACE command from the IMS master terminal.

If you trace a BMP or an MPP and you want to use the trace results with DFSDDLT0, the BMP or MPP must have exclusive write access to the databases it processes. If the application program does not have exclusive access, the results of DFSDDLT0 may differ from the results of the application program. When you trace a BMP that accesses GSAM databases, you must include an //IMSERR DD statement to get a formatted dump of the GSAM control blocks.

The following diagram shows the TRACE command format:

```
                    ┌─ON──┐
►►──/──TRACE──SET───┴─OFF─┴───PSB──psbname──┬───────────────┬──────────────────►◄
                                            │  ┌─NOCOMP─┐   │
                                            └──┴─COMP───┴───┘
```

SET ON|OFF
  Turns the trace on or off.

PSB *psbname*
  Specifies the name of the PSB you want to trace. You can trace more than one PSB at the same time by issuing a separate TRACE command for each PSB.

COMP|NOCOMP
  Specifies whether you want the image capture program to produce data and PCB compare statements to be used as input to DFSDDLT0.

Running Image Capture as a Batch Job

To run the image capture program as a batch job, you use the DLITRACE control statement in the DFSVSAMP DD data set. In the DLITRACE control statement, you specify:

- Whether you want to trace all of the DL/I calls the program issues or trace only a certain group of calls.
- Whether you want the trace output to go to:
  A sequential data set that you specify
  The IMS log data set
  Both sequential and IMS log data sets

Format of DLITRACE Control Statement

The format of the DLITRACE control statement is:

```
►►──DLITRACE──┬──────────────────┬──┬──────────────────────────────┬──┬──────────────────────┬──────────────────►◄
              │          ┌─NO──┐ │  │               ┌─DFSTROUT──┐ │  │             ┌─0─┐    │
              └─LOG──=───┴─YES─┴─┘  └─,──DDNAME──=──┴─anyname───┴─┘  └─,──START──=──┴─n─┴────┘

►►──┬──────────────────────────┬──┬──────────────┬──────────────────────────────────────────────────────────────►◄
    │              ┌─7FFFFFFF─┐│  │ ┌─,──NOCOMP─┐│
    └─,──STOP──=───┴─n────────┴┘  └─┴─,──COMP───┴┘
```

DLITRACE
: Invokes the trace. If this is the only parameter you specify, IMS uses default values for the remaining parameters.

LOG = YES|NO
: Specifies whether you want IMS to route trace output to the IMS log. NO is the default.

DDNAME = anyname|DFSTROUT
: Specifies the ddname of the sequential data set to which you want trace output sent. The default is DFSTROUT. However, if you specify LOG = YES, and you do not supply a value for the DDNAME parameter, IMS does not try to open the data set defined by DFSTROUT.

START = n|0
: Gives the number of the first Dl/I call in the program that you want traced. The value is a one- to eight-character hexadecimal number. If you do not supply a value, the trace starts with the first DL/I call in the program.

STOP = n|7FFFFFFF
: Specifies the number of the last DL/I call in the program that you want traced. The value is a one- to eight-character hexadecimal number. The default and maximum is X'7FFFFFFF'.

COMP|NOCOMP
: Specifies whether you want DFSDLTR0 to produce data and PCB comparison statements that are to be used as input to DFSDDLT0. NOCOMP is the default.

Notes:

1. DLITRACE must begin in column 1; the remainder of the parameters are nonpositional.
2. Each parameter can be used only once in the control statement.
3. Only one trace control statement is allowed for a program.

Example of DLITRACE

The example below shows a DLITRACE control statement that:

- Traces the first 14 DL/I calls or commands that the program issues
- Sends the output to the IMS log data set
- Produces data and PCB comparison statements for DFSDDLT0, program

```
//DFSVSAMP DD *
DLITRACE LOG=YES,STOP=14,COMP
/*
```

Special JCL Requirements

Special JCL requirements are as follows:

//IEFRDER DD
: If you want log data set output, this DD statement is required to define the IMS log data set.

//DFSTROUT DD|anyname

If you want sequential data set output, this DD statement is required to define that data set. If you want to specify an alternate ddname (anyname), specify it by using the DDNAME parameter on the DLITRACE control statement.

DCB parameters are required for this DD statement only when you want to use the output for problem determination and the program being traced abends. In this case, add the BLKSIZE=80 parameter to the DFSTROUT DD statement to ensure that all the generated output is written to the data set. Buffering the output may leave some of the traced data in the output buffers at abend time. If the BLKSIZE= parameter is not specified in the JCL for the DFSTROUT DD statement, the block size defaults to a system generated block size. The DCB parameters generated in the trace program are:

- RECFM=F
- LRECL=80

Notes on Using Image Capture

- If the program being traced issues `CHKP` and `XRST` calls, the checkpoint and restart information may not be directly reproducible when you use the trace output with DFSDDLT0.
- When you run DFSDDLT0 in an IMS DL/I or DBB batch region with trace output, the results are the same as the application program's results, but only if the database has not been altered.

Retrieving Image Capture Data from the Log Data Set

If the trace output is sent to the IMS log data set, you can retrieve it by using utility DFSERA10 and a DL/I call trace exit routine, DFSERA50. DFSERA50 deblocks, formats, and numbers the image capture program records that are to be retrieved. To use DFSERA50, you must insert a DD statement defining a sequential output data set in the DFSERA10 input stream. The default ddname for this DD statement is TRCPUNCH. The statement must specify BLKSIZE=80.

Examples: You can use the following examples of DFSERA10 input control statements in the SYSIN data set to retrieve the image capture program data from the log data set:

- **Print all image capture program records:**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,FLDTYP=X
```

- **Print selected image capture program records by PSB name:**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT OFFSET=25,VLDTYP=C,FLDLEN=8,
               VALUE=psbname, COND=E
```

- **Format image capture program records (in a format that can be used as input to DFSDDLT0):**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT EXITR=DFSERA50,OFFSET=25,FLDTYP=C
               VALUE=psbname,FLDLEN=8,DDNAME=OUTDDN,COND=E
```

Point to remember: The DDNAME= parameter names the DD statement to be used by DFSERA50. The data set that is defined on the OUTDDN DD statement is used instead of the default TRCPUNCH DD statement. For this example, the DD is:

```
//OUTDDN DD ...,DCB=(BLKSIZE=80),...
```

Requests for Monitoring and Debugging

You can use the following two requests to help you in debugging your program:

- The Statistics (`STAT`) call retrieves database statistics.
- The Log (`LOG`) call makes it possible for the application program to write a record on the system log.

The enhanced OSAM and VSAM `STAT` calls provide additional information for monitoring performance and fine tuning of the system for specific needs.

When the enhanced `STAT` call is issued, the following information is returned:

- OSAM statistics for each defined subpool
- VSAM statistics that also include hiperspace statistics
- OSAM and VSAM count fields that have been expanded to 10 digits

Retrieving Database Statistics: The STAT Call

This section contains product-sensitive programming interface information.

The `STAT` call is helpful in debugging a program because it retrieves IMS database statistics. It is also helpful in monitoring and fine tuning for performance. The `STAT` call retrieves OSAM database buffer pool statistics and VSAM database buffer subpool statistics.

Related Reading: For information on coding the `STAT` call, see the appropriate application programming book as described in “How to Use This Book” on page xiii.

When you issue the `STAT` call, you indicate:

- An I/O area into which the statistics are to be returned.
- A statistics function, which is the name of a 9-byte area whose contents describe the type and format of the statistics you want returned. The contents of the area are defined as follows:
  - The first 4 bytes define the type of statistics desired (OSAM or VSAM).
  - The 5th byte defines the format to be returned (formatted, unformatted, or summary).
  - The remaining 4 bytes are defined as follows:
    - The normal or enhanced STAT call contains 4 bytes of blanks.
    - The extended STAT call contains the 4-byte parameter ’ E1 ’ (a 1-byte blank, followed by a 2-byte character string, and then another 1-byte blank).

Format of OSAM Buffer Pool Statistics

For OSAM buffer pool statistics, the values are possible for the stat-function parameter and for the format of the data that is returned to the application program. If no OSAM buffer pool is present, a GE status code is returned to the program.

DBASF: This function value provides the full OSAM database buffer pool statistics in a formatted form. The application program I/O area must be at least 360 bytes. Three 120-byte records (formatted for printing) are provided as two heading lines and one line of statistics. The following diagram shows the data format.

