CA IDMS™

Navigational DML Programming Guide

Release 18.5.00, 2nd Edition

This Documentation, which includes embedded help systems and electronically distributed materials, (hereinafter referred to as the “Documentation”) is for your informational purposes only and is subject to change or withdrawal by CA at any time. This Documentation is proprietary information of CA and may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA.

If you are a licensed user of the software product(s) addressed in the Documentation, you may print or otherwise make available a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print or otherwise make available copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and such license agreement is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with “Restricted Rights.” Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2013 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

CA Technologies Product References

This document references the following CA Technologies products:

- CA IDMS™/DB
- CA IDMS™/DC Transaction Server Option (CA IDMS/DC)
- CA IDMS™ Database Universal Communications Facility Option (CA IDMS UCF)
- CA IDMS™ Distributed Database Server Option (CA IDMS DDS)
- CA ADS™ for CA IDMS™ (CA ADS)
- CA IDMS™ Online Query (CA IDMS OLQ)

Contact CA Technologies

Contact CA Support

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

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

Providing Feedback About Product Documentation

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

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

Documentation Changes

The following documentation updates were made for the 18.5.00, 2nd Edition release of this documentation:

- Error Handling (see page 97)—The description of the 2 IDMS-STATUS flavors was added.
- About the 10.2 Services Batch Interface (see page 355)—Added information about IDMSIN01 functions and the IDMS-STATUS routine.

The following documentation updates were made for the 18.5.00 release of this documentation:

- READY Statement (see page 101)—The description of the FORCE option was added.
- Area Usage Modes (see page 181)—The Default Usage Modes subsection was updated with information on the FORCE option.

Contents

Chapter 1: Introduction 11

What This Manual Is About 11

Chapter 2: The CA IDMS Environment 13

What Is CA IDMS? 13
Accessing CA IDMS Databases 14
Integrated Data Dictionary 15

Chapter 3: Database Concepts 17

Types, Occurrences, and Relationships 17
    Types and Occurrences 17
    Relationships 18
Records 19
Sets 20
    Set Mode 23
    Set Linkage for Chain Sets 23
    Set Order 24
    Index Sets 29
    Set Membership Options 31
Representing Relationships as Sets 32
    Hierarchies 32
    Networks 38
Areas 43
Logical Database Description 44
    Schemas 44
    Summary of Logical Structures 46
Physical Structures 47
    Physical Database Description 47
    Segments 47
    Database Keys (Db-key) 49
    Record Structure 50
    Set Linkage 55
    Location Mode 56
Data Structure Diagrams 59
    Representing Records Graphically 59
    Representing Sets Graphically 61

Chapter 4: Introduction to Navigational Programming 65

Housekeeping Functions 65
Establishing a Database Session 65
Binding Records 66
Readying Areas 66
Terminating a Database Session 67
Function Execution Sequence 68
Retrieving Data 68
Currency 69
Retrieval Functions 70
Updating Data 84
STORE 84
CONNECT 85
MODIFY 86
ERASE and DISCONNECT 87
Testing Set Membership 90

Chapter 5: Writing a Navigational DML Program 93

Navigational Data Manipulation Language (DML) 93
Common Considerations 94
Identifying the Operating Mode 95
Identifying the Subschema 95
Including Data Descriptions 96
IDMS Communications Block 97
Error Handling 97
Housekeeping Statements 99
BIND RUN-UNIT Statement 99
BIND RECORD Statement 100
READY Statement 101
Termination Statements 101
Subschema Considerations 102
Copying Record Definitions and Their Synonyms 105

Chapter 6: Navigational DML Programming Techniques 109

DB-Keys and Page Information 110
Run Unit Currencies 112
Use and Updating of Currency by DML Verbs 113
Retrieving Records 115
Accessing CALC Records 116
Walking a Set 117

Accessing a Sorted Set 119
Performing an Area Sweep 123
Accessing Owner Records 126
Accessing a Record by Its db-key 128
Accessing Indexed Records 131
Saving db-key, Page Information, and Bind Addresses 134
Saving a db-key 135
Saving Page Information 139
Saving a Record's BIND Address 140
Checking for Set Membership 140
Using the IF EMPTY Statement 141
Using the IF MEMBER Statement 142
Updating the Database 145
Storing Records 145
Modifying Records 148
Erasing Records 150
Connecting Records to a Set 155
Disconnecting Records from a Set 156
Accessing Bill-of-Materials Structures 159
Storing a Bill-of-Materials Structure 160
Retrieving a Bill-of-Materials Structure 161
Locking Records 166
Implicit Record Locks 168
Collecting Database Statistics 171

Chapter 7: Run Units, Locks, and Database Transactions 173

Run Units 173
Sharing Run Units Between Programs 174
Record Locks 175
Area Locks 180
Area Usage Modes 181
Database Transactions 183
Sharing Transactions Among Sessions 187

Chapter 8: Terminal Management 193

Mapping Mode 193
Housekeeping 195
Displaying Screen Output 196
Reading Screen Input 198
Modifying Map Options 202
Writing and Reading in One Step 203

Suppressing Map Error Messages 204
Testing for Identical Data 205
Using Pageable Maps 206
Pageable Map Format 207
Conducting a Map Paging Session 208
How to Code a Browse Application 212
How to Code an Update Application 214
Overriding Automatic Mapout for Pageable Maps 220
Line Mode 223
Beginning a Line Mode Session 223
Writing a Line of Data 224
Reading a Line of Data 226
Ending a Line Mode Session 227
3270-type Considerations 227

Chapter 9: Storage, Scratch, and Queue Management 229

Using Storage Pools 229
User Storage 231
User Kept Storage 233
Shared Storage 235
Shared Kept Storage 236
Storage Pool Summary 238
Using Scratch Records 241
Using Queue Records 249
Using the Terminal Screen To Transmit Data 253

Chapter 10: DC Programming Techniques 255

Passing Program Control 256
Returning to a Higher-level Program 257
Passing Control Laterally 258
Passing Control, Expecting to Return 260
Retrieving Task-Related Information 262
Maintaining Data Integrity in the Online Environment 264
Setting Longterm Explicit Locks 265
Monitoring Concurrent Database Access 268
Managing Tables 272
Retrieving the Current Time and Date 275
Writing to the Journal File 277
Collecting DC Statistics 279
Sending Messages 281
Sending a Message to the Current User 281

Sending a Message to Other Users 284
Writing to a Printer 285
Writing JCL to a JES2 Internal Reader 287
Modifying a Task's Priority 288
Initiating Nonterminal Tasks 288
Attaching a Task 288
Time-Delayed Tasks 289
External Requests 289
Queue Threshold Tasks 290
Controlling Abend Processing 290
Terminating a Task 290
Handling db-key Deadlocks 291
Performing Abend Routines 293
Establishing and Posting Events 294

Chapter 11: Advanced CA IDMS Programming Topics 297

Calling a DC Program from a CA ADS Dialog 297
Basic Mode 299
Reading Data from the Terminal 300
Writing Data to the Terminal 301
Communicating with Database Procedures 301
BIND PROCEDURE 302
ACCEPT PROCEDURE CONTROL LOCATION 303
Managing Queued Resources 304

Chapter 12: Testing 309

Preparing Programs for Execution 309
Selecting Local Mode or Central Version 310
Using SYSIDMS Parameters and DCUF SET Statements 310
Overriding Subschemas (Release 10.2) 311
Overriding a Batch Program's Subschema 312
Overriding an Online Program's Subschema 313
Setting Up an Online Test Application 314

Chapter 13: Debugging 317

Debugging Batch Programs with the Trace Facility 317
Using the CA OLQ Menu Facility 320
Reading Task Dumps 321
Contents of a Snap Dump 321
How to Use the Dump 324

Error Checking 327

Appendix A: PL/I Considerations 329

Transferring Control 329
Using the Online Debugger with PL/I 330
Computation Phase 330
Sample Online Debugger Session 332

Appendix B: Assembler Considerations 335

Appendix C: Batch Access to DC Queues and Printers 337

Appendix D: XA Considerations 339

Appendix E: Running a Program Under TCF 341

Overview of TCF 341
Defining a TCF Task to the DC System 343
Using the UCE for Communication Under TCF 344
Determining if TCF Is Active 345
Starting a New Session 346
Resuming a Suspended Session 346
Processing a Pseudoconverse 347
Suspend Processing 347
End Processing 347
Switch Processing 347
Displaying Error Messages 348
Sample Application Under TCF 348

Appendix F: Services Batch Interface 355

About the 10.2 Services Batch Interface 355

Index 359

Chapter 1: Introduction

This section contains the following topics:

What This Manual Is About (see page 11)

What This Manual Is About

This manual discusses the following topics:

- Programming navigational access to a CA IDMS database
- Programming CA IDMS applications in COBOL, PL/I, and Assembler
- Testing and debugging
- Topics of interest to advanced programmers

This manual is a guide for the developer of batch applications that access a non-SQL defined CA IDMS database. It is also for the developer of applications that execute in a DC system and may or may not access a CA IDMS database.

How Information is Presented

- **Programming functions** are explained by task.
- **Step-by-step instructions** guide you through the programming operations for each function.
- **Special considerations** for using each function are provided where appropriate.
- **Programming examples** are presented in context with other DML and host language-specific source statements.

  All programming examples in this manual are given in COBOL. For specific information regarding PL/I or Assembler, see Appendix A, “PL/I Considerations” or Appendix B, “Assembler Considerations”.

Note: This manual uses the term CA IDMS to refer to any one of the following CA IDMS components:

- CA IDMS/DB- The database management system
- CA IDMS/DC- The data communications system and proprietary teleprocessing monitor
- DC/UCF- The universal communications facility for accessing CA IDMS database and data communications services through another teleprocessing monitor, such as CICS
- CA IDMS DDS- The distributed database system

This manual uses the terms DB, DC, UCF, and DDS to identify the specific CA IDMS component only when it is important to your understanding of the product. References to DC apply equally to UCF unless otherwise noted.

Chapter 2: The CA IDMS Environment

This chapter provides an introduction to the CA IDMS environment.

This section contains the following topics:

What Is CA IDMS? (see page 13)
Accessing CA IDMS Databases (see page 14)
Integrated Data Dictionary (see page 15)

What Is CA IDMS?

Advantage CA Integrated Data Management System (CA IDMS) is a family of products that manage and provide access to data. At its heart is a **database management** system that enables consistent, reliable, and robust data access services to applications running on diverse platforms ranging from the desktop to the enterprise mainframe. CA IDMS databases can be defined and accessed as either relational databases using SQL or as network databases using either SQL or a non-SQL Codasyl-based language referred to as navigational DML.

CA IDMS is also an **application server** (or teleprocessing monitor) providing a run-time environment for the execution of online application programs. These may be client or server applications communicating with a partner through a protocol such as TCP/IP or APPC or they may interact directly with a user at a 3270 terminal. Online programs may be written in COBOL, PL/I, Assembler, or CA ADS, a 4-GL language that facilitates programming in a CA IDMS environment.

Underpinning all of the CA IDMS family of products is an **integrated data dictionary** (IDD) that serves as the repository for data definitions, application definitions, and CA IDMS system configuration parameters. Tools and compilers retrieve information from the dictionary and in turn update it so that it becomes an actively maintained source of usage and cross-reference information.

A number of application related products and facilities extend the core database, application, and dictionary functionality. These include:

- CA IDMS Server- a facility providing SQL access to CA IDMS databases from desktop and mid-range platforms
- CA IDMS Mapping Facility- a tool that enables the definition of 3270 screen layouts
- CA ADS- a 4-GL application development environment
- Advantage CA-Culprit- a reporting and data extract tool for CA IDMS databases and conventional files
- CA OLQ- a query and reporting tool for CA IDMS databases

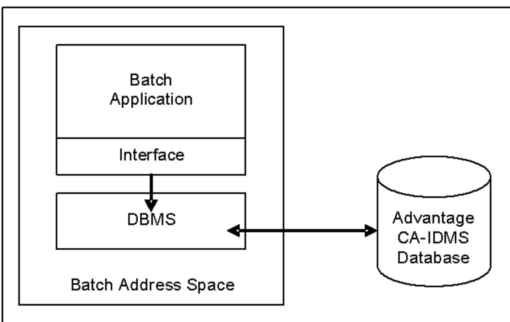
Accessing CA IDMS Databases

CA IDMS databases are accessed using a Data Manipulation Language (DML). DML is a sub-language that is imbedded within another host language, such as COBOL or PL/I. Since the host-language compiler does not understand DML, a program containing DML must be pre-processed using a CA-supplied pre-compiler. The pre-compiler converts the DML requests into host-language statements that result in a run-time call to CA IDMS.

There are two types of CA IDMS database environments: local mode and central version.

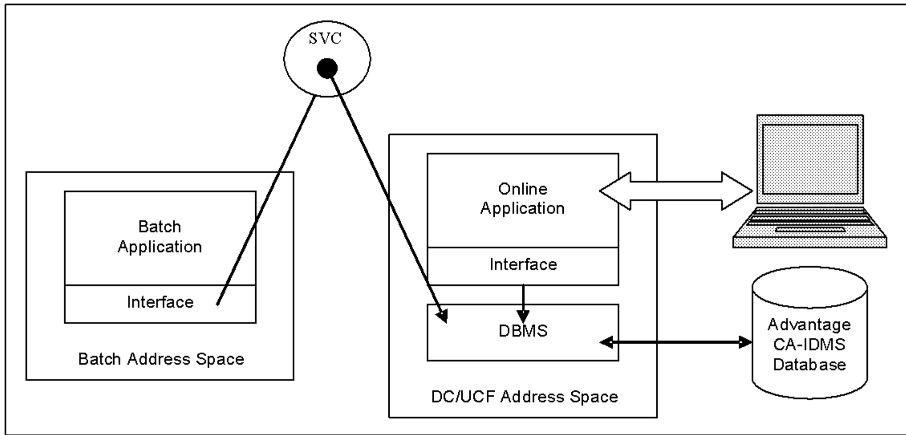
Local Mode

Local mode is a single-user environment providing database services to just one application. Only batch application programs can access a database in local mode. The CA IDMS database manager (DBMS) executes within the same address space as the batch application program as illustrated in the following diagram.



Central Version (CV)

Batch applications *may* and online applications *always* access CA IDMS databases in central version mode. A central version environment supports concurrent access from multiple applications, controls concurrent use of resources, and provides automatic data recovery in the event of failure. The following diagram illustrates both an online and a batch application accessing a database through a central version that is a component of a DC/UCF system.



Any number of local mode or central version environments may be active within an operating system at one time.

Integrated Data Dictionary

The Integrated Data Dictionary (IDD) is a CA IDMS database used as a repository of information by many products and tools in the CA IDMS family. It contains database definitions, source code, application definitions, 3270 screen layouts, DC/UCF system configuration parameters, and security authorizations. Compilers and tools populate the dictionary as part of defining objects such as databases and CA ADS applications and extensive facilities are provided for reporting and ad hoc querying.

When an application program containing DML requests is precompiled, the dictionary is used as a source of information about the database being accessed. Both data descriptions and source code are copied from the dictionary into the program. Additionally, the dictionary can be updated with usage and cross-reference information so that a record is maintained of which objects (maps, source modules, database records, and so on) are used by which program.

There are typically two types of dictionaries within a CA IDMS environment: a system dictionary and an application dictionary. A system dictionary is used to record information about CA IDMS systems and physical database definitions. Application dictionaries contain logical database definitions and application-related entities such as screen layouts and pre-defined source code. As a programmer, you will access application dictionaries to compile programs and reference the logical description of a database. .mg// imbedding (chap2 iddmlh00)

Chapter 3: Database Concepts

This chapter introduces many of the concepts associated with CA IDMS databases. It describes how data in the database is organized, stored and inter-related and how that information is represented graphically through a data structure diagram.

It is important for navigational DML programmers to understand these concepts because, as discussed in the next chapter, the way in which data is organized affects the types of navigational DML commands that can be used to access it. Furthermore, the choice of which type of access technique is used can significantly impact an application's performance.

This section contains the following topics:

Types, Occurrences, and Relationships (see page 17)
Records (see page 19)
Sets (see page 20)
Representing Relationships as Sets (see page 32)
Areas (see page 43)
Logical Database Description (see page 44)
Physical Structures (see page 47)
Data Structure Diagrams (see page 59)

Types, Occurrences, and Relationships

From a data modeling perspective, data can be represented as types and relationships can exist between types. Physical instances of a type are referred to as occurrences and all occurrences conform to the rules specified in the type definition.

Types and Occurrences

A CA IDMS database consists of physical occurrences of data whose types are defined within the database's logical description. Types of data are generic descriptions of occurrences of that type.

As an example of data occurrences and types, consider a file of employee information at a company called Commonweather Corporation. The names of the employees on the file, such as John Done and June Moon, are data **occurrences**. The names actually exist as characters in a disk file. The **type** for these data occurrences is a description that fits all possible occurrences, which in this case might be defined as "An employee's name consists of 25 alphanumeric characters. Short names shall be padded on the right with blanks. Long names shall be truncated."

To be useable by a program, the description must be specified in a standard way. For example, a COBOL program using employees' names might include the following in its DATA DIVISION:

```
EMPL-NAME PICTURE X(25).
```

For CA IDMS databases, types are defined separately from programs and are part of the database's definition residing in an application dictionary. Type definitions are copied into a program during the pre-compilation process.

Types can be singular items, such as that representing an employee's name or groups of items representing a **complex type** such as an address or an **entity** such as an employee. When modeling a process or an enterprise, entities often represent things that impact or are impacted by other things.

As shown in the following table, the terms used for these various constructs depend on whether they appear in the definition of a relational (SQL defined) or network (non-SQL defined) database.

Construct	Relational Term	Network Term
Singular type	Column	Element (record element, data item, field)
Complex type	--	Group element
Entity type	Table	Record type

Relationships

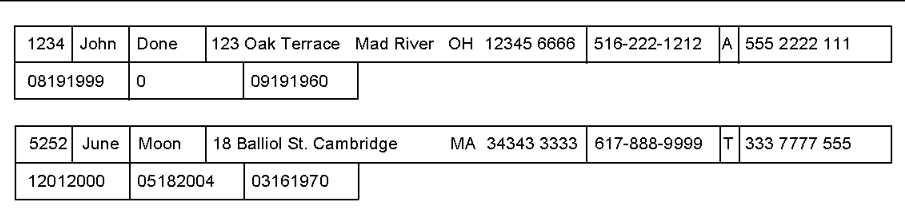
A relationship is a logical association between two or more entity types. For example, at Commonweather Corporation, there are employees and departments each of which is represented by an entity type in the database. The group of employees that work in a given department is an example of a relationship between the department and employee entity types. The employee that is the head of a department is another example of a relationship between these same types.

A relationship may be optional or it may be required. For example, if every employee must work in some department, the relationship is required and it imposes a constraint on occurrences of the employee type. In an SQL defined database, such relationships are called referential constraints; in a non-SQL defined database, such relationships are called sets.

Records

A **record occurrence** (or simply a **record**) is the basic addressable unit of data using navigational DML. It consists of a fixed or variable number of bytes of data subdivided into units called **elements** or fields. Every occurrence of a record is described by a **record type** defined in a **schema**, the logical description of a database. All records of the same type contain the same elements arranged in the same order.

For example, John Done's record consists of eleven items including employee ID, first name, last name, address, phone number, status, social insurance number, start date, termination date, and birth date. June Moon's record also consists of the same eleven items.



In COBOL, the EMPLOYEE record might be expressed as follows:

```
01  EMPLOYEE.

   02 EMP-ID                    PIC 9(4).
   02 EMP-NAME.
     03 EMP-FIRST-NAME            PIC X(10).
     03 EMP-LAST-NAME             PIC X(15).
   02 EMP-ADDRESS.
     03 EMP-STREET              PIC X(20).
     03 EMP-CITY              PIC X(15).
     03 EMP-STATE               PIC XX.
     03 EMP-ZIP.
       04 EMP-ZIP-FIRST-FIVE       PIC X(5).
       04 EMP-ZIP-LAST-FOUR        PIC X(4).
   02 EMP-PHONE                 PIC 9(10).
   02 STATUS                  PIC XX.
```

```
02 SS-NUMBER              PIC 9(9).
02 START-DATE.
  03 START-YEAR           PIC 99.
  03 START-MONTH          PIC 99.
  03 START-DAY            PIC 9(4).
02 TERMINATION-DATE.
  03 TERMINATION-YEAR     PIC 99.
  03 TERMINATION-MONTH    PIC 99.
  03 TERMINATION-DAY      PIC 9(4).
02 BIRTH-DATE.
  03 BIRTH-YEAR           PIC 99.
  03 BIRTH-MONTH          PIC 99.
  03 BIRTH-DAY            PIC 9(4).
```

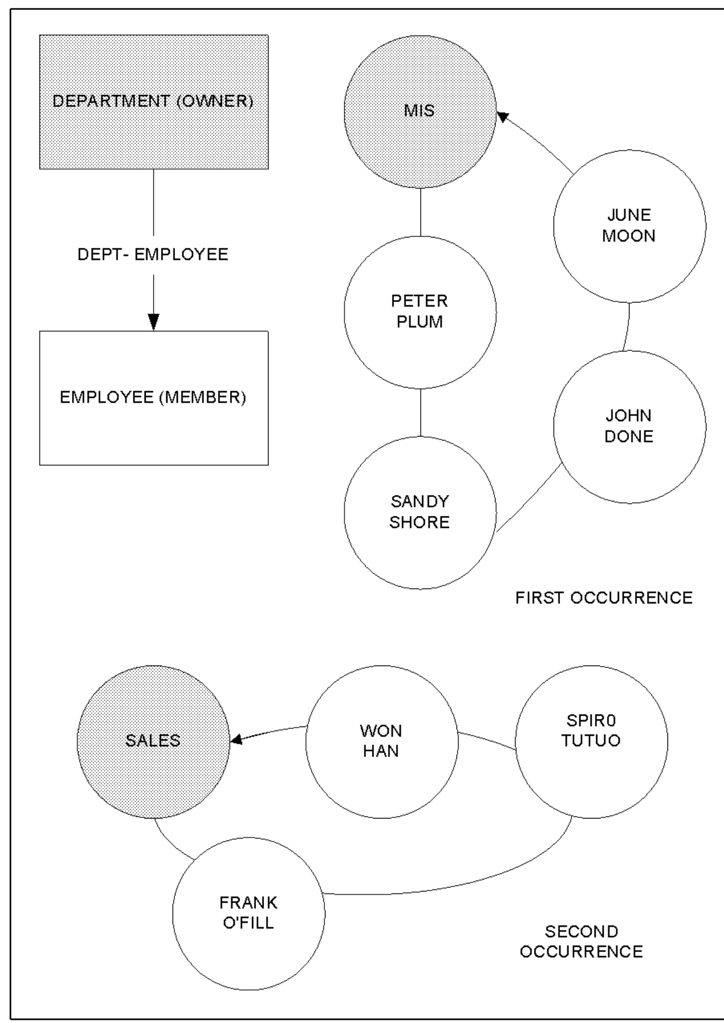
The record definition enables an application program to identify and locate each element in a record occurrence. For example, having retrieved John Done's record occurrence you know that its type is EMPLOYEE. Within EMPLOYEE, the first element in the record is the employee's identification and the last element their birth date.

Sets

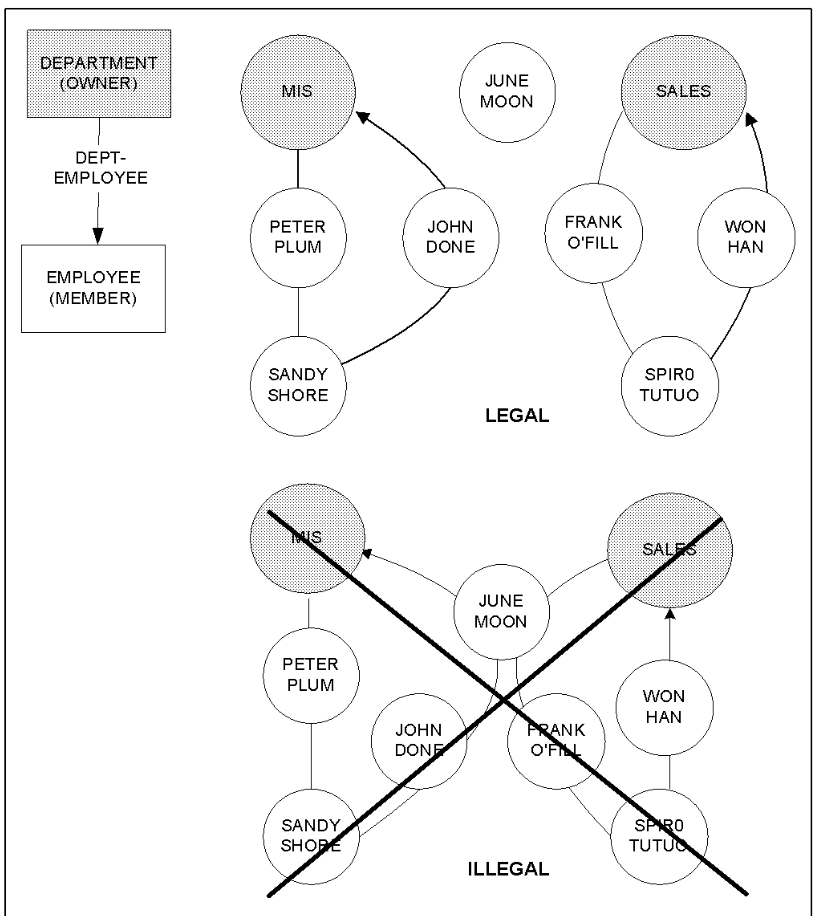
A **set type** expresses a relationship between two (or more) record types, where one record type is the owner and the other is the member. For example, to show the departments to which the employees belong, we might establish a DEPT-EMPLOYEE set type, where the record type DEPARTMENT is the owner and the record type EMPLOYEE is the member.

One **set occurrence** exists for each occurrence of the owner record. Any number of member record occurrences may be part of one set occurrence. For example, each DEPT-EMPLOYEE set occurrence consists of the one DEPARTMENT record plus any numbers of EMPLOYEE records, depending on how many employees are in the department.

The following figure illustrates the DEPT-EMPLOYEE set type and two occurrences of the set. Note that each set occurrence contains only one DEPARTMENT record (the owner) but several EMPLOYEE records (members).



A member record occurrence does not have to participate in a set occurrence but it cannot be connected to more than one set occurrence within a given set type. The following figure shows that June Moon does not have to participate in a DEPT-EMPLOYEE set but she cannot belong to both MIS and Sales.



Attributes of a set determine how it is represented within the database and specify the rules for membership. The most significant of these are:

- Set mode
- Set linkage
- Set order
- Set membership options

Set Mode

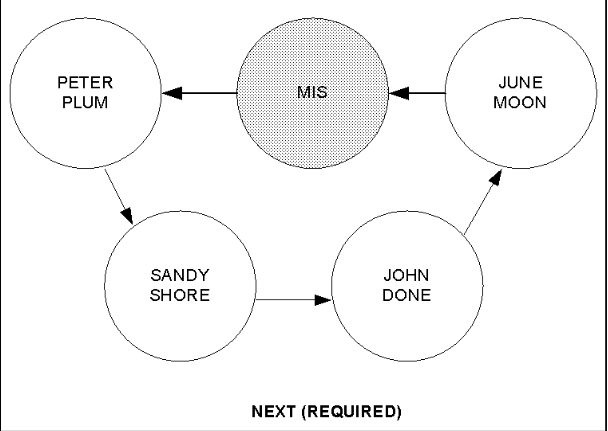
The mode of a set identifies how the relationship between participating record occurrences is represented within the database. The choices are:

- **CHAIN**- physically links each record in the set to the next record in the set. In other words, each record points to the next record in the set.
- **INDEX**- an index structure, or pointer array associates the owner of the set with each of its members. There is no physical linkage between adjacent member records.

Set Linkage for Chain Sets

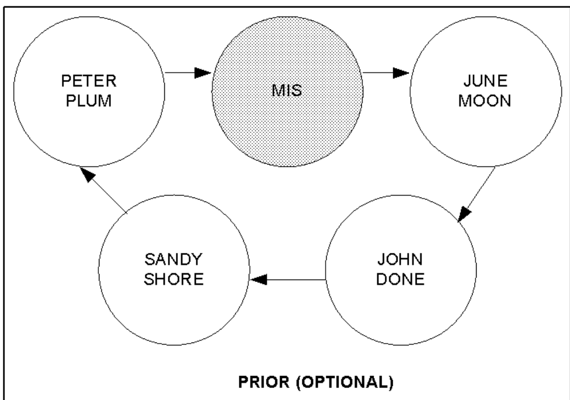
The way in which the records in a chain set are linked together is specified by one or a combination of the following pointer options:

- **NEXT (required)**- The owner points to the first member. Each member points to the member after it. The final member points back to the owner establishing a ring structure.



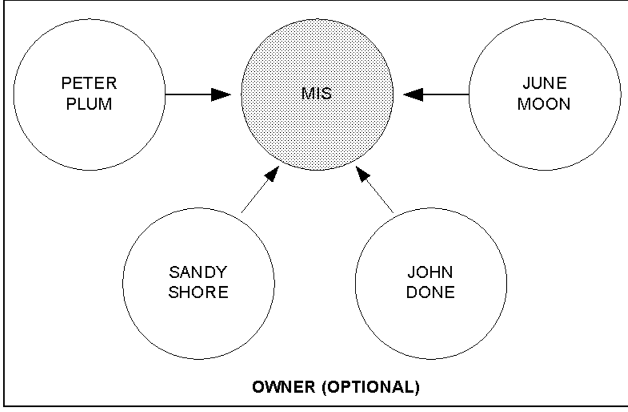
The NEXT pointer in John Done's record occurrence addresses June Moon's record occurrence.