```
  BLOCK    FOUND    READS     BUFF     OSAM   BLOCKS      NEW   CHAIN
    REQ  IN POOL   ISSUED     ALTS   WRITES  WRITTEN   BLOCKS  WRITES
nnnnnnn  nnnnnnn    nnnnn  nnnnnnn  nnnnnnn  nnnnnnn    nnnnn   nnnnn

WRITTEN  LOGICAL    PURGE  RELEASE
 AS NEW      CYL      REQ      REQ  ERRORS
          FORMAT
nnnnnnn  nnnnnnn  nnnnnnn  nnnnnnn  nn/nn
```

BLOCK REQ	Number of block requests received.
FOUND IN POOL	Number of times the block requested was found in the buffer pool.
READS ISSUED	Number of OSAM reads issued.
BUFF ALTS	Number of buffers altered in the pool.
OSAM WRITES	Number of OSAM writes issued.
BLOCKS WRITTEN	Number of blocks written from the pool.
NEW BLOCKS	Number of new blocks created in the pool.
CHAIN WRITES	Number of chained OSAM writes issued.
WRITTEN AS NEW	Number of blocks created.
LOGICAL CYL FORMAT	Number of format logical cylinder requests issued.
PURGE REQ	Number of purge user requests.
RELEASE REQ	Number of release ownership requests.
ERRORS	Number of write error buffers currently in the pool or the largest number of errors in the pool during this execution.

DBASU: This function value provides the full OSAM database buffer pool statistics in an unformatted form. The application program I/O area must be at least 72 bytes. Eighteen fullwords of binary data are provided:

Word	**Contents**
1	A count of the number of words that follow.
2-18	The statistic values in the same sequence as presented by the DBASF function value.

DBASS: This function value provides a summary of the OSAM database buffer pool statistics in a formatted form. The application program I/O area must be at least 180 bytes. Three 60-byte records (formatted for printing) are provided. The following diagram shows the data format.

```
DATA BASE BUFFER POOL:  SIZE nnnnnnn
  REQ1 nnnnn REQ2 nnnnn READ nnnnn WRITES nnnnn LCYL nnnnn
  PURG nnnnn OWNRR nnnnn ERRORS nn/nn
```

SIZE	Buffer pool size.
REQ1	Number of block requests.
REQ2	Number of block requests satisfied in the pool plus new blocks created.
READ	Number of read requests issued.
WRITES	Number of OSAM writes issued.
LCYL	Number of format logical cylinder requests.
PURG	Number of purge user requests.

OWNRR	Number of release ownership requests.
ERRORS	Number of permanent errors now in the pool or the largest number of permanent errors during this execution.

Format of VSAM Buffer Subpool Statistics

Because there might be several buffer subpools for VSAM databases, the STAT call is iterative when requesting these statistics. If more than one VSAM local shared resource pool is defined, statistics are retrieved for all VSAM local shared resource pools in the order in which they are defined. For each local shared resource pool, statistics are retrieved for each subpool according to buffer size.

The first time the call is issued, the statistics for the subpool with the smallest buffer size are provided. For each succeeding call (without intervening use of the PCB), the statistics for the subpool with the next-larger buffer size are provided.

If index subpools exist within the local shared resource pool, the index subpool statistics always follow statistics of the data subpools. Index subpool statistics are also retrieved in ascending order based on the buffer size.

The final call for the series returns a GA status code in the PCB. The statistics returned are totals for all subpools in all local shared resource pools. If no VSAM buffer subpools are present, a GE status code is returned to the program.

VBASF: This function value provides the full VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 360 bytes. Three 120-byte records (formatted for printing) are provided as two heading lines and one line of statistics. Each successive call returns the statistics for the next data subpool. If present, statistics for index subpools follow the statistics for data subpools.

The following diagram shows the data format.

```
                   BUFFER HANDLER STATISTICS
BSIZ NBUF RET RBA RET KEY ISRT ES ISRT KS BFR ALT   BGWRT SYN PTS
nnnK  nnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn

                 VSAM STATISTICS POOLID: xxxx
   GETS  SCHBFR   FOUND   READS USR WTS NUR WTS ERRORS
nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn  nn/nn
```

POOLID	ID of the local shared resource pool.
BSIZ	Size of the buffers in this VSAM subpool. In the final call, this field is set to ALL.
NBUF	Number of buffers in this subpool. In the final call, this is the number of buffers in all subpools.
RET RBA	Number of retrieve-by-RBA calls received by the buffer handler.
RET KEY	Number of retrieve-by-key calls received by the buffer handler.
ISRT ES	Number of logical records inserted into ESDSs.
ISRT KS	Number of logical records inserted into KSDSs.
BFR ALT	Number of logical records altered in this subpool. Delete calls that result in erasing records from a KSDS are not counted.
BGWRT	Number of times the background-write function was executed by the buffer handler.
SYN PTS	Number of Synchronization calls received by the buffer handler.
GETS	Number of VSAM GET calls issued by the buffer handler.

SCHBFR	Number of VSAM SCHBFR calls issued by the buffer handler.
FOUND	Number of times VSAM found the control interval already in the subpool.
READS	Number of times VSAM read a control interval from external storage.
USR WTS	Number of VSAM writes initiated by IMS.
NUR WTS	Number of VSAM writes initiated to make space in the subpool.
ERRORS	Number of write error buffers currently in the subpool or the largest number of write errors in the subpool during this execution.

VBASU: This function value provides the full VSAM database subpool statistics in a unformatted form. The application program I/O area must be at least 72 bytes. Eighteen fullwords of binary data are provided for each subpool:

Word	Contents
1	A count of the number of words that follow.
2-18	The statistic values in the same sequence as presented by the VBASF function value, except for POOLID, which is not included in this unformatted form.

VBASS: This function value provides a summary of the VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 180 bytes. Three 60-byte records (formatted for printing) are provided.

The following diagram shows the data format.

```
DATA BASE BUFFER POOL:  BSIZE nnnnnnn  POOLID xxxx  Type x
  RRBA nnnnn RKEY nnnnn BFALT nnnnn NREC nnnnn SYN PTS nnnnn
  NMBUFS nnn VRDS nnnnn FOUND nnnnn VWTS nnnnn  ERRORS nn/nn
```

BSIZE	Size of the buffers in this VSAM subpool.
POOLID	ID of the local shared resource pool.
TYPE	Indicates a data (D) subpool or an index (I) subpool.
RRBA	Number of retrieve-by-RBA requests.
RKEY	Number of retrieve-by-key requests.
BFALT	Number of logical records altered.
NREC	Number of new VSAM logical records created.
SYN PTS	Number of sync point requests.
NMBUFS	Number of buffers in this VSAM subpool.
VRDS	Number of VSAM control interval reads.
FOUND	Number of times VSAM found the requested control interval already in the subpool.
VWTS	Number of VSAM control interval writes.
ERRORS	Number of permanent write errors now in the subpool or the largest number of errors in this execution.

Format of Enhanced/Extended OSAM Buffer Subpool Statistics

The enhanced OSAM buffer pool statistics provide additional information generated for each defined subpool. Because there might be several buffer subpools for OSAM databases, the enhanced `STAT` call repeatedly requests these statistics. The first time the call is issued, the statistics for the subpool with the smallest buffer size is provided. For each succeeding call (without intervening use of the PCB), the statistics for the subpool with the next-larger buffer size is provided.

The final call for the series returns a GA status code in the PCB. The statistics returned are the totals for all subpools. If no OSAM buffer subpools are present, a GE status code is returned.

Extended OSAM buffer pool statistics can be retrieved by including the 4-byte parameter 'ƀE1ƀ' following the enhanced call function. The extended stat call returns all of the stats returned with the enhanced call, plus the stats on the coupling facility buffer invalidates, OSAM caching, and sequential buffering IMMED/SYNC read counts.

Restriction: The extended format parameter is supported by the DBESO, DBESU, and DBESF functions only.

DBESF: This function value provides the full OSAM subpool statistics in a formatted form. The application program I/O area must be at least 600 characters. For OSAM subpools, five 120-byte records (formatted for printing) are provided. Three of the records are heading lines and two of the records are lines of subpool statistics.

Example: The following shows the enhanced stat call format:

```
       B U F F E R   H A N D L E R   O S A M   S T A T I S T I C S    FIXOPT=X/X   POOLID: xxxx
BSIZ  NBUFS      LOCATE-REQ    NEW-BLOCKS   ALTER- REQ   PURGE- REQ  FND-IN-POOL  BUFRS-SRCH  READ- REQS   BUFSTL-WRT
                   PURGE-WRTS    WT-BUSY-ID   WT-BUSY-WR   WT-BUSY-RD   WT-RLSEOWN   WT-NO-BFRS    ERRORS
nn1K  nnnnnnnn   nnnnnnnnnn    nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn
                   nnnnnnnnnn    nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn    nnnnnnn/nnnnnnn
```

Example: The following shows the extended stat call format:

```
       B U F F E R   H A N D L E R   O S A M   S T A T I S T I C S   STG CLS=     FIXOPT=N/N  POOLID:
BSIZ  NBUFS      LOCATE-REQ    NEW-BLOCKS   ALTER- REQ   PURGE- REQ  FND-IN-POOL  BUFRS-SRCH  READ- REQS   BUFSTL-WRT
                   PURGE-WRTS    WT-BUSY-ID   WT-BUSY-WR   WT-BUSY-RD   WT-RLSEOWN   WT-NO-BFRS    ERRORS
nn1K  nnnnnnn5   nnnnnnnnn0    nnnnnnnnn0   nnnnnnnnn0   nnnnnnnnn0  nnnnnnnnn0   nnnnnnnnn0  nnnnnnnnn0   nnnnnnnnn0
                   nnnnnnnnnn    nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn    nnnnnnn/nnnnnnn
      CF-READS   EXPCTD-NF    CFWRT-PRI  CFWRT-CHG  STGCLS-FULL       XI-CNT        VECTR-XI    SB-SEQRD   SB-ANTICIP
      nnnnnnnnnn nnnnnnnnnn   nnnnnnnnnn nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn      nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn
```

FIXOPT	Fixed options for this subpool. Y or N indicates whether the data buffer prefix and data buffers are fixed.
POOLID	ID of the local shared resource pool.
BSIZ	Size of the buffers in this subpool. Set to ALL for total line.[1]
NBUFS	Number of buffers in this subpool. This is the total number of buffers in the pool for the ALL line.
LOCATE-REQ	Number of LOCATE-type calls.
NEW-BLOCKS	Number of requests to create new blocks.
ALTER-REQ	Number of buffer alter calls. This count includes NEW BLOCK and BYTALT calls.
PURGE-REQ	Number of PURGE calls.
FND-IN-POOL	Number of LOCATE-type calls for this subpool where data is already in the OSAM pool.
BUFRS-SRCH	Number of buffers searched by all LOCATE-type calls.
READ-REQS	Number of READ I/O requests.

BUFSTL-WRT	Number of single block writes initiated by buffer steal routine.
PURGE-WRTS	Number of blocks for this subpool written by purge.
WT-BUSY-ID	Number of LOCATE calls that waited due to busy ID.
WT-BUSY-WR	Number of LOCATE calls that waited due to buffer busy writing.
WT-BUSY-RD	Number of LOCATE calls that waited due to buffer busy reading.
WT-RLSEOWN	Number of buffer steal or purge requests that waited for ownership to be released.
WT-NO-BFRS	Number of buffer steal requests that waited because no buffers are available to be stolen.
ERRORS	Total number of I/O errors for this subpool or the number of buffers locked in pool due to write errors.
CF-READS	Number of blocks read from CF.
EXPCTD-NF	Number of blocks expected but not read.
CFWRT-PRI	Number of blocks written to CF (prime).
CFWRT-CHG	Number of blocks written to CF (changed).
STGGLS-FULL	Number of blocks not written (STG CLS full).
XI-CNTL	Number of XI buffer invalidate calls.
VECTR-XI	Number of buffers found invalidated by XI on VECTOR call.
SB-SEQRD	Number of immediate (SYNC) sequential reads (SB stat).
SB-ANTICIP	Number of anticipatory reads (SB stat).

Note: [1]For the summary totals (BSIZ=ALL), the FIXOPT and POOLID fields are replaced by an OSM= field. This field is the total size of the OSAM subpool.

DBESU: This function value provides full OSAM statistics in an unformatted form. The application program I/O area must be at least 84 bytes. Twenty-one fullwords of binary data are provided for each subpool:

Word	Contents
1	A count of the number of words that follow.
2-19	The statistics provided in the same sequence as presented by the DBESF function value.
20	The POOLID provided at subpool definition time.
21	The second byte contains the following fix options for this subpool: • X'04' = DATA BUFFER PREFIX fixed • X'02' = DATA BUFFERS fixed. The summary totals (word 2=ALL), for word 21, contain the total size of the OSAM pool.
22-30	Extended stat data in same sequence as on DBESF call.

DBESS: This function value provides a summary of the OSAM database buffer pool statistics in a formatted form. The application program I/O area must be at least 360 bytes. Six 60-byte records (formatted for printing) are provided. This STAT call is a restructured DBASF STAT call that allows for 10-digit count fields. In addition, the subpool header blocks give a total of the number of OSAM buffers in the pool.

The following shows the data format:

```
DATA BASE BUFFER POOL:  NSUBPL nnnnnn  NBUFS nnnnnnnn
  BLKREQ nnnnnnnnnn   INPOOL nnnnnnnnnn   READS  nnnnnnnnnn
  BUFALT nnnnnnnnnn   WRITES nnnnnnnnnn   BLKWRT nnnnnnnnnn
  NEWBLK nnnnnnnnnn   CHNWRT nnnnnnnnnn   WRTNEW nnnnnnnnnn
  LCYLFM nnnnnnnnnn   PURGRQ nnnnnnnnnn   RLSERQ nnnnnnnnnn
  FRCWRT nnnnnnnnnn   ERRORS nnnnnnnn/nnnnnnnn
```

NSUBPL	Number of subpools defined for the OSAM buffer pool.
NBUFS	Total number of buffers defined in the OSAM buffer pool.
BLKREQ	Number of block requests received.
INPOOL	Number of times the block requested is found in the buffer pool.
READS	Number of OSAM reads issued.
BUFALT	Number of buffers altered in the pool.
WRITES	Number of OSAM writes issued.
BLKWRT	Number of blocks written from the pool.
NEWBLK	Number of blocks created in the pool.
CHNWRT	Number of chained OSAM writes issued.
WRTNEW	Number of blocks created.
LCYLFM	Number of format logical cylinder requests issued.
PURGRQ	Number of purge user requests.
RLSERQ	Number of release ownership requests.
FRCWRT	Number of forced write calls.
ERRORS	Number of write error buffers currently in the pool or the largest number of errors in the pool during this execution.

DBESO: This function value provides the full OSAM database subpool statistics in a formatted form for online statistics that are returned as a result of a `/DIS POOL` command. This call can also be a user-application `STAT` call. When issued as an application DL/I `STAT` call, the program I/O area must be at least 360 bytes. Six 60-byte records (formatted for printing) are provided.

Example: The following shows the enhanced stat call format:

```
OSAM DB BUFFER POOL:ID xxxx BSIZE nnnnnK NBUFnnnnnnn  FX=X/X
  LCTREQ  nnnnnnnnnn  NEWBLK  nnnnnnnnnn  ALTREQ  nnnnnnnnnn
  PURGRQ  nnnnnnnnnn  FNDIPL  nnnnnnnnnn  BFSRCH  nnnnnnnnnn
  RDREQ   nnnnnnnnnn  BFSTLW  nnnnnnnnnn  PURGWR  nnnnnnnnnn
  WBSYID  nnnnnnnnnn  WBSYWR  nnnnnnnnnn  WBSYRD  nnnnnnnnnn
  WRLSEO  nnnnnnnnnn  WNOBFR  nnnnnnnnnn  ERRORS nnnnn/nnnnn
```

Example: The following shows the extended stat call format:

```
OSAM DB BUFFER POOL:ID xxxx BSIZE nnnnnK NBUFnnnnnnn  FX=X/X
  LCTREQ  nnnnnnnnnn  NEWBLK  nnnnnnnnnn  ALTREQ  nnnnnnnnnn
  PURGRQ  nnnnnnnnnn  FNDIPL  nnnnnnnnnn  BFSRCH  nnnnnnnnnn
  RDREQ   nnnnnnnnnn  BFSTLW  nnnnnnnnnn  PURGWR  nnnnnnnnnn
  WBSYID  nnnnnnnnnn  WBSYWR  nnnnnnnnnn  WBSYRD  nnnnnnnnnn
  WRLSEO  nnnnnnnnnn  WNOBFR  nnnnnnnnnn  ERRORS nnnnn/nnnnn
  CFREAD  nnnnnnnnnn  CFEXPC  nnnnnnnnnn  CFWRPR nnnnn/nnnnn
  CFWRCH  nnnnnnnnnn  STGCLF  nnnnnnnnnn  XIINV  nnnnn/nnnnn
  XICLCT  nnnnnnnnnn  SBSEQR  nnnnnnnnnn  SBANTR nnnnn/nnnnn
```

POOLID	ID of the local shared resource pool.
BSIZE	Size of the buffers in this subpool. Set to ALL for summary total line.[1]

NBUF	Number of buffers in this subpool. Total number of buffers in the pool for the ALL line.
FX=	Fixed options for this subpool. Y or N indicates whether the data buffer prefix and data buffers are fixed.
LCTREQ	Number of LOCATE-type calls.
NEWBLK	Number of requests to create new blocks.
ALTREQ	Number of buffer alter calls. This count includes NEW BLOCK and BYTALT calls.
PURGRQ	Number of PURGE calls.
FNDIPL	Number of LOCATE-type calls for this subpool where data is already in the OSAM pool.
BFSRCH	Number of buffers searched by all LOCATE-type calls.
RDREQ	Number of READ I/O requests.
BFSTLW	Number of single-block writes initiated by buffer-steal routine.
PURGWR	Number of buffers written by purge.
WBSYID	Number of LOCATE calls that waited due to busy ID.
WBSYWR	Number of LOCATE calls that waited due to buffer busy writing.
WBSYRD	Number of LOCATE calls that waited due to buffer busy reading.
WRLSEO	Number of buffer steal or purge requests that waited for ownership to be released.
WNOBRF	Number of buffer steal requests that waited because no buffers are available to be stolen.
ERRORS	Total number of I/O errors for this subpool or the number of buffers locked in pool due to write errors.
CFREAD	Number of blocks read from CF.
CFEXPC	Number of blocks expected but not read.
CFWRPR	Number of blocks written to CF (prime).
CFWRCH	Number of blocks written to CF (changed).
STGCLF	Number of blocks not written (STG CLS full).
XIINV	Number of XI buffer invalidate calls.
XICLCT	Number of buffers found invalidated by XI on VECTOR call.
SBSEQR	Number of immediate (SYNC) sequential reads (SB stat).
SBANTR	Number of anticipatory reads (SB stat).

Note: [1]For the summary totals (BSIZE=ALL), the FX= field is replaced by the OSAM= field. This field is the total size of the OSAM buffer pool. The POOLID is not shown.

Format of Enhanced VSAM Buffer Subpool Statistics

The enhanced VSAM buffer subpool statistics provide information on the total size of VSAM subpools in virtual storage and in hiperspace. All count fields are 10 digits.

Because there might be several buffer subpools for VSAM databases, the enhanced `STAT` call repeatedly requests these statistics. If more than one VSAM local shared resource pool is defined, statistics are retrieved for all VSAM local shared resource pools in the order in which they are defined. For each local shared resource pool, statistics are retrieved for each subpool according to buffer size.

The first time the call is issued, the statistics for the subpool with the smallest buffer size are provided. For each succeeding call (without intervening use of the PCB), the statistics for the subpool with the next-larger buffer size are provided.

If index subpools exist within the local shared resource pool, the index subpool statistics always follow the data subpools statistics. Index subpool statistics are also retrieved in ascending order based on the buffer size.

The final call for the series returns a GA status code in the PCB. The statistics returned are totals for all subpools in all local shared resource pools. If no VSAM buffer subpools are present, a GE status code is returned to the program.

VBESF: This function value provides the full VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 600 bytes. For each shared resource pool ID, the first call returns five 120-byte records (formatted for printing). Three of the records are heading lines and two of the records are lines of subpool statistics.

The following shows the data format:

```
      B U F F E R   H A N D L E R   S T A T I S T I C S  /  V S A M   S T A T I S T I C S   FIXOPT=X/X/X   POOLID: xxxx
BSIZ NBUFFRS HS-NBUF  RETURN-RBA  RETURN-KEY  ESDS-INSRT  KSDS-INSRT  BUFFRS-ALT  BKGRND-WRT  SYNC-POINT  ERRORS
              VSAM-GETS  SCHED-BUFR  VSAM-FOUND  VSAM-READS  USER-WRITS   VSAM-WRITS  HSRDS-SUCC  HSWRT-SUCC HSR/W-FAIL
nn1K  nnnnnn nnnnnnn  nnnnnnnnn   nnnnnnnnn   nnnnnnnnn   nnnnnnnnn   nnnnnnnnn   nnnnnnnnn   nnnnnnnnn   nnnnnn/nnnnnn
             nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn nnnnn/nnnnn
```

FIXOPT	Fixed options for this subpool. Y or N indicates whether the data buffer prefix, the index buffers, and the data buffers are fixed.
POOLID	ID of the local shared resource pool.
BSIZ	Size of the buffers in this subpool. Set to ALL for total line.[1]
NBUFFRS	Number of buffers in this subpool. Total number of buffers in the VSAM pool that appears in the ALL line.
HS-NBUF	Number of hiperspace buffers defined for this subpool.
RETURN-RBA	Number of retrieve-by-RBA calls received by the buffer handler.
RETURN-KEY	Number of retrieve-by-key calls received by the buffer handler.
ESDS-INSRT	Number of logical records inserted into ESDSs.
KSDS-INSRT	Number of logical records inserted into KSDSs.
BUFFRS-ALT	Number of logical records altered in this subpool. Delete calls that result in erasing records from a KSDS are not counted.
BKGRND-WRT	Number of times the background write function was executed by the buffer handler.
SYNC-POINT	Number of Synchronization calls received by the buffer handler.
ERRORS	Number of write error buffers currently in the subpool or the largest number of write errors in the subpool during this execution.
VSAM-GETS	Number of VSAM Get calls issued by the buffer handler.
SCHED-BUFR	Number of VSAM Scheduled-Buffer calls issued by the buffer handler.
VSAM-FOUND	Number of times VSAM found the control interval in the buffer pool.
VSAM-READS	Number of times VSAM read a control interval from external storage.
USER-WRITS	Number of VSAM writes initiated by IMS.
VSAM-WRITS	Number of VSAM writes initiated to make space in the subpool.
HSRDS-SUCC	Number of successful VSAM reads from hiperspace buffers.
HSWRT-SUCC	Number of successful VSAM writes from hiperspace buffers.
HSR/W-FAIL	Number of failed VSAM reads from hiperspace buffers/number of failed VSAM writes to hiperspace buffers. This indicates the number of times a VSAM READ/WRITE request from or to hiperspace resulted in DASD I/O.

Note: [1]For the summary totals (BSIZ=ALL), the FIXOPT and POOLID fields are replaced by a VS= field and a HS= field. The VS= field is the total size of the VSAM subpool in virtual storage. The HS= field is the total size of the VSAM subpool in hiperspace.

VBESU: This function value provides full VSAM statistics in an unformatted form. The application program I/O area must be at least 104 bytes. Twenty-five fullwords of binary data are provided for each subpool.

Word	Contents
1	A count of the number of words that follow.
2-23	The statistics provided in the same sequence as presented by the VBESF function value.
24	The POOLID provided at the time the subpool is defined.
25	The first byte contains the subpool type, and the third byte contains the following fixed options for this subpool: • X'08' = INDEX BUFFERS fixed • X'04' = DATA BUFFER PREFIX fixed • X'02' = DATA BUFFERS fixed The summary totals (word 2=ALL) for word 25 and word 26 contain the virtual and hiperspace pool sizes.

VBESS: This function value provides a summary of the VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 360 bytes. For each shared resource pool ID, the first call provides six 60-byte records (formatted for printing).

The following shows the data format:

```
VSAM DB BUFFER POOL:ID xxxx   BSIZE nnnnnnK   TYPE x   FX=X/X/X
  RRBA  nnnnnnnnnn    RKEY       nnnnnnnnnn   BFALT  nnnnnnnnnn
  NREC  nnnnnnnnnn    SYNC PT    nnnnnnnnnn   NBUFS  nnnnnnnnnn
  VRDS  nnnnnnnnnn    FOUND      nnnnnnnnnn   VWTS   nnnnnnnnnn
  HSR-S nnnnnnnnnn    HSW-S      nnnnnnnnnn   HS NBUFS nnnnnnnn
  HS-R/W-FAIL   nnnnn/nnnnn       ERRORS    nnnnnn/nnnnnn
```

POOLID	ID of the local shared resource pool.
BSIZE	Size of the buffers in this VSAM subpool.
TYPE	Indicates a data (D) subpool or an index (I) subpool.
FX	Fixed options for this subpool. Y or N indicates whether the data buffer prefix, the index buffers, and the data buffers are fixed.
RRBA	Number of retrieve-by-RBA calls received by the buffer handler.
RKEY	Number of retrieve-by-key calls received by the buffer handler.
BFALT	Number of logical records altered.
NREC	Number of new VSAM logical records created.
SYNC PT	Number of sync point requests.
NBUFS	Number of buffers in this VSAM subpool.
VRDS	Number of VSAM control interval reads.
FOUND	Number of times VSAM found the requested control interval already in the subpool.

VWTS	Number of VSAM control interval writes.
HSR-S	Number of successful VSAM reads from hiperspace buffers.
HSW-S	Number of successful VSAM writes to hiperspace buffers.
HS NBUFS	Number of VSAM hiperspace buffers defined for this subpool.
HS-R/W-FAIL	Number of failed VSAM reads from hiperspace buffers and number of failed VSAM writes to hiperspace buffers. This indicates the number of times a VSAM READ/WRITE request to or from hiperspace resulted in DASD I/O.
ERRORS	Number of permanent write errors now in the subpool or the largest number of errors in this execution.

Writing Information to the System Log: The LOG Request

An application program can write a record to the system log by issuing the `LOG` call. When you issue the `LOG` request, you specify the I/O area that contains the record you want written to the system log. You can write any information to the log that you want, and you can use different log codes to distinguish between different types of information.

Related Reading: For information about coding the `LOG` request, see the appropriate application programming reference book as described in “How to Use This Book” on page xiii.

What to Do When Your Program Terminates Abnormally

When your program terminates abnormally, you can take the following actions to simplify the task of finding and fixing the problem:

- Record as much information as possible about the circumstances under which the program terminated abnormally.
- Check for certain initialization and execution errors.

If You Have a Problem

Many installations have guidelines on what you should do if your program terminates abnormally. The suggestions given here are common installation guidelines:

- Document the error situation to help in investigating and correcting it. The following information can be helpful:
  - The program’s PSB name
  - The transaction code that the program was processing (online programs only)
  - The text of the input message being processed (online programs only)
  - The call function
  - The name of the originating logical terminal (online programs only)
  - 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 database 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 the required error information to a standard error routine. You should not use STAE or ESTAE routines in your program; IMS uses STAE or ESTAE routines to notify the control region of any abnormal termination of the application program. If you call your

own STAE or ESTAE routines, IMS may not get control if an abnormal termination occurs. For additional information about STAE or ESTAE routines, see "Use of STAE or ESTAE and SPIE in IMS Programs" on page 48.

- Online programs might want to send a message to the originating logical terminal to inform the person at the terminal that an error has occurred. Unless you are using a CCTL, your 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. (This is not applicable if you are using a CCTL.)
- You might also want to send a message to a printer so that you will have a hard-copy record of the error.
- You can send a message to the system log by issuing a `LOG` request.
- Some installations run a BMP at the end of the day to list all the errors 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. (This is not applicable if you are using a CCTL.)

Finding the Problem

If your program does not run correctly when you are testing it or when it is 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 requests) to a system problem. 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 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 do not have any initialization errors, check:

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 included?
   - Is the correct entry point specified?
3. Your JCL:
   - Is the information that described the files that contain the databases 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 and 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? Did you save the list of PCB addresses at entry? Does register 1 point to a parameter list of fullwords before issuing any DL/I calls?
- For COBOL and PL/I, are the literals you are 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, and 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.

Chapter 10. Testing a CICS Application Program

This chapter tells you what is involved in testing a CICS application program as a unit and gives you some suggestions on how to do testing. 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 ensure that the program correctly handles its input data, processing, and output data.

The amount and type of testing you do depends on the individual program. Though strict rules for testing are not available, the guidelines provided in this chapter might be helpful.

In this Chapter:

What You Need to Test a Program

When you are ready to test your program, be aware of the test procedures at your installation before you start. To start testing, you need the following three items:

- Test JCL.
- A test database. When you are testing a program, do not execute it against a production database because the program, if faulty, might damage valid data.
- Test input data. The input data that you use need not be current, but it should be valid data. You cannot 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 the situations that it might encounter.

To thoroughly test the program, try to test as many of the paths that the program can take as possible. For example:

- Test each path in the program by using 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 Your Program

You can use different tools to test a program, depending on the type of program. Table 15 summarizes the tools that are available.

Table 15. Tools You Can Use for Testing Your Program

Tool	Online (DBCTL)	Batch	BMP
Execution Diagnostic Facility (EDF)	Yes	No	No
CICS Dump Control	Yes	No	No
CICS Trace Control	Yes	Yes	No

Table 15. Tools You Can Use for Testing Your Program (continued)

Tool	Online (DBCTL)	Batch	BMP
DFSDDLT0	No	Yes	Yes
DL/I Image Capture Program	Yes	Yes	Yes

Notes:
[1] For online, command-level programs only.
[2] For call-level programs only. (For a command-level batch program, you can use DL/I image capture program first, to produce calls for DFSDDLT0.)

Using the Execution Diagnostic Facility (Command-Level Only)

You can use the Execution Diagnostic Facility (EDF) to test command-level programs online. EDF can display EXEC CICS and EXEC DLI commands in online programs; it cannot intercept DL/I calls. (To test a call-level online program, you can use the CICS dump control facility or the CICS trace facility, described in the following sections.)

With EDF you can:

- Display and modify working storage; you can change values in the DIB.
- Display and modify a command before it is executed. You can modify the value of any argument, and then execute the command.
- Modify the return codes after the execution of the command. After the command has been executed, but before control is returned to the application program, the command is intercepted to show the response and any argument values set by CICS.

You can run EDF on the same terminal as the program you are testing.

Related Reading: For more information about using EDF, see “Execution (Command-Level) Diagnostic Facility” in *CICS/ESA Application Programming Reference*.

Using CICS Dump Control

You can use the CICS dump control facility to dump virtual storage areas, CICS tables, and task-related storage areas.

For more information about using the CICS dump control facility, see the CICS application programming reference manual that applies to your version of CICS.

Using CICS Trace Control

You can use the trace control facility to help debug and monitor your online programs in the DBCTL environment. You can use trace control requests to record entries in a trace table. The trace table can be located either in virtual storage or on auxiliary storage. If it is in virtual storage, you can gain access to it by investigating a dump; if it is on auxiliary storage, you can print the trace table. For more information about the control statements you can use to produce trace entries, see the information about trace control in the application programming reference manual that applies to your version of CICS.

Using the DL/I Test Program (DFSDDLT0)

See “Testing DL/I Call Sequences (DFSDDLT0)” on page 135 for a description of DFSDDLT0. DFSDDLT0 can be used for testing batch or BMP programs.

Tracing DL/I Calls with Image Capture

DL/I image capture program (DFSDLTR0) is a trace program that can trace and record DL/I calls issued by batch, BMP, and online (DBCTL environment) programs. You can also use the image capture program with command-level programs, and you can produce calls for use as input to DFSDDLT0. You can use the image capture program to:

Test your program

If the image capture program detects an error in a call it traces, it reproduces as much of the call as possible, although it cannot document where the error occurred, and cannot always reproduce the full SSA.

Produce input for DFSDDLT0 (DL/I test program)

You can use the output produced by the image capture program as input to DFSDDLT0. The image capture program produces status statements, comment statements, call statements, and compare statements for DFSDDLT0. For example, you can use the image capture program with a command-level program, to produce calls for DFSDDLT0.

Debug your program

When your program terminates abnormally, you can rerun the program using the image capture program. The image capture program can then reproduce and document the conditions that led to the program failure. You can use the information in the report produced by the image capture program to find and fix the problem.

Using Image Capture with DFSDDLT0

The image capture program produces the following control statements that you can use as input to DFSDDLT0:

Status statements

When you invoke the image capture program, it produces the status statement. The status statement it produces:

- Sets print options so that DFSDDLT0 prints all call trace comments, all DL/I calls, and the results of all comparisons.
- Determines the new relative PCB number each time a PCB change occurs while the application program is executing.

Comments statement

The image capture program also produces a comments statement when you invoke it. The comments statements give:

- The time and date IMS started the trace
- The name of the PSB being traced

The image capture program also produces a comments statement preceding any call in which IMS finds an error.

Call statements

The image capture program produces a call statement for each DL/I call or EXEC DLI command the application program issues. It also generates a `CHKP` call when it starts the trace and after each commit point or `CHKP` request.

Compare statements

If you specify COMP on the DLITRACE control statement, the image capture program produces data and PCB comparison statements.

Running Image Capture Online

This section applies to a CICS (or CCTL) online program (running in the DBCTL environment only) or BMP programs (DBCTL environment). When you run the image capture program online, the trace output goes to the IMS log data set. To run the image capture program online, you issue the IMS `TRACE` command from the MVS console.

If you trace a BMP and you want to use the trace results with DFSDDLT0, the BMP must have exclusive write access to the databases it processes. If the application program does not have exclusive access, the results of DFSDDLT0 may differ from the results of the application program.

The following diagram shows `TRACE` command format:

```
                  ┌─ON──┐
►►──/──TRACE──SET──┴─OFF─┴──PSB──psbname──┬─────────────────┬──────────────────────►◄
                                          │ ┌─NOCOMP──┐     │
                                          └─┴─COMP────┴─────┘
```

SET ON|OFF
: Turns the trace on or off.

PSB psbname
: Specifies the name of the PSB you want to trace. You can trace more than one PSB at the same time, by issuing a separate `TRACE` command for each PSB.

COMP|NOCOMP
: Specifies whether you want the image capture program to produce data and PCB compare statements to be used with DFSDDLT0.

Running Image Capture in Batch

This section applies to batch programs. To run the image capture program as a batch job, you use the DLITRACE control statement in the DFSVSAMP DD data set. In the DLITRACE control statement you specify:

- Whether you want to trace all of the DL/I calls the program issues or trace only a certain group of calls.
- Whether you want the trace output to go to:
  - A sequential data set that you specify
  - The IMS log data set
  - Both sequential and IMS log data sets

The format of the DLITRACE control statement is:

```
►►──DLITRACE──┬─────────────────┬──┬────────────────────────────────┬───────────►
              │          ┌─NO──┐│  │                ┌─DFSTROUT──┐   │
              └─LOG──=───┴─YES─┴┘  └─,──DDNAME──=───┴─anyname───┴───┘

►──┬──────────────────┬─────────────────────────────────────────────────────────►◄
   │           ┌─0─┐  │
   └─,──START──=─┴─n─┴──┘

►►──┬──────────────────────────┬──┬──────────────────┬──────────────────────────►◄
    │            ┌─7FFFFFFF─┐  │  │  ┌─,──NOCOMP──┐  │
    └─,──STOP──=─┴─n────────┴──┘  └──┴─,──COMP────┴──┘
```

DLITRACE
Invokes the trace. If this is the only parameter you specify, IMS uses default values for the remaining parameters.

LOG = YES|NO
Specifies whether you want IMS to route trace output to the IMS log.

DDNAME = anyname|DFSTROUT
Specifies the ddname of the sequential data set to which you want trace output sent. The default is DFSTROUT; but if you specify LOG = YES, and you do not supply a value for the DDNAME parameter, IMS does not try to open the data set defined by DFSTROUT.

START = n|0
Gives the number of the first Dl/I call in the program that you want traced. The value is a one- to eight-character hexadecimal number. If you do not supply a value, the trace starts with the first DL/I call in the program.

STOP = n|7FFFFFFF
Specifies the number of the last DL/I call in the program that you want traced. The value is a one- to eight-character hexadecimal number.

COMP|NOCOMP
Specifies whether you want the image capture program to produce data and PCB comparison statements to be used as input to the DFSDDLT0 test program. NOCOMP is the default.

Notes:

1. DLITRACE must begin in column 1; the remainder of the parameters are nonpositional.
2. Each parameter may be used only once in the control statement.
3. Only one trace control statement is allowed for a program.

Example of DLITRACE

The example below shows a DLITRACE control statement that:

- Traces the first 14 DL/I calls or commands that the program issues
- Sends the output to the IMS log data set
- Produces data and PCB comparison statements for DFSDDLT0