- **PRIOR (optional)**- The owner points to the last member in the set. The last member points to the next-to-the-last member. The first member points to the owner creating a reverse ring.



The PRIOR pointer in John Done's record occurrence addresses Sandy Shore's record occurrence.

- **OWNER (optional)**- Each member points to the owner.



The OWNER pointer in each member addresses the OWNER record.

Set Order

The logical order in which new members are linked into a set occurrence follows one of the following user-specified methods:

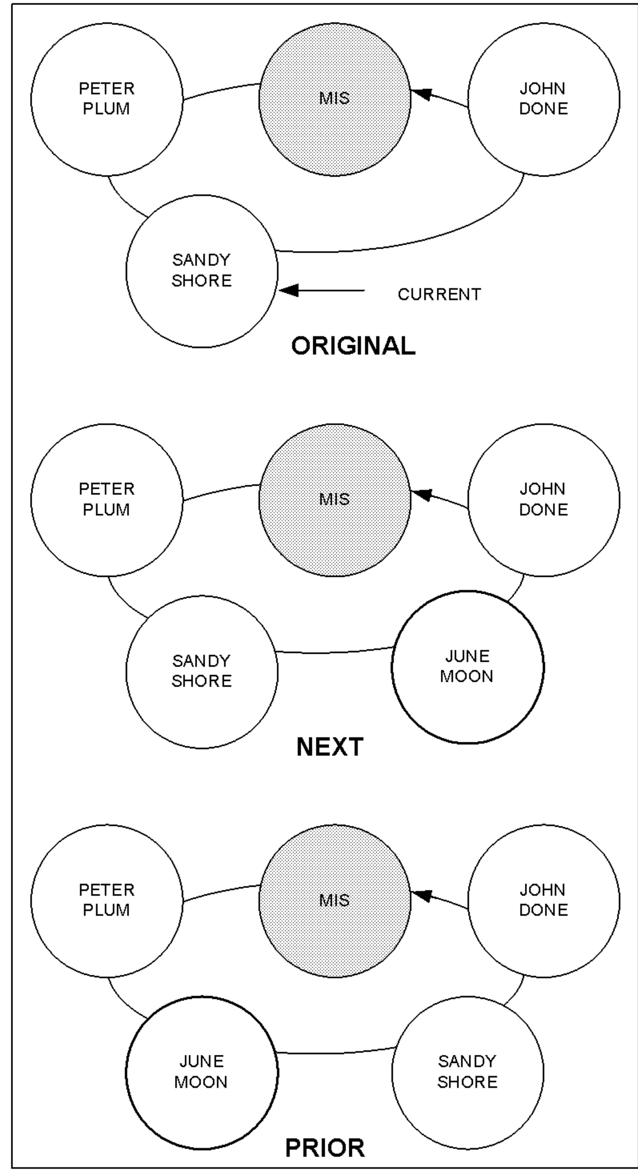
- **FIRST**- A LIFO (last-in first-out) order. The new record is positioned immediately after the owner record.
- **LAST**- A FIFO (first-in first-out) order. The new record is positioned immediately before the owner record (that is, after the last existing member record). PRIOR pointers are required.

- **NEXT**- A simple list. The new record is positioned immediately after the current (most recently accessed) record.
- **PRIOR**- A reverse list. The new record is positioned immediately before the current record. PRIOR pointers are required.
- **SORTED**- A sorted list or index. The new record is positioned according to the value of one or more of its elements (called a sort key) relative to the values of the same element(s) in the other member records. Member records can be arranged in ascending or descending order with respect to the designated sort key. For example, if the sort key is employee last name arranged in ascending order, then June Moon will be added after John Done in the set.

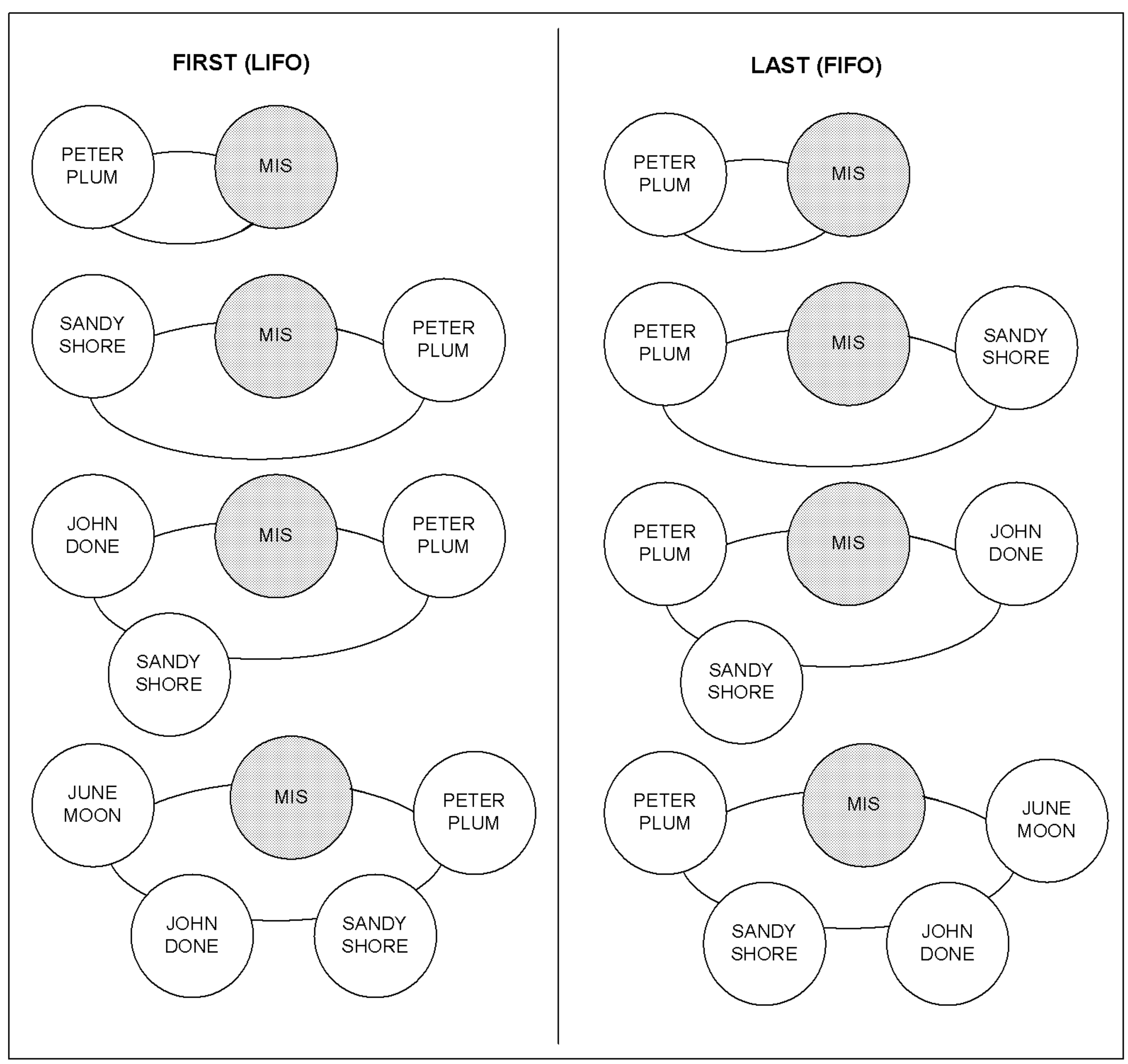
  Records with duplicate sort key values can be positioned first or last of members with the same values, or they can be disallowed, ensuring that each member in the set has a unique sort key value.

Note: An element that is part of a sort key is also referred to as a **sort control item**.

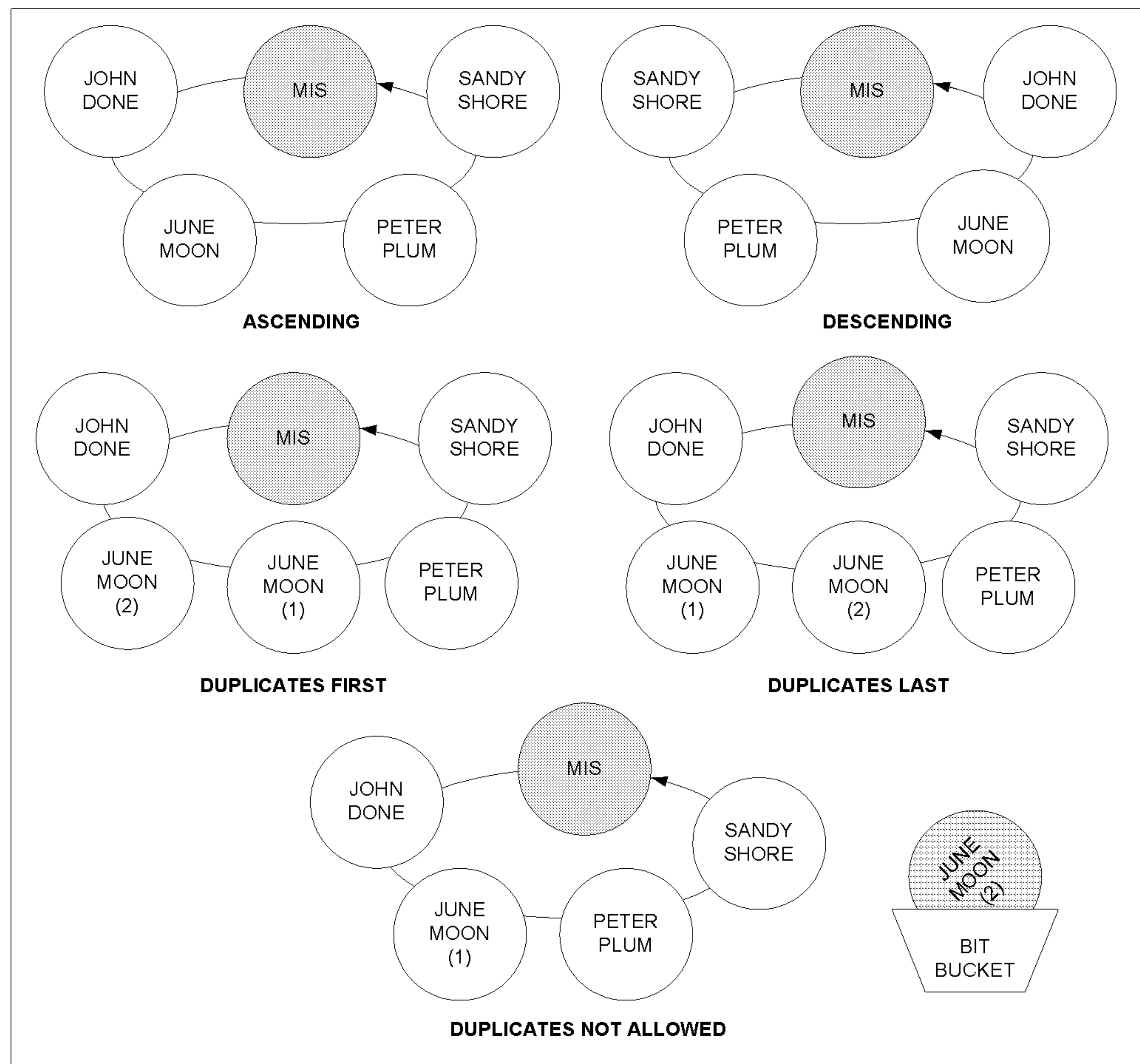
The following diagram illustrates the effects of NEXT and PRIOR set order in a chain set. We are positioned on Sandy Shore and add June Moon. If the set order is NEXT, June Moon follows Sandy Shore. In a PRIOR-ordered set, June Moon precedes Sandy Shore.



The following diagram illustrates the effects of FIRST and LAST set order in a chain set. In each case, we add four records to the database: Peter Plum, Sandy Shore, John Done, and June Moon. If the set order is LAST, each new record is added at the end of all current members. The result is a First-In-First-Out (FIFO) ordering of the members, since the first member in the set is the first member added. If the set order is FIRST, each new record is added ahead of all current members, resulting in a Last-In-First-Out (LIFO) ordering of the members.



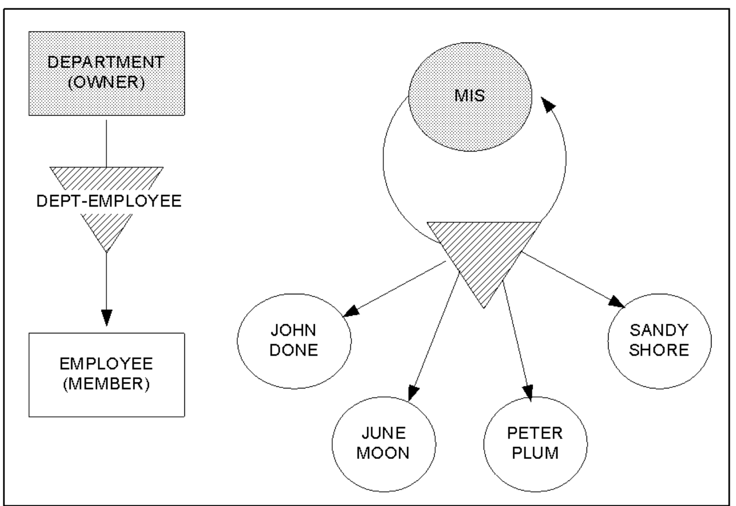
The following diagram illustrates the effects of SORTED set order and various duplicates options in a chain set. The sort key is the employee's last name. The first set occurrence represents a set whose members are sorted ascending by last name, while the second represents a set sorted descending by last name. The remaining three show the impact of different duplicates options when a second occurrence of June Moon is added to a set whose sort order is ascending.



Index Sets

In an index set, a pointer array, or index, associated with each owner occurrence contains pointers to all related member record occurrences. There is no direct linkage between the members of an index set.

The following diagram illustrates an occurrence of the DEPT-EMPLOYEE set implemented as an index rather than a chain set. In this example, the MIS department record has an associated index structure that contains an entry for each associated employee.

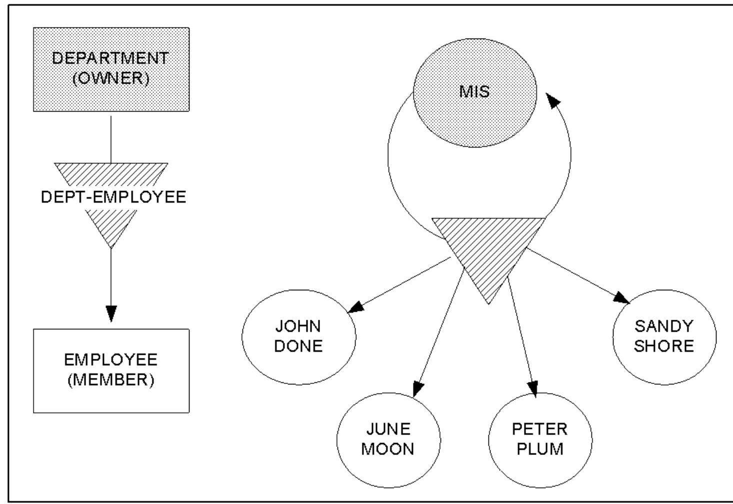


Each entry in the index contains a pointer to a member record. The order of the entries determines the logical order of the members in the set. Just as for a chain set, the set order of an index set determines where new members are logically inserted into the set and can be: FIRST, LAST, NEXT, PRIOR, or SORTED. If the set order is sorted on a symbolic key, each index entry also contains the member record's sort key value and the entries in the index are sequenced to reflect the ascending or descending option with respect to those values. Records with duplicate sort keys can be positioned first or last of members with the same value, or they can be disallowed. The term **index key** is often used to refer to the sort key of an index set.

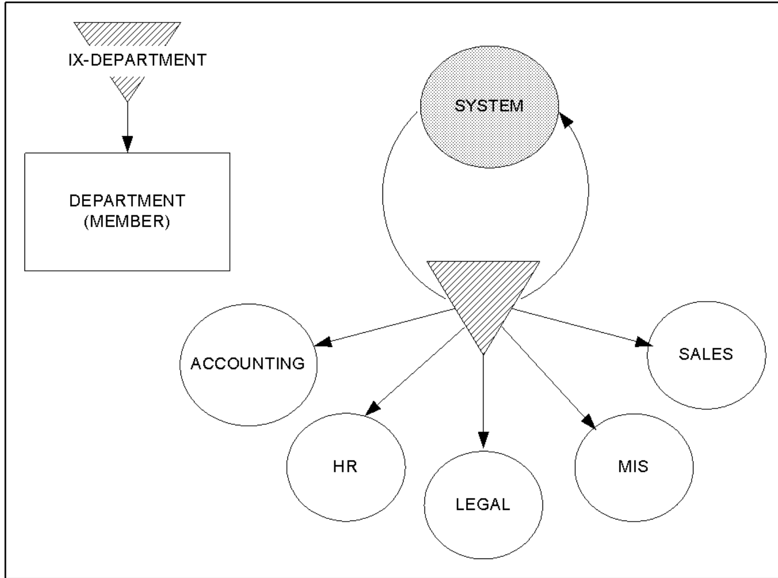
User-Owned and System-Owned Index Sets

There are two types of index sets: user-owned and system-owned. A **user-owned index** implements a logical relationship between the owner entity type and the member entity type. A **system-owned index** provides direct access to member record occurrences for performance purposes and as the name implies, its owner is a special SYSTEM owner record type. A system-owned index is often referred to simply as an "index."

If the DEPT-EMPLOYEE set is implemented as an index set, it is a user-owned index because the owner record type is DEPARTMENT. There will be one index structure for each occurrence of the DEPARTMENT record type.



There is always exactly one occurrence of a system-owned index set. The owner is a special CA-defined record and all occurrences of the member record type may be members. It is often used to provide a way to access member records by an exact or generic key value. For example, we may need to access departments by name instead of department id. A system-owned index called IX-DEPARTMENT has been defined with DEPT-NAME as the sort or symbolic key. The following figure illustrates this index.



Set Linkage for Index Sets

The linkage options for an index set are different than those for a chain set. The owner of an index set always points to the index structure and not the first member record occurrence. The pointers maintained in the member record differ depending on whether the index is user- or system-owned.

In a user-owned index, member record occurrences always point into the index structure. This is called an **INDEX** pointer. Member records may also optionally point to the owner record occurrence (**OWNER** pointers).

In a system-owned index, member records never point to the owning SYSTEM record and may or may not point into the index structure depending on whether the index is defined as linked or unlinked:

- **Linked**- index pointers are maintained in the member records.
- **Unlinked**- index pointers are not maintained in the member records.

Set Membership Options

Relationships can be optional or required. If a relationship is optional, an occurrence of the member record may or may not belong to an occurrence of the set that represents the relationship. Conversely, if a relationship is required, all occurrences of the member record type must belong to some occurrence of the set.

Set membership options control whether an occurrence of the member record type always belongs to an occurrence of the set. Membership options are specified as a pair of sub-options often referred to as disconnect/connect options.

The **connect** option of a set can be Automatic or Manual. If a set is defined as **Automatic (A)**, then when an occurrence of the member record type is added to the database, it is automatically connected into an occurrence of the set. In contrast, record occurrences must be explicitly (programmatically) associated with an occurrence of a **Manual (M)** set.

The **disconnect** option controls when a member record can be removed from a set. If a set is defined as **Mandatory (M)**, the only way to remove a member record from the set is to delete the record from the database. Member records of **Optional (O)** sets can be removed from the set without deleting the record.

In the Commonweather database, for example, the membership options of the DEPT-EMPLOYEE set are OPTIONAL/AUTOMATIC (OA). The Automatic option specifies that when an employee record is added to the database, it is automatically associated with the department in which the employee will work. The Optional option specifies that employee record occurrences can be transferred from one department (one set occurrence) to another without being deleted from the database.

Representing Relationships as Sets

The logical data relationships that exist among entities can be classified as hierarchies or networks. The following discussion examines variations of these relationship classes and illustrates how they can be represented as sets.

Hierarchies

Hierarchical relationships can be two-level hierarchies or multilevel hierarchies and an entity can be the owner in multiple relationships. All of these can be represented using sets.

Two-level Hierarchies

A single set represents a two-level hierarchy. The owner record is the first level and the member records comprise the second level. This represents a one-to-n (or one-to-many) relationship between the owner record and the member records. Hierarchical relationships can be represented as repeating elements within a record or multiple record types. For example, in the DEPT-EMPLOYEE set described earlier in this chapter, each employee's information is stored as a separate record. The data could alternatively be expressed as a single record type by making EMPLOYEE a repeating element in DEPARTMENT. In COBOL, the record would appear as follows:

```
01 DEPARTMENT.
  02 DEPT-ID                 PIC 9(4)
  02 DEPT-NAME               PIC X(45)
  02 DEPT-HEAD-ID            PIC 9(4)
  02 EMP-CNT                 PIC S9(4) COMP SYNC.
  02 EMPLOYEE                OCCURS 0 TO 100 TIMES
                           DEPENDING ON EMP-CNT
    03 EMP-ID                PIC 9(4).
    03 EMP-NAME.
      04 EMP-FIRST-NAME          PIC X(10).
      04 EMP-LAST-NAME           PIC X(15).

    03 EMP-ADDRESS.
      . . .
    03 BIRTH-DATE.
      04 BIRTH-YEAR            PIC 99.
      04 BIRTH-MONTH             PIC 99.
      04 BIRTH-DAY             PIC 9(4).
```

The main reasons for expressing relationships as sets rather than as repeating elements within records are as follows:

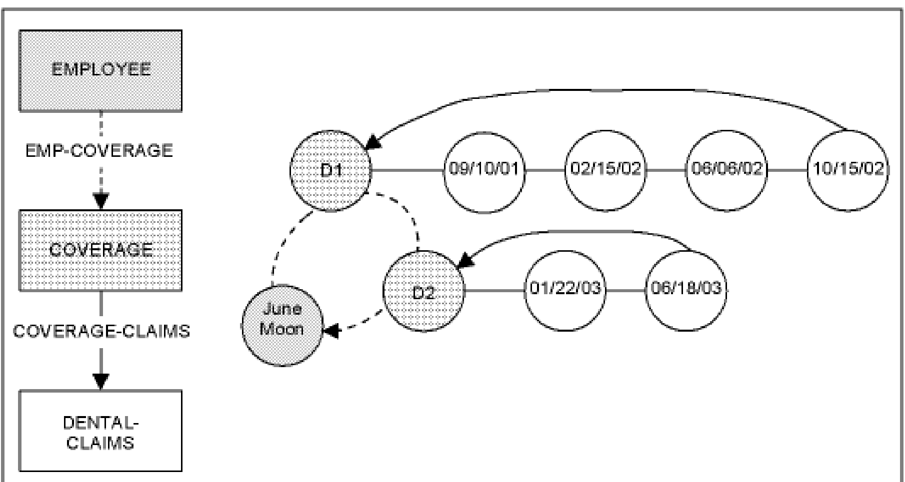
- A record can be accessed directly; an element cannot.
- Records can participate in many sets, thereby permitting a combination of relationships that would be impossible among individual elements.
- It is often difficult to know how many occurrences to allow for and over-estimation wastes storage since programs must reserve enough space to hold the maximum number.

On the other hand, maintaining information as elements means fewer records and more information per record access. Whether to break out repeating elements as member records of a set is a basic question in designing a database and one which is best approached on a case-by-case basis.

Consider, for example, dental claims made by an employee of Commonweather Corporation. Each procedure for which a claim is made must be described and recorded on the database. While multiple procedures can be claimed together, the number is limited and the amount of information recorded about each is small. For these reasons, dental procedure information is stored as repeating items within the DENTAL-CLAIM record occurrence with which they are associated rather than as separate records.

Multilevel Hierarchies

A member of a set can be the owner of another set, representing a multilevel hierarchy. At Commonweather, for example, dental claims constitute a separate record type called DENTAL-CLAIM. Dental claims are associated with the insurance coverage under which they are made through the COVERAGE-CLAIMS set. The owner of this set, an occurrence of the COVERAGE record type, represents the terms of coverage that an employee has under one of the company's insurance plans. The owning coverage record is a member of the EMP-COVERAGE set, resulting in the three-level hierarchy shown in the figure below. It further illustrates an occurrence of the EMP-COVERAGE set and two related occurrences of the COVERAGE-CLAIMS set.

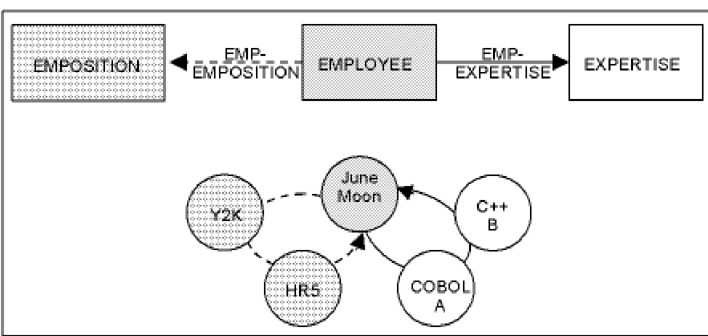


Using several sets to form a multilevel hierarchy is similar to using nested repeating groups in a COBOL record description. The hierarchy shown above, for example, could be represented in the COBOL DATA DIVISION as follows:

```
01 EMPLOYEE.
  02 EMP-ID                PIC 9(4).
  02 EMP-NAME.
    03 EMP-FIRST-NAME        PIC X(10).
    03 EMP-LAST-NAME         PIC X(15).
  02 COV-CNT                 PIC S9(4) COMP SYNC.
  02 COVERAGE            OCCURS 0 TO 10 TIMES
                         DEPENDING ON COV-CNT.
    03 INS-PLAN-CODE         PIC X(3).
    03 SELECTION-DATE        PIC 9(8).
    03 CLAIM           OCCURS 0 TO 10 TIMES
                         DEPENDING ON CLAIM-CNT.
      04 CLAIM-DATE          PIC 9(8).
```

Multiple Ownership

One record type can own more than one set reflecting the fact that an entity can be the owner in multiple 1-to-n relationships. For example, the EMPLOYEE record type at Commonweather Corporation owns the EMP-EXPERTISE set identifying the skills held by a given employee and the EMP-EMPOSITION set to keep track of the various projects to which the employee is assigned. The following figure illustrates this example of multiple ownership: occurrences of the EMP-EXPERTISE set and the EMP-EMPOSITION set are shown with both set occurrences owned by the June Moon record occurrence.



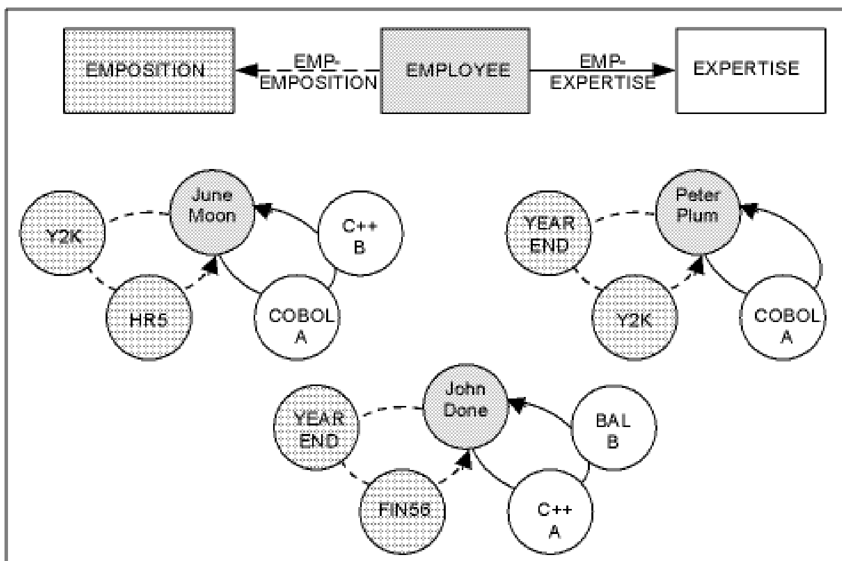
Multiple set ownership logically unites the member records of two (or more) types while maintaining discrete linkages and orders. The common owner record provides the means of movement between the different member records. For example, using the following diagram, if we are trying to locate employees on the Y2K project that have at least a B proficiency in C++, we could satisfy this query by asking the following questions:

Question: For each EXPERTISE that indicates a B proficiency in C++, what record is the owner in the EMP-EXPERTISE set?

Answer: The June Moon and John Done record occurrences.

Question: What are the members of the EMP-EMPOSITION set that June Moon owns? That John Done owns?