```
//DFSVSAMP DD *
DLITRACE LOG=YES,STOP=14,COMP
/*
```

Special JCL Requirements

The following are special JCL requirements:

//IEFRDER DD
If you want log data set output, this DD statement is required to define the IMS log data set.

//DFSTROUT DD|anyname
If you want sequential data set output, this DD statement is required to define that data set. If you want to specify an alternate DDNAME (anyname), it must be specified using the DDNAME parameter on the DLITRACE control statement.

The DCB parameters on the JCL statement are not required. The data set characteristics are:

- RECFM=F

- LRECL=80

Notes on Using Image Capture

- If the program being traced issues CHKP and XRST calls, the checkpoint and restart information may not be directly reproducible when you use the trace output with the DFSDDLT0.
- When you run DFSDDLT0 in an IMS DL/I or DBB batch region with trace output, the results are the same as the application program's results provided the database has not been altered.

Retrieving Image Capture Data from the Log Data Set

If the trace output is sent to the IMS log data set, you can retrieve it by using utility DFSERA10 and a DL/I call trace exit routine, DFSERA50. DFSERA50 deblocks, formats, and numbers the image capture program records to be retrieved. To use DFSERA50, you must insert a DD statement defining a sequential output data set in the DFSERA10 input stream. The default ddname for this DD statement is TRCPUNCH. The card must specify BLKSIZE=80.

Examples: You can use the following examples of DFSERA10 input control statements in the SYSIN data set to retrieve the image capture program data from the log data set:

- **Print all image capture program records:**

```
Column 1        Column 10
OPTION          PRINT OFFSET=5,VALUE=5F,FLDTYP=X
```

- **Print selected image capture program records by PSB name:**

```
Column 1        Column 10
OPTION          PRINT OFFSET=5,VALUE=5F,COND=M
OPTION          PRINT OFFSET=25,VLDTYP=C,FLDLEN=8,
                VALUE=psbname, COND=E
```

- **Format image capture program records (in a format that can be used as input to DFSDDLT0):**

```
Column 1        Column 10
OPTION          PRINT OFFSET=5,VALUE=5F,COND=M
OPTION          PRINT EXITR=DFSERA50,OFFSET=25,FLDTYP=C
                VALUE=psbname,FLDLEN=8,DDNAME=OUTDDN,COND=E
```

The DDNAME= parameter is used to name the DD statement used by DFSERA50. The data set defined on the OUTDDN DD statement is used instead of the default TRCPUNCH DD statement. For this example, the DD appears as:

```
//OUTDDN DD ...,DCB=(BLKSIZE=80),...
```

Requests for Monitoring and Debugging

You can use the following two requests to help you in debugging your program:

- The statistics (STAT) request retrieves database statistics. STAT can be issued from both call- and command-level programs. See “Retrieving Database Statistics: The STAT Call” on page 141 for a description of the STAT request.
- The log (LOG) request makes it possible for the application program to write a record on the system log. You can issue LOG as a command or call in a batch program; in this case, the record is written to the IMS log. You can issue LOG as a call or command in an online program in the DBCTL environment; in this case, the record is written to the DBCTL log. See “Writing Information to the System Log: The LOG Request” on page 151 for a description of the LOG request.

What to Do When Your Program Terminates Abnormally

Whenever your program terminates abnormally, you can take some actions to simplify the task of finding and fixing the problem. 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.

If You Have a Problem

Many installations have guidelines on what you should do if your program terminates abnormally. The suggestions given here are some common installation guidelines:

- 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 (online programs only)
  - The text of the input screen being processed (online programs only)
  - The call function
  - The terminal ID (online programs only)
  - The contents of the PCB or the DIB
  - The contents of the I/O area when the problem occurred
  - If a database request was executing, the SSAs or SEGMENT and WHERE options, if any, the request used
  - The date and time of day
- When your program encounters an error, it can pass all the required error information to a standard error routine.
- An online program might also want to send a message to the master terminal destination (CSMT) and application terminal operator, giving information about the program's termination.
- You can send a message to the system log by issuing a `LOG` request.

Finding the Problem

If your program does not run correctly when you are testing it or when it is 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 requests) to a system problem. 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 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 do not have any initialization errors, check the following in your program:

Abnormal Program Termination

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 included?
   - Is the correct entry point specified (for batch programs only)?
3. Your JCL:
   - Is the information that described the files that contain the databases correct? If not, check with your DBA.
   - Have you included the DL/I parameter statement in the correct format (for batch programs only)?
   - 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 and your program (for batch programs only)?
4. Your call-level 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? Did you save the list of PCB addresses at entry? Does register 1 point to a parameter list of fullwords before issuing any DL/I calls?
   - For COBOL and PL/I, are the literals you are 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, and 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.
5. Your command-level program:
   - Did you use the `FROM` option with your `ISRT` or `REPL` command? If not, data will not be transferred to the database.
   - Check translator messages for errors.

Chapter 11. Testing an ODBA Application Program

This chapter tells you what is involved in testing an ODBA application program as a unit and gives you some suggestions on how to do testing. 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 ensure that the program correctly handles its input data, processing, and output data. The amount and type of testing you do depends on the individual program. Though strict rules for testing are not available, the guidelines provided in this chapter might be helpful.

When you are ready to test your program, be aware of the test procedures at your installation before you start. To start testing, you need the following three items:

- Test JCL.
- A test database. When you are testing a program, do not execute it against a production database because the program, if faulty, might damage valid data.
- Test input data. The input data that you use need not be current, but it should be valid data. You cannot 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 the situations that it might encounter. To thoroughly test the program, try to test as many of the paths that the program can take as possible. For example:

Test each path in the program by using 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.

Tool	Online (IMS DB)	Batch	BMP
DFSDDLT0	No	Yes[1]	Yes
DL/I Image Capture Program	Yes	Yes	Yes
Note: 1. For call-level programs only. (For a command-level batch program, you can use DL/I image capture program first, to produce calls for DFSDDLT0).			

Using the DL/I Test Program (DFSDDLT0)

See "Testing DL/I Call Sequences (DFSDDLT0)" on page 135 for a description of DFSDDLT0. DFSDDLT0 can be used for testing batch or BMP programs.

Tracing DL/I Calls with Image Capture

The DL/I image capture program (DFSDLTR0) is a trace program that can trace and record DL/I calls issued by batch, BMP, and online (IMS DB environment) programs. You can produce calls for use as input to DFSDDLT0. You can use the image capture program to:

- Test your program

  If the image capture program detects an error in a call it traces, it reproduces as much of the call as possible, although it cannot document where the error occurred, and cannot always reproduce the full SSA.

- Produce input for DFSDDLT0 (DL/I test program)

  You can use the output produced by the image capture program as input to DFSDDLT0. The image capture program produces status statements, comment statements, call statements, and compare statements for DFSDDLT0. For example, you can use the image capture program with a ODBA application, to produce calls for DFSDDLT0.
- Debug your program

  When your program terminates abnormally, you can rerun the program using the image capture program. The image capture program can then reproduce and document the conditions that led to the program failure. You can use the information in the report produced by the image capture program to find and fix the problem.

Using Image Capture with DFSDDLT0

The image capture program produces the following control statements that you can use as input to DFSDDLT0:

- Status statements

  When you invoke the image capture program, it produces the status statement. The status statement it produces:
  - Sets print options so that DFSDDLT0 prints all call trace comments, all DL/I calls, and the results of all comparisons
  - Determines the new relative PCB number each time a PCB change occurs while the application program is running
- Comments statement

  The image capture program also produces a comments statement when you run it. The comments statements give:
  - The time and date IMS started the trace
  - The name of the PSB being traced

  The image capture program also produces a comments statement preceding any call in which IMS finds an error.
- Call statements

  The image capture program produces a call statement for each DL/I call.
- Compare statements

  If you specify COMP on the DLITRACE control statement, the image capture program produces data and PCB comparison statements.

Running Image Capture Online

This section applies to a CICS (or CCTL) online program (running in the IMS DB environment only) or BMP programs (IMS DB environment). When you run the image capture program online, the trace output goes to the IMS log data set. To run the image capture program online, you issue the IMS TRACE command from the MVS console. If you trace a BMP and you want to use the trace results with DFSDDLT0, the BMP must have exclusive write access to the databases it processes. If the application program does not have exclusive access, the results of DFSDDLT0 may differ from the results of the application program. The following diagram shows TRACE command format:

```
              ┌─ON──┐
►►──/──TRACE──SET──┴─OFF─┴───PSB──psbname──┬───────────────┬──────────────────►◄
                                            │  ┌─NOCOMP─┐  │
                                            └──┴─COMP───┴──┘
```

SET ON|OFF
Turns the trace on or off.

PSB *psbname*
Specifies the name of the PSB you want to trace. You can trace more than one PSB at the same time by issuing a separate TRACE command for each PSB.

COMP|NOCOMP
Specifies whether you want the image capture program to produce data and PCB compare statements to be used with DFSDDLT0.

Retrieving Image Capture Data from the Log Data Set

If the trace output is sent to the IMS log data set, you can retrieve it by using utility DFSERA10 and a DL/I call trace exit routine, DFSERA50. DFSERA50 deblocks, formats, and numbers the image capture program records to be retrieved. To use DFSERA50, you must insert a DD statement defining a sequential output data set in the DFSERA10 input stream. The default ddname for this DD statement is TRCPUNCH. The card must specify BLKSIZE=80.

Examples: You can use the following examples of DFSERA10 input control statements in the SYSIN data set to retrieve the image capture program data from the log data set:

- Print all image capture program records:

```
Column 1      Column 10
OPTION        PRINT OFFSET=5,VALUE=5F,FLDTYP=X
```

- Print selected image capture program records by PSB name:

```
Column 1      Column 10
OPTION        PRINT OFFSET=5,VALUE=5F,COND=M
OPTION        PRINT OFFSET=25,VLDTYP=C,FLDLEN=8,
                  VALUE=psbname, COND=E
```

- Format image capture program records (in a format that can be used as input to DFSDDLT0):

```
Column 1      Column 10
OPTION        PRINT OFFSET=5,VALUE=5F,COND=M
OPTION        PRINT EXITR=DFSERA50,OFFSET=25,FLDTYP=C
              VALUE=psbname,FLDLEN=8,DDNAME=OUTDDN,COND=E
```

  The DDNAME= parameter is used to name the DD statement used by DFSERA50. The data set defined on the OUTDDN DD statement is used instead of the default TRCPUNCH DD statement. For this example, the DD appears as:

```
//OUTDDN DD ...,DCB=(BLKSIZE=80),...
```

Requests for Monitoring and Debugging

You can use the following two requests to help you in debugging your program:

- The statistics (STAT) request retrieves database statistics. STAT can be issued from both call- and command-level programs. See “Retrieving Database Statistics: The STAT Call” on page 141 for a description of the STAT request.

- The log (LOG) request makes it possible for the application program to write a record on the system log. You can issue LOG as a command or call in a batch program; in this case, the record is written to the IMS log. You can issue LOG as a call or command in an online program in the IMS DB environment; in this case, the record is written to the IMS DB log. See "Writing Information to the System Log: The LOG Request" on page 151 for a description of the LOG request.

What to Do When Your Program Terminates Abnormally

Whenever your program terminates abnormally, you can take some actions to simplify the task of finding and fixing the problem.

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.

If You Have a Problem

Many installations have guidelines on what you should do if your program terminates abnormally. The suggestions given here are some common installation guidelines:

- 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 call function
  - The terminal ID (online programs only)
  - The contents of the PCB or the DIB
  - The contents of the I/O area when the problem occurred
  - If a database request was executing, the SSAs or SEGMENT and WHERE options, if any, the request used
  - The date and time of day
- When your program encounters an error, it can pass all the required error information to a standard error routine.
- You can send a message to the system log by issuing a LOG request.

Finding the Problem

If your program does not run correctly when you are testing it or when it is running, 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 requests) to a system problem. 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 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.

Running Errors

If you do not 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 included?
3. Your JCL. Is the information that described the files that contain the databases correct? If not, check with your DBA.

Chapter 12. Documenting an Application Program

This chapter provides guidelines for program documentation. The purposes for documenting an application program are described.

In this Chapter:

Many installations establish standards for program documentation; make sure you are aware of any standards at your installation.

Documentation for Other Programmers

Documenting a program is not something you do at the end of the project; 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 must work with your program.

The reason you record this information is so that people who maintain your program know why you chose certain commands, options, call structures, and command codes. For example, if the DBA was considering reorganizing the database 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. You can use the DB/DC Data Dictionary, or its successor, DataAtlas (a part of the IBM VisualGen Team Suite), for this purpose.

Related Reading: For information on how to use these products to document a data processing environment: the application system, the programs, the programs' modules, and the IMS system, refer to *OS/VS DB/DC Data Dictionary Applications Guide*, *Introducing VisualGen*, or *VisualGen: Running Applications on MVS*.

Information you can 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

All this information relates to structuring and coding the program. In addition, you should include the information described in "Documentation for Users" on page 170 with the documentation for programmers.

Documentation for Other Programmers

Again, the amount of information you include and the form in which you document it depend on you and your installation. These documentation guidelines are provided as suggestions.

Documentation for Users

All the information listed in the previous section 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 the following information for users of your program:

- What the user needs in order to use the program
  - For online programs, is there a password?
  - For batch programs, what is the required JCL?
- The input that users need to supply to the program
  - For example, for an MPP, what is the MOD name that the user enters to initially format the screen?
  - Or for a CICS online program, what is the CICS transaction code a user must enter? What terminal input is expected?
  - For a batch program, is the input in the form of cards, tape, or a disk data set? Is it output from a previous job?
- The content and form of the program's output
  - If it is 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 must make decisions, explain 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. What should be done if the task or program abends? Should the program be restarted, or does the database need recovery? Although you may not be responsible for providing this kind of information, you should provide any information that is unique to your application to the person who is responsible for this information.

Appendix. IMS Spool API Overall Design

The IMS Spool API is an expansion of the IMS application program interface that allows applications to interface directly to JES and create print data sets on the JES spool. These print data sets can then be made available to print managers and spool servers to serve the needs of the application.

This appendix describes the design of the Spool API and how an application program uses it. Considerations for coding an application program are included.

Related Reading: For more information about the Spool API, see *IMS/ESA Application Programming: Transaction Manager* and *IMS/ESA Application Programming: Database Manager*.

Design Concept

The Spool API design provides the application program with the ability to create print data sets on the JES spool using the standard DL/I call interface. The functions provided are:

Definition of the data set output characteristics

Allocation of the data set

Insertion of lines of print into the data set

Closing and deallocation of the data set

Backout of uncommitted data within the limits of the JES interface

Assistance in controlling an in-doubt print data set

The Spool API support uses existing DL/I calls to provide data set allocation information and to place data into the print data set. These calls are:

- The `CHNG` call. This call is expanded so print data set characteristics can be specified for the print data set that will be allocated. The process uses the alternate PCB as the interface block associated with the print data set.
- The `ISRT` call. This call is expanded to perform dynamic allocation of the print data set on the first insert, and to write data to the data set. The data set is considered in-doubt until the unit of work (UOW) terminates. If possible, the sync point process deletes all in-doubt data sets for abending units of work and closes and deallocates data sets for normally terminating units of work.
- The `SETO` call. This is a call, `SETO` (Set Options), introduced by this support. Use this call to create dynamic output text units to be used with the subsequent `CHNG` call. If the same output descriptor is used for many print data sets, the overhead can be reduced by using the `SETO` call to prebuild the text units necessary for the dynamic output process.

Related Reading: The use of the SETO call is covered in more detail in *IMS/ESA Application Programming: Transaction Manager*.

Start Up

Application programs can send data to the JES spool data sets using the same method that is used to send output to an alternate terminal. Use the DL/I call to change the output destination to a JES spool data set. Use the DL/I `ISRT` or `PURG` call to insert a message.

Design Concept

The options list parameter on the CHNG and SETO calls contains the data set printer processing options. These options direct the output to the appropriate Spool API data set. These options are validated for the DL/I call by the MVS/ESA Scheduler JCL Facility (SJF). If the options are invalid, error codes are returned to the application. To receive the error information, the application program specifies a feedback area in the CHNG or SETO DL/I call parameter list. If the feedback area is present, information about the options list error is returned directly to the application.

Performance Considerations

The Spool API interface uses MVS services within an IMS application while minimizing the performance impact of the MVS services on the other IMS transactions and services. For this reason, the spool API support places the print data directly on the JES spool at insert time instead of using the IMS message queue for intermediate storage. The processing of Spool API requests is performed under the TCB of the dependent region to ensure maximum usage of N-way processors. This design reduces the error recovery and JES job orientation problems.

JES Initiator Considerations

Because the dependent regions are normally long-running jobs, some of the initiator or job specifications might must be changed if the dependent region is using the Spool API. You might need to limit the amount of JES spool space used by the dependent region to contain the dynamic allocation and deallocation messages. For example, you can use the JOB statement MSGLEVEL to eliminate the dynamic allocation messages from the job log for the dependent region. You might be able to eliminate these messages for dependent regions executing as MVS started tasks.

Another initiator consideration is the use of the JES job journal for the dependent region. If the jobstep has a journal associated with it, the information for MVS checkpoint restart is recorded in the journal. Because IMS dependent regions cannot use MVS checkpoint restart, specify JOURNAL=NO for the JES2 initiator procedure and the JES3 class associated with the dependent regions execution class. You can also specify the JOURNAL= parameter on the JES3 //*MAIN statement for dependent regions executing as jobs.

Application Managed Text Units

The application can manage the dynamic descriptor text units instead of IMS. If the application manages the text units, overhead for parsing and text unit build can be reduced. Use the SETO call to have IMS build dynamic descriptor text units. After they are built, these text units can be used with subsequent CHNG calls to define the print characteristics for a data set.