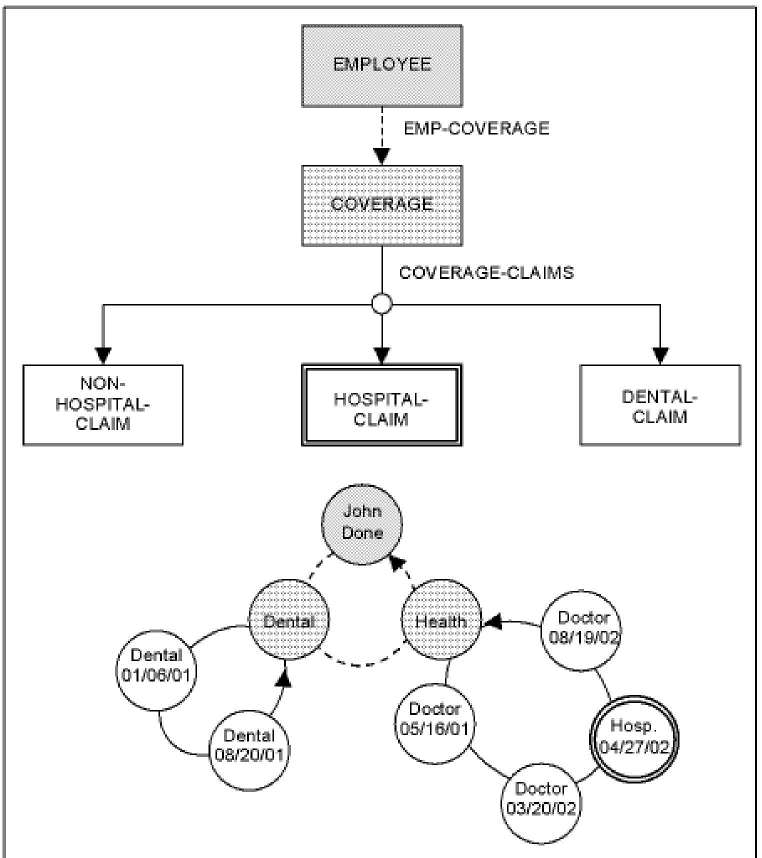
Answer: June Moon: Y2K, HR5. John Done: YEAR END, FIN56.



Multiple Ownership Using One Set

If a record type is the owner in multiple relationships, it is possible to represent those relationships using a single set in which there is one owning record type but more than one member record type. Such a set is called a *multimember set*.

Multimember sets reduce the number of links and therefore the storage needed to represent multiple relationships. They are often used when a set occurrence will typically have members of only one type or when the member records are often processed together. For example, Commonweather keeps track of three types of insurance claims: dental claims, hospital claims, and others such as doctor and life. Since different information is recorded for each type of claim, each has its own record type: DENTAL-CLAIM, HOSPITAL-CLAIM, NON-HOSPITAL-CLAIM. The relationship between an employee's insurance coverage (the COVERAGE record type) and the three types of claims could be represented by three sets. However, Commonweather has chosen to represent them using one set, COVERAGE-CLAIMS, because a given coverage is usually associated with only one type of claim (for example, only dental claims are associated with coverage under a dental plan). The exception to this is coverage under a health insurance plan against which claims can be made for both hospital and non-hospital services. By ordering the set LAST, claims made under a given coverage will appear in the order in which they are made. The following diagram illustrates this example, showing John Done's coverage in two insurance plans and the claims that he has made against those plans.

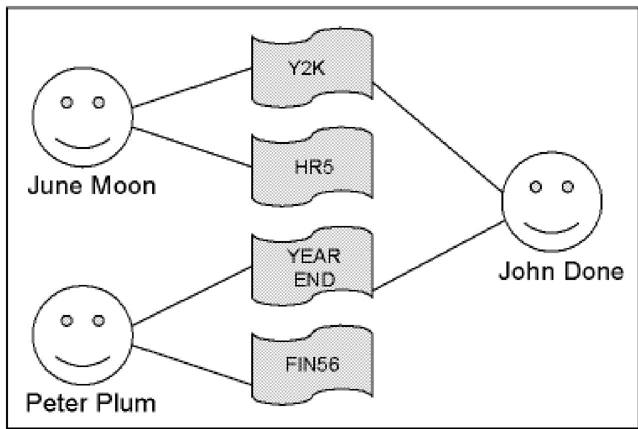


Networks

In a one-to-many relationship, a set occurrence contains one owner record and multiple member records. An owner can own more than one set providing a way to move from one set to another. It is also possible to have a many-to-many relationship, called a *network*, where an owner is related to multiple members and those members are related to multiple owners. Networks provide a link from the owner of one set to the owner of another set.

In a previous example, we related an employee to the projects to which they are assigned. We know that June Moon works on the Y2K and HR5 projects, John Done works on the YEAR END and FIN56 projects, and Peter Plum works on the YEAR END and Y2K projects. If an employee were assigned to only one project at a time, we could establish an EMPLOYEE-PROJECT set to represent the relationship between projects and employees. However, a single project often has several employees assigned to it and each employee usually is assigned to several projects. This creates a many-to-many or network relationship. None of the relationships discussed so far are many-to-many relationships. The solution to the network problem, as we will see, lies in multiple membership.

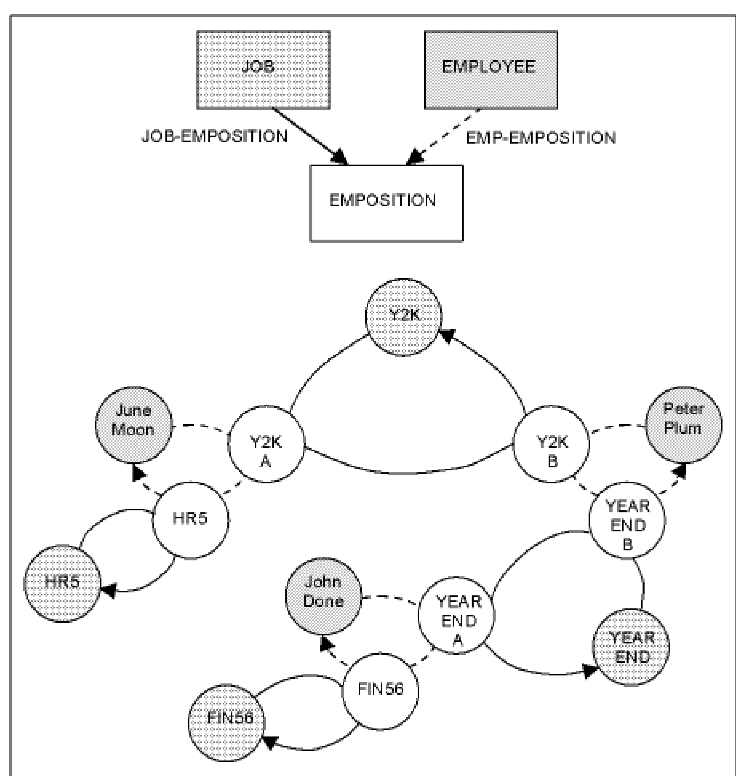
The figure below illustrates the many-to-many relationship between projects and employees. Both June Moon and John Done are assigned to Y2K. June Moon is also assigned to HR5; John Done is also assigned to YEAR END as is Peter Plum who is the only one working on FIN56.



Multiple Membership

Networks are created when one record type is a member of more than one set. Multiple membership logically relates the owner records of two (or more) sets that have common members. An owner record of the first set type is related to many members. Each of these members is related to an owner of the second set type. This creates a one-to-many relationship between the owners of the first set type and owners of the second set type. Since the logic applies equally in reverse, we have a many-to-many relationship between the two owner types.

In our example, the EMPOSITION record type at Commonweather Corporation is owned by both the EMPLOYEE and JOB record types. The following diagram illustrates this example of multiple membership for June Moon, John Done, Peter Plum, and the four project positions to which they are assigned.



We can use this example to determine who is assigned to the Y2K project by answering the following questions:

Question: What are the members of the JOB-EMPOSITION set when Y2K is the owner?

Answer: Y2K-A and Y2K-B.

Question: Which record owns the EMP-EMPOSITION set occurrence when Y2K-A is a member? When Y2K-B is a member?

Answer: John Done and June Moon.

A record type can be a member of an unlimited number of sets. This allows the representation of complex interrelationships within a database without replicating data.

Junction Records

A junction record is the common member record in a multiple membership structure. A junction record serves two functions:

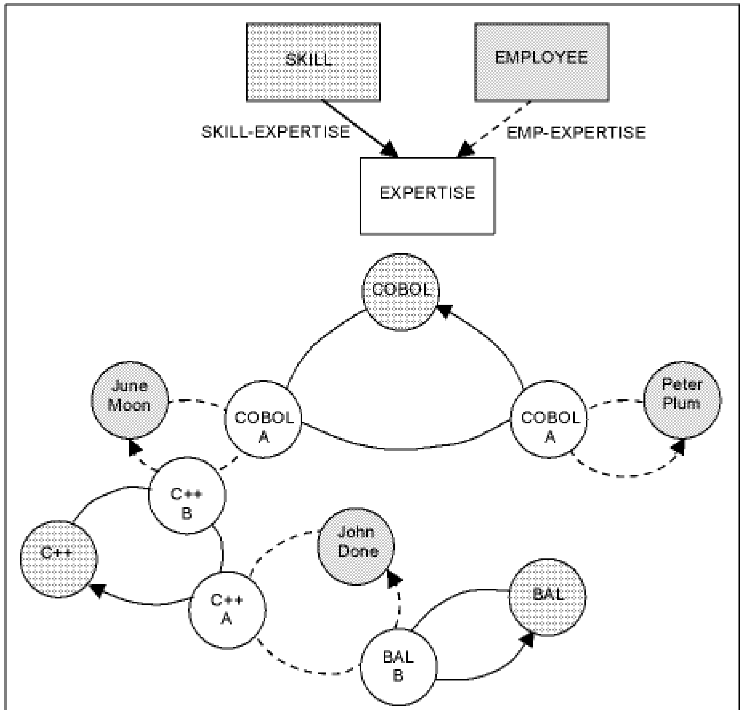
- Enables the many-to-many relationship between its owners.
- Contains data specific to the intersection of the owner records.

The EMPOSITION record of the previous example enables the representation of the many-to-many relationship between the EMPLOYEE and JOB record types. At the same time, the contents of an EMPOSITION occurrence describe the relationship between its two owners by recording the beginning and ending dates of the employee's assignment to the project and their position on the project.

Another example of a many-to-many relationship at Commonweather is between employee and skills. An employee can have many skills while many employees can have the same skill. Since a direct EMPLOYEE-SKILL set is not possible, the EXPERTISE junction record is used to:

- Join the EMPLOYEE and SKILL record types in a many-to-many relationship
- Store an employee's proficiency in a skill

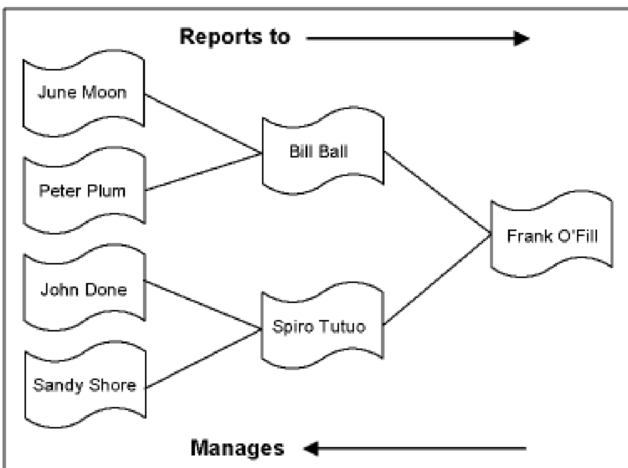
The following figure illustrates three occurrences of the SKILL-EXPERTISE set and the three related occurrences of the EMP-EXPERTISE set. The junction record (EXPERTISE) denotes a skill that an employee has.



Bill-of-Materials Structures

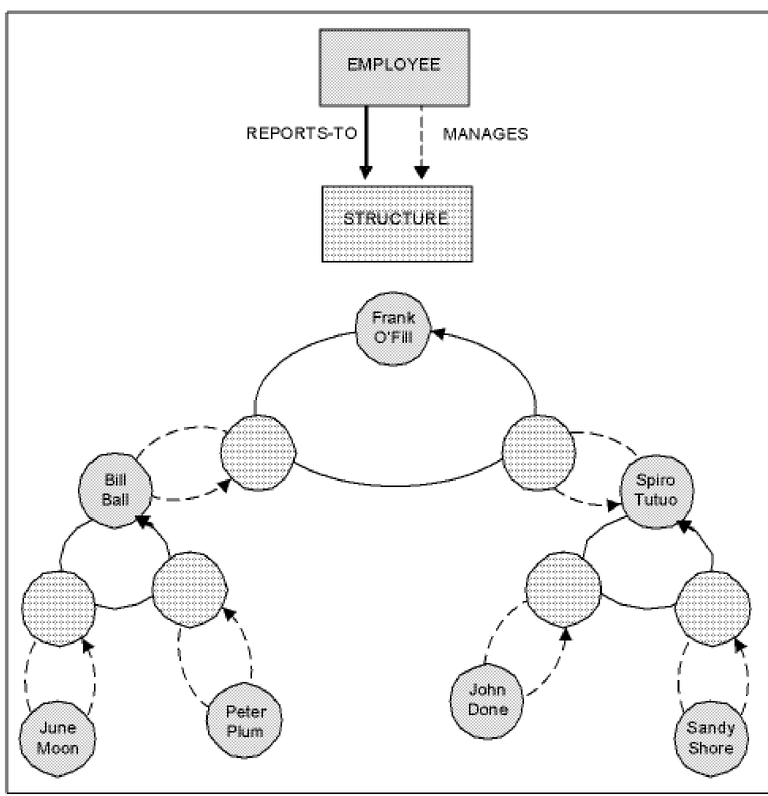
A bill-of-materials structure is a network relationship between record occurrences of the same type. The name comes from the relationship among parts in an industrial assembly operation, where a part may be a component of another part and itself be comprised of other parts as its components.

At Commonweather, we find a bill-of-materials relationship among employees: an employee may report to a manager and also manage other people. The following diagram illustrates this relationship. Bill Ball manages June Moon and Peter Plum; Spiro Tutuo manages John Done and Sandy Shore. Both Bill Ball and Spiro Tutuo report to Frank O'Fill.



The standard technique for representing a bill-of-materials relationship is to use multiple membership and a junction record. Normally such a structure involves three record types and two sets, but in a bill-of-materials structure, two of the record types are the same.

The following diagram illustrates the bill-of-materials relationship among employees at Commonweather Corporation. A junction record, called STRUCTURE is used to relate employees and their managers. By finding the members of an occurrence of the REPORTS-TO set, and then finding their respective owners in the MANAGES set, we identify the managers of an employee. (If Commonweather uses matrix management, there might be multiple members in an occurrence of the REPORTS-TO set.) Reversing the process allows us to determine the employees that report to a given manager.



Areas

Databases are divided into one or more logically contiguous storage units called areas. Each record type is associated with one of these areas. All occurrences of the same record type reside in the area to which its record type is assigned. Each system-owned index is also assigned to an area indicating where the associated index structure resides.

Logical Database Description

A CA IDMS database description consists of three components:

- A logical description represented by a **schema** reflecting the entity types and relationships contained within the database.
- A physical description specifying how the data is physically stored on direct access storage devices (DASD).
- A program's view, represented by a **subschema**, which is often a subset of the schema types and relationships.

By separating the logical and physical definitional components, it is possible for a single schema to describe multiple physical database instances. For example, a single schema can describe both the test database used for application development and the QA database used for staging applications into production.

Data independence is achieved by allowing a program to view the database differently from how it is defined in the schema.

Schemas

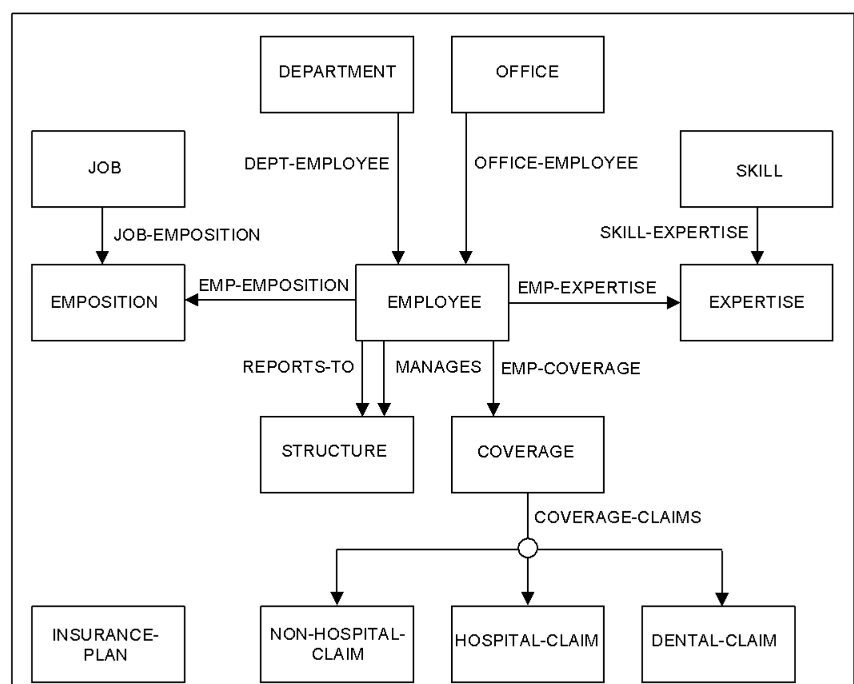
A **schema** provides the description of the logical contents if a database. Schemas can describe a database in terms of:

- Tables- a relational database whose schema is defined using SQL Data Definition Language (DDL)
- Record Types- a network database whose schema is defined using a (non-SQL) Codasyl-based DDL

Only a database described by a non-SQL schema can be accessed using navigational DML.

A non-SQL schema defines the record types and sets that implement the entities and relationships within the database. The schema also identifies the logical areas into which the database is divided and assigns record types and system-owned indexes to areas.

The following diagram illustrates the contents of the schema for Commonweather Corporation's database.



Summary of Logical Structures

The following table summarizes the logical structures used in representing data and data relationships in CA IDMS.

NAME	TYPE	OCCURRENCE	DESCRIPTION
Record			Basic unit of addressable data
Set			Basic expression of a hierarchical (1 to *n*) relationship
Multimember Set			Hierarchy with more than one member record type
Multiple Ownership			Multiple relationships
Multilevel Hierarchy			Hierarchy with more than two levels
Multiple Membership			Basic expression of a network (*n*-to-*n*) relationship
Bill-of-Materials			Multiple relationships between two records
User Owned Index Sets			A hierarchy (1-*n*) relationship implemented through an index
System Owned Index Sets			An index providing direct access to record occurrences

Physical Structures

Physical Database Description

A database instance is represented by one or more **segments**. A segment definition describes the physical attributes of a database such as how data is stored on Direct Access Storage Devices (DASD) and how much space the database contains.

Segments are included into a **DMCL**, a component used at run time to define the universe of databases accessible by a CA IDMS run-time environment (either local mode or central version). Each run-time environment uses a single DMCL load module generated from a DMCL source definition residing in a system dictionary.

A **DBTABLE** optionally groups segments into DBNAMEs. A **DBNAME** enables multiple segments to be accessed together as a single database. A DMCL identifies the DBTABLE with which it is associated and like DMCLs, DBTABLEs are generated into load modules for use at runtime.

Segments

A segment represents a physical instance of all or a portion of a database whose logical contents are described by a schema. A segment defines the areas and files that contain the data for one database instance.

At Commonweather, for example, two database instances exist: one for testing and one for production. A single schema describes the logical contents of both databases, but two segments are defined to hold the data: the TESTHR segment identifies the physical characteristics of the test database and the PRODHR segment identifies the physical characteristics of the production database.

Areas and Pages

The physical characteristics of the areas identified in the schema are defined in a segment. An area contains record occurrences stored in blocks, or **pages**, of the database. A page is a block of data whose format is dictated by CA IDMS. When data is read from or written to a database, an entire page (or block) of data is transferred at one time.

An area definition specifies the page size and number of pages in the area. Each page within an area is consecutively numbered beginning with the starting page number specified in the area's definition. The following rules apply when defining an area:

- An area can only contain sequentially numbered pages.
- Gaps in the page numbers may occur between areas.
- All areas within a segment must have non-overlapping page ranges.
- All pages in one area must be the same size. (Page sizes can vary from area to area.)

Both the test and production databases at Commonweather, for example, are divided into three areas:

- ORG-DEMO-REGION- Contains organizational information. The JOB, DEPARTMENT, OFFICE, and SKILL record types are assigned to this area. Within the TESTHR segment, the area contains pages 2001 through 3500.
- EMP-DEMO-REGION- Contains all employee-related information except for insurance. The EMPLOYEE, EMPOSITION, EXPERTISE, and STRUCTURE record types are assigned to this area. Within the TESTHR segment, the area contains pages 6001 through 8000.
- INS-DEMO-REGION- Contains insurance-related information. Occurrences of the INSURANCE-PLAN, COVERAGE, HOSPITAL-CLAIM, NON-HOSP-CLAIM, and DENTAL-CLAIM record types reside in this area. Within the TESTHR segment, the area contains pages 10001 through 10500.

The TESTHR database contains a total of 4000 pages among its three areas.

Files

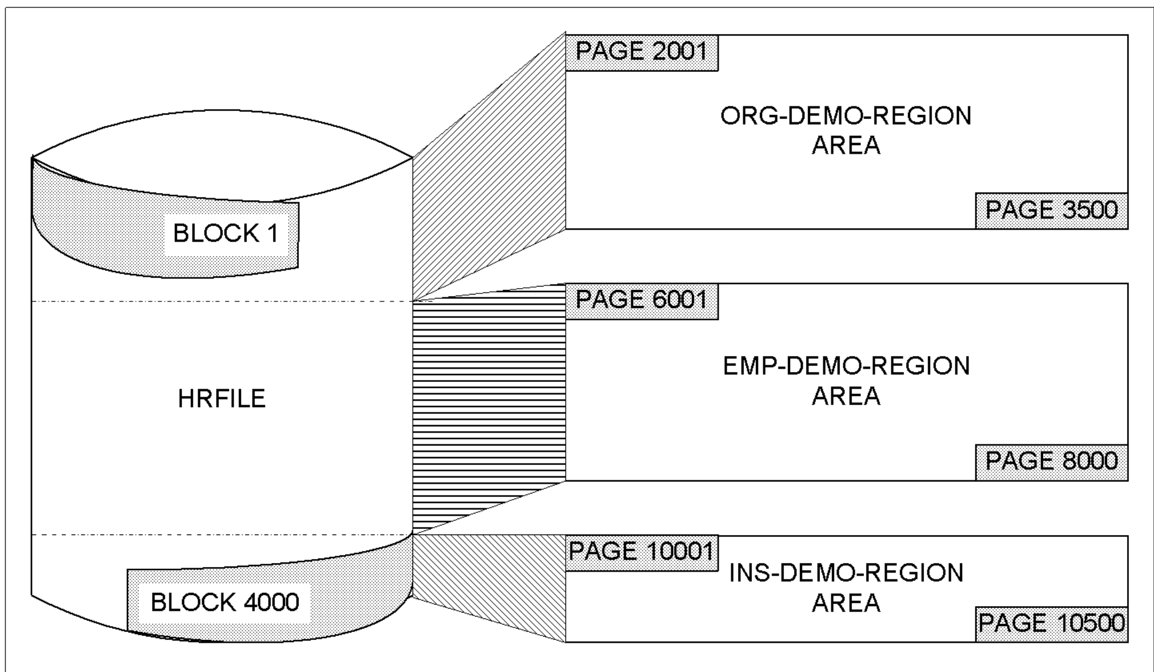
The database is stored as one or more files (data sets) on direct access devices. These files are formatted into a number of BDAM blocks or VSAM control intervals. Each direct access block corresponds to a database page so that data transfers are always accomplished one page at a time.

Sufficient file space must be allocated for all areas of the database. Areas may be mapped into the file(s) in any manner.

- The entire database may be mapped into one file.
- Each area may be mapped into a different file.
- Many areas may be mapped into one file.
- One area may be mapped into many files.

All areas mapping to a single file must be defined with the same page size, which is the file block size.

The Commonweather TESTHR database, for example, is mapped as one 4000-block file containing all three areas, as illustrated in the following figure. The PRODHR database is instead mapped to three files each containing one area. It would also be possible to map two of the areas to one file and the remaining area to its own file. Large areas are often mapped to several files. The only requirement is that each database page must correspond to one, and only one, direct access block.

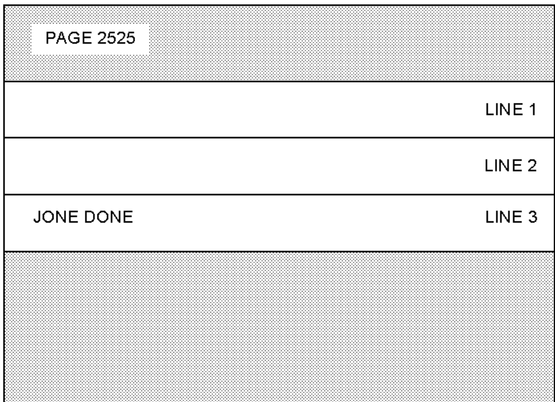


Database Keys (Db-key)

When a record occurrence is stored in the database, it is assigned an identifier, called a **database key**, based on its physical location within the database. The database key remains the same as long as the record remains in the database.

Each page has an identifying **page number**. Within the page, each record occurrence is assigned a unique **line number**. The database key of a record is the combination of the number of the page on which it resides and its line number.

For example, John Done's record occurrence is located on line 3 of page 2525. The database key for his record is 2525/3.



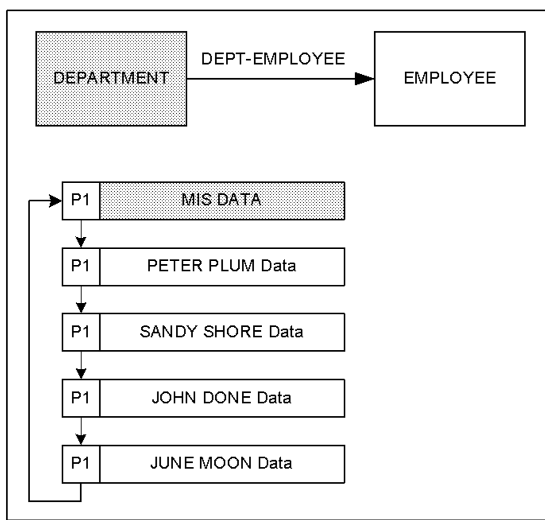
Record Structure

A record occurrence, as it physically appears on a database page, has two parts:

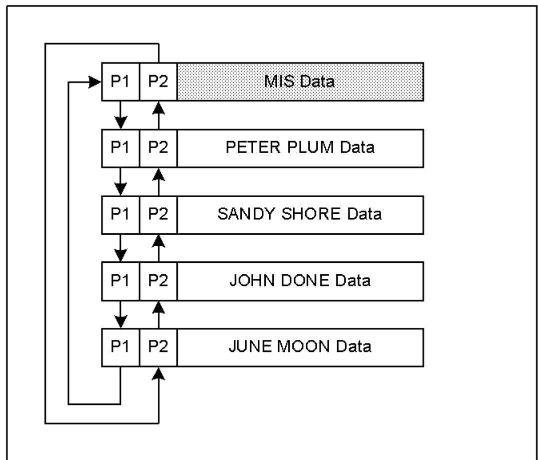
- **Prefix**- An area containing set linkages. The prefix contains the NEXT, PRIOR, OWNER, or INDEX pointers for sets in which the record participates. A pointer is a database key.
- **Data**- The values of the elements that together represent the entity occurrence. The values are in character, binary, floating point, or packed decimal format depending on the data types of the elements defined in the schema's record type definition. Compression can be used to conserve space by substituting codes for repeating data.

A record's prefix is never visible to a program. Only data is returned to a program when a record is accessed.

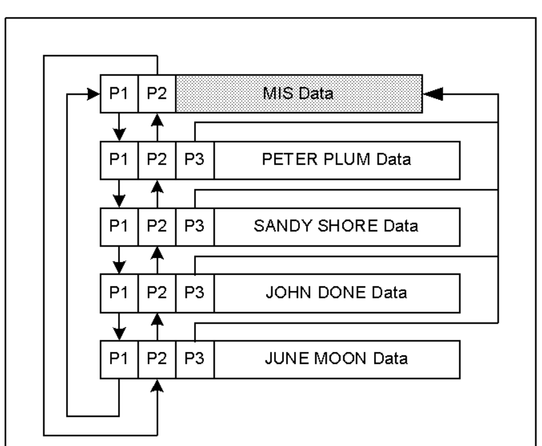
For example, if the DEPARTMENT and EMPLOYEE records participate only in the DEPT-EMPLOYEE set, which uses only NEXT linkage, each record in the set consists of one pointer (to the next record in the set) followed by the value of the record, as illustrated in the following figure.



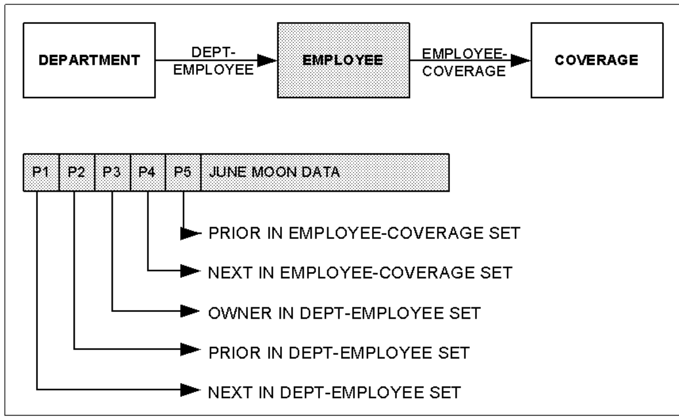
If however, the set also uses PRIOR linkage, each record consists of two pointers plus the data, as shown below.