To reduce overhead by managing the text units, the text units should be used with several change calls. An example of this is a wait-for-input (WFI) transaction. The same data set attributes can be used for all print data sets. For the first message processed, the application uses the SETO call to build the text units for dynamic descriptors and a subsequent CHNG call with the TXTU= parameter referencing the prebuilt text units. For all subsequent messages, only a CHNG call using the prebuilt text units is necessary.

Be aware of the following: No testing has been done to determine the amount of overhead that might be saved using prebuilt text units.

BSAM I/O Area

The I/O area for spool messages can be very large. It is not uncommon for the area to be 32 KB in length. To reduce the overhead incurred with moving large buffers, IMS attempts to write to the spool data set from the application's I/O area. BSAM does not support I/O areas in 31-bit storage for SYSOUT files. If IMS finds that the application's I/O area is in 31-bit storage:

- A work area is obtained from 24-bit storage.
- The application's I/O area is moved to the work area.
- The spool data set is written from the work area.

If the application's I/O area can easily be placed in 24-bit storage, the need to move the I/O area can be avoided and possible performance improvements achieved.

Be aware of the following: No testing has been done to determine the amount of performance improvement possible.

Since a record can be written by BSAM directly from the application's I/O area, the area must be in the format expected by BSAM. The format must contain:

- Variable length records
- A Block Descriptor Word (BDW)
- A Record Descriptor Word (RDW)

Related Reading: For more information on the formats of the BDW and RDW, see *MVS/XA Data Administration Guide*. The format of the I/O area is described in more familiar IMS terms in *IMS/ESA Application Programming: Transaction Manager*.

Application Coding Considerations

Your application can send data to a JES Spool or Print server using a print data set. This section describes this process and includes options for message integrity and recovering data when failures occur.

Print Data Formats

The IMS Spool API attempts to provide a transparent interface for the application to insert data to the JES spool. The data can be in line, page, IPDS, AFPDS, or any format that can be handled by a JES Spool or Print server that processes the print data set. The IMS Spool API does not translate or otherwise modify the data inserted to the JES spool.

Message Integrity Options

The IMS Spool API provides support for message integrity. This is necessary because IMS cannot properly control the disposition of a print data set when:

- IMS abnormal termination does not execute because of a hardware or software problem.
- A dynamic deallocation error exists for a print data set.
- Logic errors are in the IMS code.

In these conditions, IMS might not be able to stop the JES subsystem from printing partial print data sets. Also, the JES subsystems do not support a two-phase sync point.

Print Disposition

The most common applications using Advanced Function Printing (AFP) are TSO users and batch jobs. If any of these applications are creating print data sets when a failure occurs, the partial print data sets will probably print and be handled in a manual fashion. Many IMS applications creating print data sets can manage partial print data sets in the same manner. For those applications that need more control over the automatic printing by JES of partial print data sets, the IMS Spool API provides the following integrity options. However, these options alone might not guarantee the proper disposition of partial print data sets. These options are the **b** variable following the IAFP keyword used with the `CHNG` call.

b=0

Indicates no data set protection

This is probably the most common option. When this option is selected, IMS does not do any special handling during allocation or deallocation of the print data set. If this option is selected, and any condition occurs that prevents IMS from properly disposing the print data set, the partial data set probably prints and must be controlled manually.

b=1

Indicates SYSOUT HOLD protection

This option ensures that a partial print data set is not released for printing without a JES operator taking direct action. When the data set is allocated, the allocation request indicates to JES that this print data set be placed in SYSOUT HOLD status. The SYSOUT HOLD status is maintained for this data set if IMS cannot deallocate the data set for any reason. Because the print data set is in HOLD status, a JES operator must identify the partial data set and issue the JES commands to delete or print this data set.

If the print data set cannot be deleted or printed:

- Message DFS0012I is issued when a print data set cannot be deallocated.
- Message DFS0014I is issued during IMS emergency restart when an in-doubt print data set is found. The message provides information to help the JES operator find the proper print data set and effect the proper print disposition.

  Some of the information includes:
  - JOBNAME
  - DSNAME
  - DDNAME
  - A recommendation on what IMS believes to be the proper disposition for the data set (for example, printing or deleting).

By using the Spool Display and Search Facility (SDSF), you can display the held data sets, identify the in-doubt print data set by DDNAME and DSNAME, and issue the proper JES command to either delete or release the print data set.

b=2

Indicates a nonselectable destination

This option prevents the automatic printing of partial print data sets. The IMS Spool API function requests a remote destination of IMSTEMP for the data set when the data set is allocated. The JES system must have a remote destination of IMSTEMP defined so that JES does not attempt to print any data sets that are sent to the destination.

If **b=2**, the name of the remote destination for the print data set must be specified in the destination name field of the call parameter list when the `CHNG` call is issued. When IMS deallocates the data set at sync point, and the data set prints, IMS requests that the data set be transferred to the requested final remote destination.

If the remote destination is not defined to the JES system, a dynamic allocation failure occurs. Because this remote destination is defined as nonselectable, and if IMS is unable to deallocate the print data set and control its proper disposition, the print data set remains associated with remote destination IMSTEMP when deallocated by MVS.

When an deallocation error occurs, message DFS0012I is issued to provide details of the deallocation error and help identify the print data set that requires operator action. When partial print data sets are left on this special remote destination, the JES operator can display all the print data sets associated with this JES destination to locate the data set that requires action. The **b=2** option simplifies the operator's task of locating partial print data sets.

Message Options

The third option on the IAPF keyword controls informational messages issued by the IMS Spool API support. These messages inform the JES operator of in-doubt data sets that need action.

c=0
Indicates that no DFS0012I or DFS0014I messages are issued for the print data set. You can specify **c=0** only if **b=0** is specified.

c=m
Indicates that DFS0012I and DFS0014I messages are issued if necessary. You can specify **c=m** or if **b=1** or if **b=2**, it is the default.

Option **c** does not affect issuing message DFS0013E.

IMS Emergency Restart: When IMS emergency restart is performed, DFS0014I messages might be issued if IMS finds that the proper disposition of a print data set is in-doubt, as a result of the restart. This message is only issued if the message option for the print data set was requested or **c=m** on the IAFP variable. When a DFS0014I message is received, a JES operator might need to find and properly dispose of the print data set. The DFS0014I message provides a recommended disposition (that is, deletion or printing).

Destination Name (LTERM) usage

The standard `CHNG` call parameter list contains a destination name field. For traditional message calls, this field contains the LTERM or transaction code that becomes the destination of messages sent via this alternate PCB. When `ISRT` calls are issued against the PCB, the data is sent to the LTERM or transaction.

However, the destination name field has no meaning to the IMS Spool API function unless **b=2** is specified following the IAFP keyword.

When **b=2** is specified:

- The name must be a valid remote destination supported by the JES system that receives the print data sets.
- If the name is not a valid remote destination, an error occurs during dynamic deallocation.

If any option other than **2** is selected, the name is not used by IMS.

The LTERM name appears in error messages and log records. Use a name that identifies the routine creating the print data set. This information can aid in debugging application program errors.

Bibliography

This bibliography includes all the publications cited in this book, including the publications in the IMS library.

BTS Program Reference/Operations Manual, SH20-5523

CICS/ESA Application Programming Guide, SC33-0675

CICS/ESA Application Programming Reference, SC33-0676

CICS/ESA Database Control Guide, SC33-0660

CICS/ESA General Information, GC33-0155

CICS/ESA Messages and Codes, SC33-0672

CICS/ESA Recovery and Restart Guide, SC33-0658

Common Programming Interface Communications Reference, SC26-4399

IBM DATABASE 2 Version 5 Application Programming and SQL Guide, SC26-8958

Introducing VisualGen, GH23-6570-01

MVS/XA Data Administration Guide, GC26-4140

OS PL/I Version 2 Programming Guide, SC26-4307

OS/VS DB/DC Data Dictionary Applications Guide, SH20-9190

SNA LU 6.2 Reference: Peer Protocols, SN61-0102

SNA Transaction Programmer's Reference Manual for LU Type 6.2, GC30-3084

VisualGen: Running Application on MVS, SH23-6550-00

IMS/ESA Version 6 Library

SC26-8725	ADB	Administration Guide: Database Manager
SC26-8730	AS	Administration Guide: System
SC26-8731	ATM	Administration Guide: Transaction Manager
SC26-8727	APDB	Application Programming: Database Manager
SC26-8728	APDG	Application Programming: Design Guide
SC26-8726	APCICS	Application Programming: EXEC DLI Commands for CICS and IMS
SC26-8729	APTM	Application Programming: Transaction Manager
SC26-8732	CG	Customization Guide
SC26-9517	CQS	Common Queue Server Reference
SC26-8733	DBRC	Database Recovery Control Guide and Reference
LY37-3731	DGR	Diagnosis Guide and Reference
LY37-3732	FAST	Failure Analysis Structure Tables (FAST) for Dump Analysis
GC26-8736	IIV	Installation Volume 1: Installation and Verification
GC26-8737	ISDT	Installation Volume 2: System Definition and Tailoring
SC26-8740	MIG	Master Index and Glossary
GC26-8739	MC	Messages and Codes
SC26-8743	OTMA	Open Transaction Manager Access Guide
SC26-8741	OG	Operations Guide
SC26-8742	OR	Operator's Reference
GC26-8744	RPG	Release Planning Guide
SC26-8767	SOP	Sample Operating Procedures
SC26-8769	URDB	Utilities Reference: Database Manager
SC26-8770	URS	Utilities Reference: System
SC26-8771	URTM	Utilities Reference: Transaction Manager

	Supplementary Publications	
GC26-8738	LPS	Licensed Program Specifications
SC26-8766	SOC	Summary of Operator Commmands

	Online Softcopy Publications	
LK3T-2326	CDROM	IMS/ESA Version 6 Softcopy Library
SK2T-0730	CDROM	IBM Online Library: Transaction Processing and Data
SK2T-0710	CDROM	MVS Collection
SK2T-6700	CDROM	OS/390 Collection

Index

Program Number: 5655-158

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-8728-05