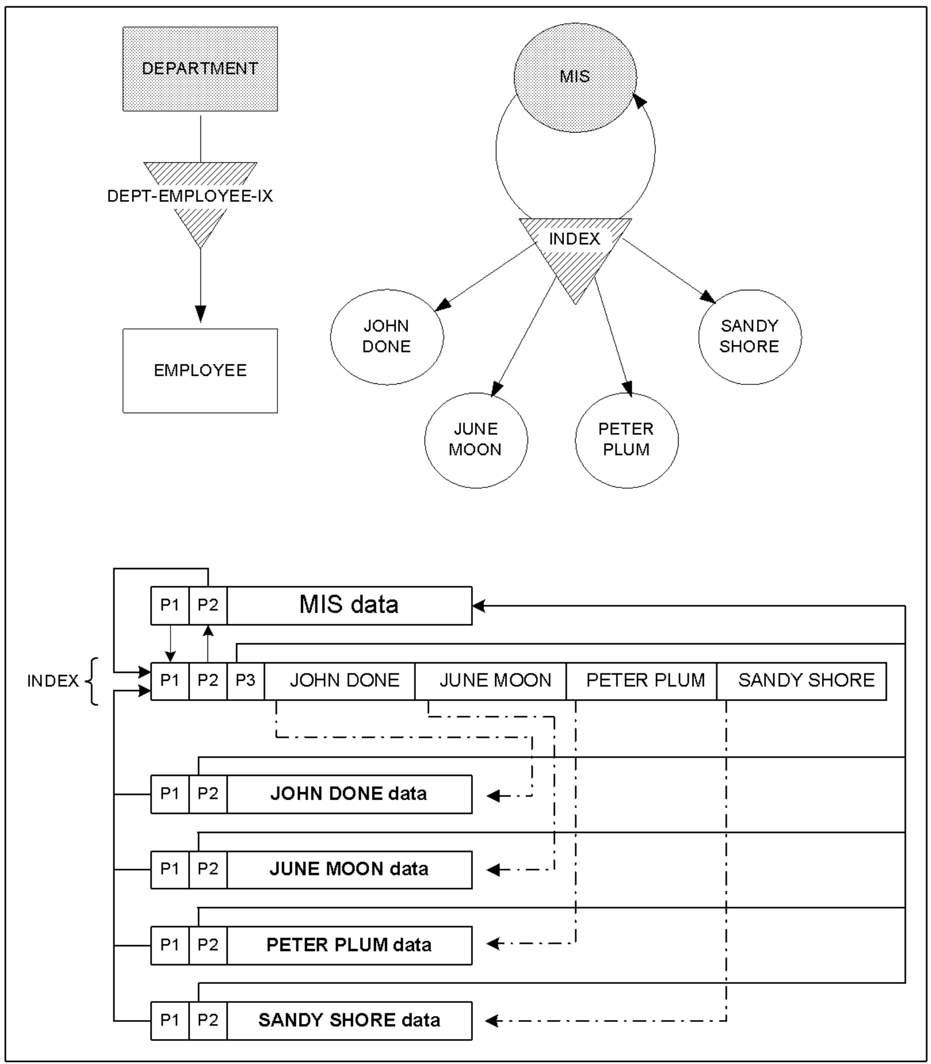
OWNER linkage requires an additional pointer in the prefix of each member record. The owner record does not need an additional pointer.



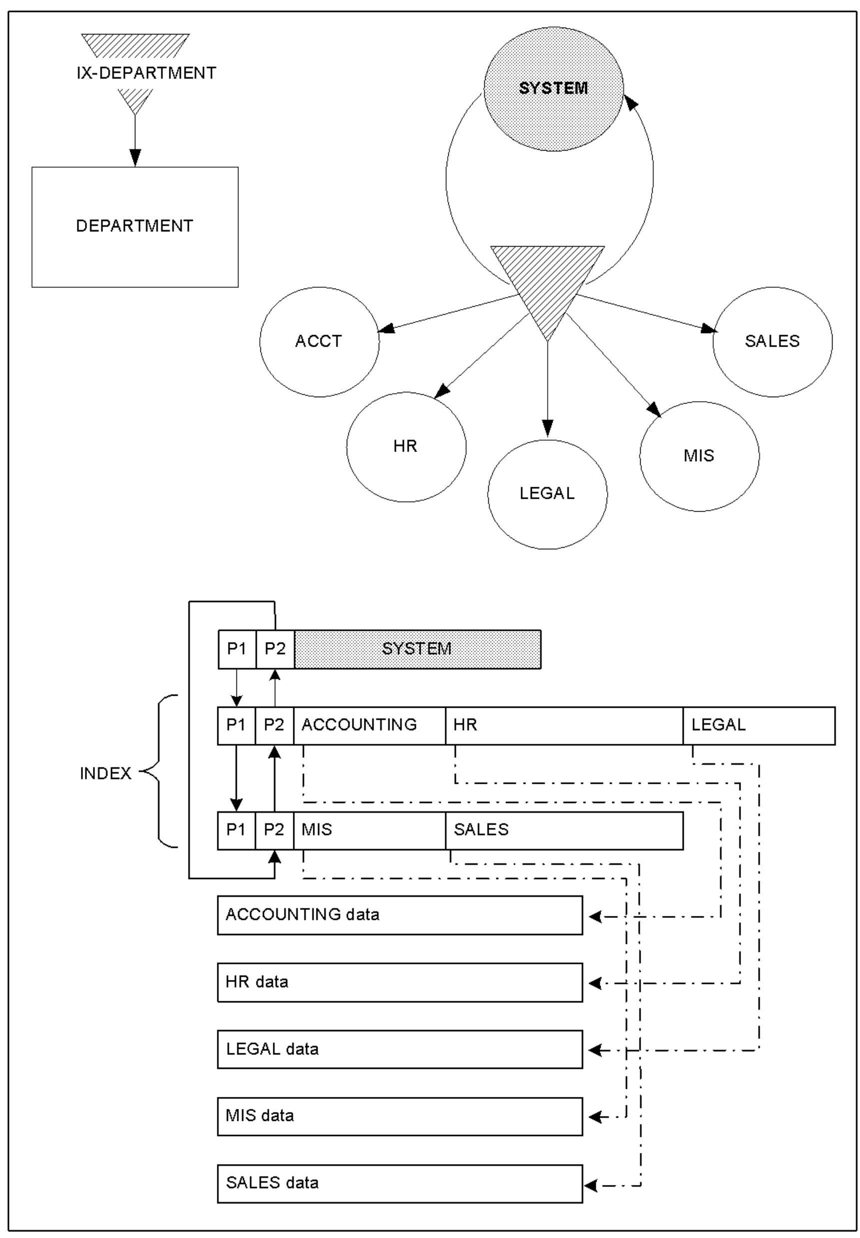
Participation in additional sets requires additional pointers. An EMPLOYEE record that is a member in the DEPT-EMPLOYEE set (with NEXT, PRIOR, and OWNER linkage) and the owner in the EMP-COVERAGE set (with NEXT, PRIOR, and OWNER linkage) requires a total of five pointers, as illustrated in the following figure.



Participation in user-owned index sets also requires pointers. The owner occurrence always contains NEXT and PRIOR pointers to the index structure and member occurrences always contain an INDEX pointer into the index structure. Member occurrences may optionally contain an OWNER pointer to the owner occurrence. The following diagram illustrates the DEPT-EMPLOYEE user-owned index set defined with OWNER linkage.



Unlinked system-owned index sets require no pointers in the indexed record occurrences. Linked system-owned indexes require an INDEX pointer in each member record. System-owned indexes never have OWNER pointers. The following figure illustrates an unlinked system-owned index, DEPARTMENT-IX, defined on the DEPARTMENT record to allow access by name.

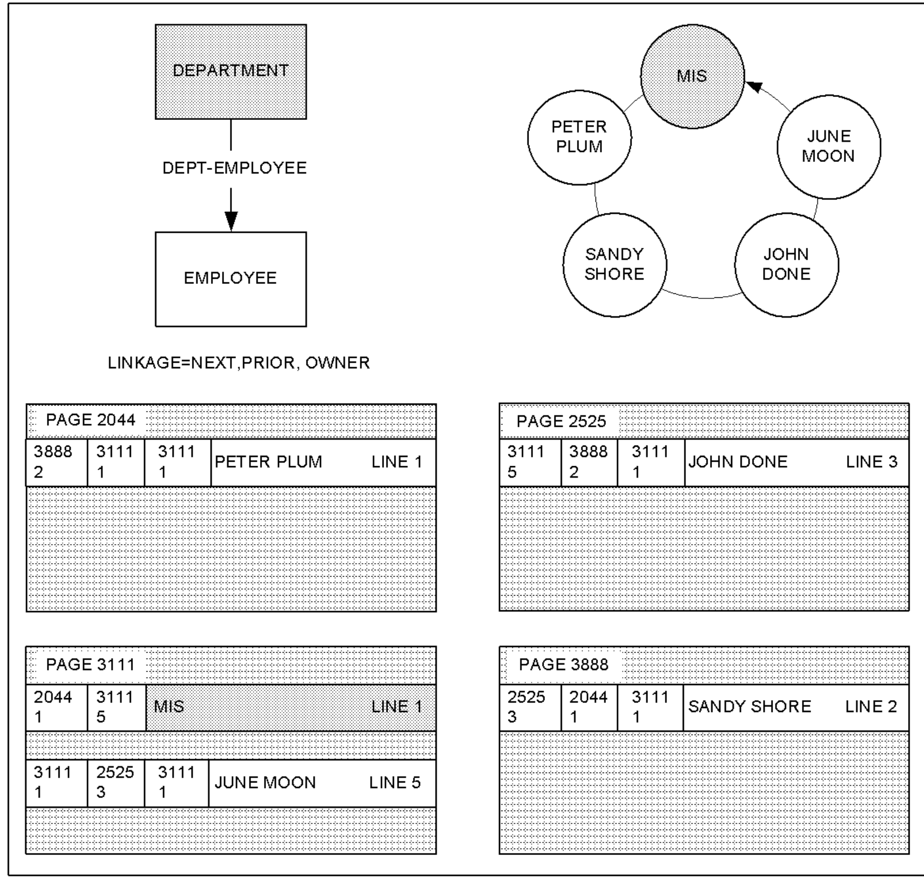


Set Linkage

The physical placement of record occurrences on pages is independent of set relationships. Set relationships are implemented in the database using db-keys as pointers. Each record in a chain set has a four byte binary pointer in its prefix whose value is the page and line number (the db-key) of the next record in the set.

For example, the owner and member records of the DEPT-EMPLOYEE set for MIS are located on four non-contiguous pages, as illustrated in the following figure.

Although the records are not stored next to each other, it is possible to determine the logical structure of the MIS set occurrence by following the pointers. The next pointer for Peter Plum is 3888/2. Sandy Shore is located on page 3888, line 2 so she is the NEXT record in the DEPT-EMPLOYEE set for MIS. The next pointer for Sandy Shore is 2525/3 which leads to John Done's record occurrence. John follows Sandy in the DEPT-EMPLOYEE set.



Location Mode

Location mode is the user-specified method that determines where record occurrences are stored within an area. The following methods are available:

- CALC- A record is stored on or near a page calculated by the database engine (DBMS) using the value of the element(s) comprising the CALC key.
- VIA (Clustering)- A set member record is stored on or near the page containing its owner record occurrence. If the owner and member are assigned to different areas, the member record is stored at the same relative position in its area as the owner record is in its area.
- DIRECT- A record is stored on or near a user-specified page.

The location mode is used to determine the target page for a new record. The new occurrence is placed on the target page if there is sufficient space. If not, DBMS places it on the next page in the area that has sufficient space. If the end of the area is reached without locating a page, the DBMS searches from the beginning of the area for the first page with sufficient space available. If a record cannot be placed on the target page due to space limitations, it is said to **overflow**.

CALC Location Mode

The CALC location mode requires the specification of a CALC key made up of one or more record elements. Usually, a CALC key is chosen so that its value is unique across all record occurrences, although this is not a requirement. At Commonweather, for example, EMPLOYEE records are stored CALC. The EMP-ID, a 4-digit number unique to each employee, is chosen as the CALC key element.

The CALC method is used for:

- Randomization- Records are distributed evenly over all the pages in the area using a randomizing algorithm, thereby minimizing overflow and leaving space for VIA occurrences.
- Direct Retrieval by Symbolic Key- A record occurrence can be retrieved with a single access using its CALC key (rather than by reading all records in an area or searching through an index).

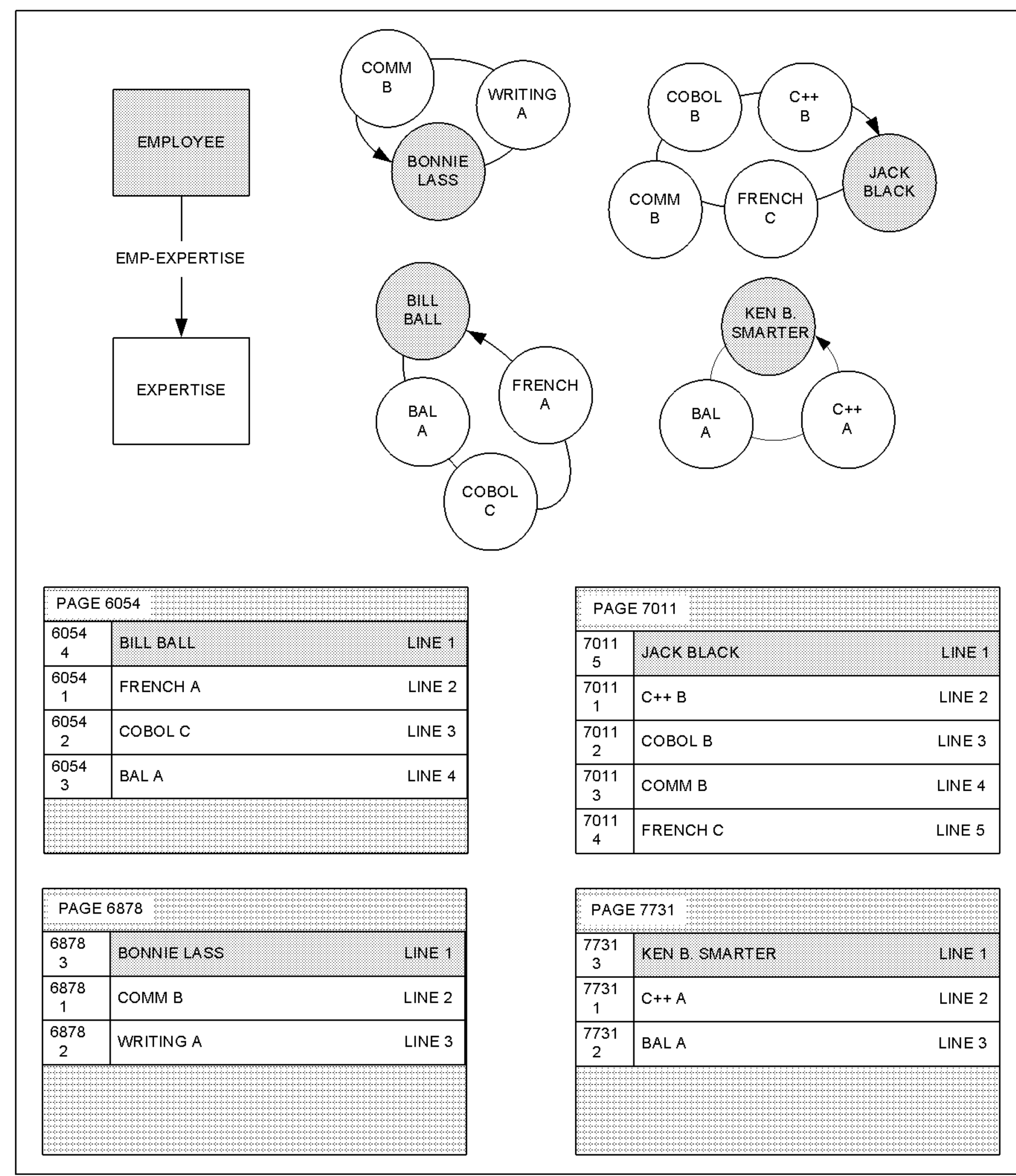
A record whose location mode is CALC can be retrieved by simply specifying the value of its CALC key; DBMS automatically converts the CALC key into the same target page member as that used to store the record. If the CALC record had overflowed to another page when it was added to the database, the DBMS can locate it using an internally maintained set called a CALC set whose owner resides on the record's original target page.

VIA Location Mode

The VIA location mode is used to group (cluster) records that are likely to be accessed together on the same page or as close to each other as possible. This location mode is also sometimes referred to as **clustering**.

At Commonweather, for example, employee expertise is usually retrieved in conjunction with information about an employee. Therefore, EXPERTISE records are stored VIA the EMP-EXPERTISE set so that they will be clustered around their owning EMPLOYEE record.

In the following figure, the EMPLOYEE records are randomly located on four pages using the CALC location mode. The EXPERTISE records are located on the same pages as their respective owners in the EMP-EXPERTISE set through the VIA location mode. This enables both employee and expertise information to be retrieved with a single page access.



It is also possible to store records via a system-owned index. In this situation, the target page for a new record is the page of the prior record within the index. Consequently, the indexed record occurrences tend to be physically placed within the area in the same order as their entries occur within the index. If the index is sorted on a symbolic key, the record occurrences tend to be physically sequenced in the order of their respective index key values.

DIRECT Location Mode

DIRECT location mode allows the user to suggest the page on which to store a record. It is used less frequently tha CALC or VIA because it places the burden of locating records on the application program.

One use of DIRECT mode is to store records serially in an area, which is accomplished by targeting each new record to the page of the most recently stored record. (The database key of the record just stored is returned to the program). The DBMS either stores the record on the suggested target page or on the next page with sufficient space.

Data Structure Diagrams

Databases often become large and complex, and it is useful to depict their contents graphically. A **data structure diagram** describes the record types and sets whose occurrences form the contents of the database. The following sections describe the conventions used in a data structure diagram.

Representing Records Graphically

The following information about each record type is included in the data structure diagram of a database:

- **Record Name**- The name of the record type.
- **Record Identification (Record ID)**-The identification number of the record type. Each record type in the database is assigned a number that is unique across all record types assigned to the same area.

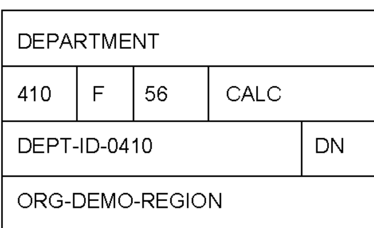
- **Record Format**- The format of occurrences of the record type:
  - Fixed (F)- all occurrences have the same length
  - Variable (V)- occurrences vary in length
  - Fixed compressed (FC)-all occurrences logically have the same length, but their contents are compressed when stored in the database
  - Variable Compressed (VC)-occurrences vary in length and are compressed

    Note: CA IDMS supplies routines for compressing and decompressing database records. All compressed records are treated internally as variable length even if their record definition appears to be fixed in length. FC (fixed compressed) records are returned to the program as fixed-length records and VC (variable compressed) records are returned as variable-length records.
- **Length**- The actual data length (in bytes) for fixed length records; for variable length records, the maximum or average length of the record.
- **Location Mode**- The location mode for the record type:
  - CALC
  - VIA
  - DIRECT

  See Location Mode for a discussion of these options.

- **CALC Key or VIA Set Name**- For CALC records, the name(s) of the element(s) that form the CALC key; for VIA records, the name of the set through which record placement is determined; for DIRECT records, blank.
- **Duplicates Option**- For CALC records, one of the following options:
  - DN (duplicates not allowed)
  - DF (duplicates first)
  - DL (duplicates last)
- **Area Name**- Name of the area in which the record occurrences are stored.

The DEPARTMENT record type within the Commonweather database is graphically represented by the following figure. From the diagram it can be determined that its location mode is CALC using DEPT-ID-0410 as a CALC key whose values must be unique. It is a fixed length record in which every occurrence contains 56 bytes of data. Its internal numeric identifier is 410 and DEPARTMENT record occurrences are stored in the ORG-DEMO-REGION of the database.



Representing Sets Graphically

The following information about each set type is included in the data structure diagram:

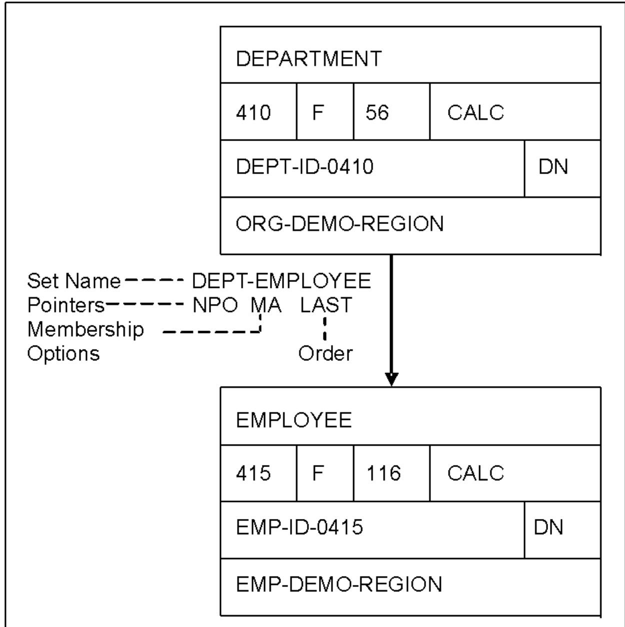
- **Set Name**- Name of the set type
- **Set Linkage**- For chain sets, the set linkage options as:
  - N (next pointers)
  - NP (next and prior pointers)
  - NO (next and owner pointers)
  - NPO (next, prior, and owner pointers)

  For index sets, the set linkage options as:
  - I (index pointers)
  - IO (index and owner pointers)
  - U or blank (unlinked - no pointers)
- **Membership Options**- The disconnect/connect options:
  - MA (mandatory automatic)
  - MM (mandatory manual)
  - OA (optional automatic)
  - OM (optional manual)

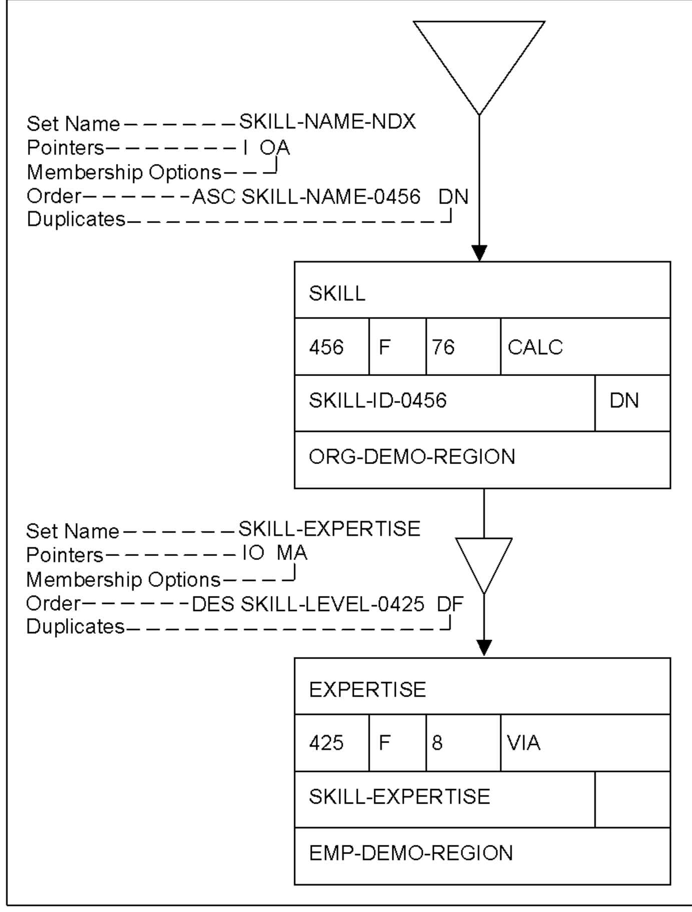
- **Order**- The position in which new records are added to the set
  - FIRST
  - LAST
  - NEXT
  - PRIOR
  - SORTED identifying the ordering sequence (ASC for ascending, DES for Descending), sort or symbolic key element(s) and duplicates options (DN, DF, DL)

For more information about the meaning of the above options, see Sets.

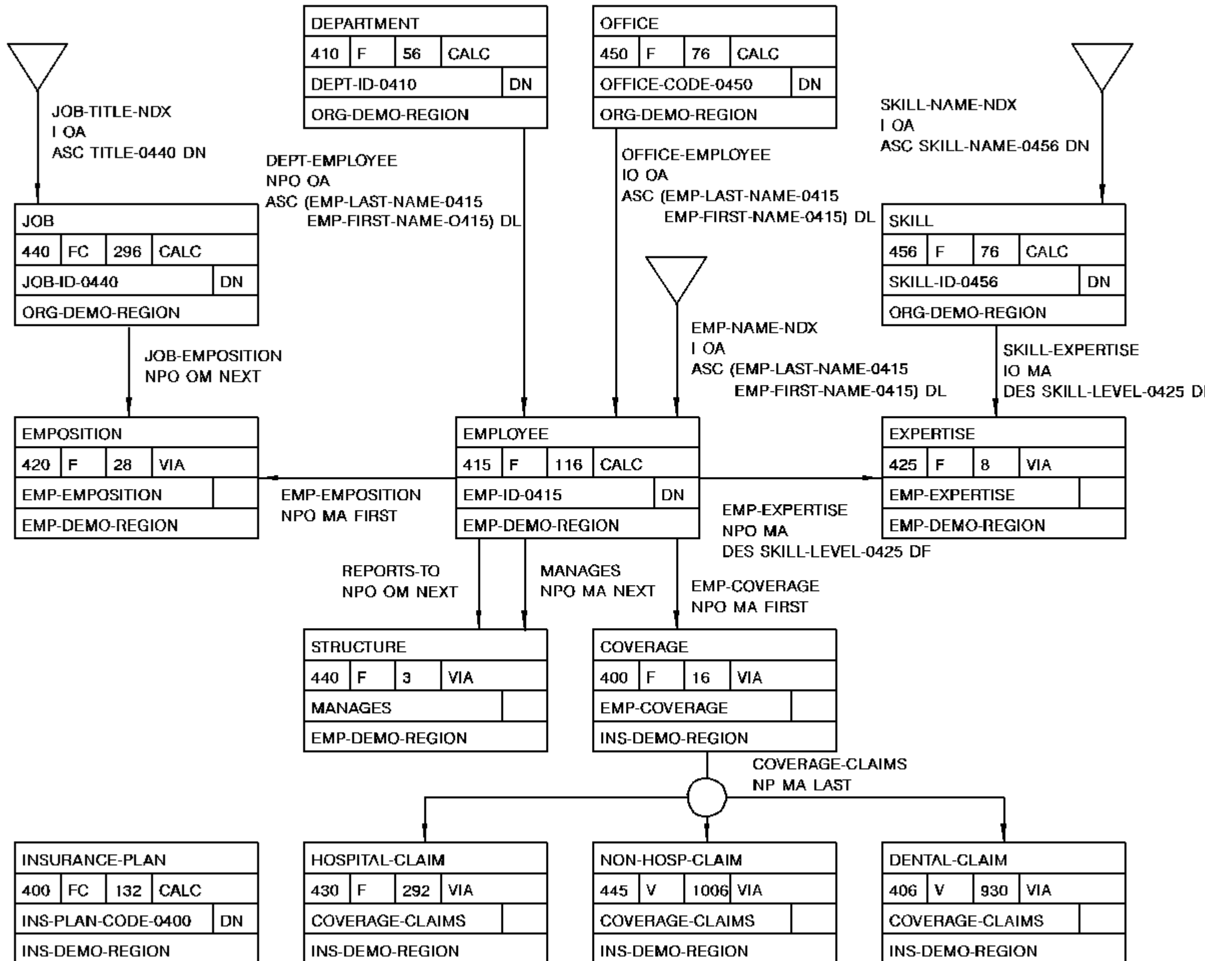
The following figure illustrates the graphical representation of the DEPT-EMPLOYEE chain set. This set is linked through NEXT, PRIOR, and OWNER pointers. It is a Mandatory Automatic set and new members are inserted at the end of all existing members.



The next figure illustrates the graphical representation of the system-owned index SKILL-NAME-NDX and the user-owned index set SKILL-EXPERTISE. Both sets are linked through an INDEX pointer and sorted on an index key: SKILL-NAME-0456 and SKILL-LEVEL-0425 respectively. However, owner pointers are maintained for the user-owned index set SKILL-EXPERTISE and skill names are unique whereas skill levels are not because several employees might have the same level in a given skill. Furthermore, members of the SKILL-EXPERTISE set are maintained in descending skill level sequence and a new record whose skill level matches that of an existing member is inserted logically before the existing member.



The following is the complete data structure diagram for Commonweather Corporation.



Chapter 4: Introduction to Navigational Programming

This chapter introduces navigational programming concepts. It first discusses the basic housekeeping needed to access a CA IDMS database and then describes the functions used to retrieve data, update data, and test certain conditions. This material is an introduction to navigational programming. A more thorough discussion of selected topics is presented in the chapters *Writing a Navigational DML Program*; *Navigational DML Programming Techniques;* and *Run Units, Locks, and Database Transactions*. For detailed information about individual functions, see the language-specific *CA IDMS DML Reference Guide*.

This section contains the following topics:

Housekeeping Functions (see page 65)
Retrieving Data (see page 68)
Updating Data (see page 84)
Testing Set Membership (see page 90)

Housekeeping Functions

In order to access a CA IDMS database using navigational DML, an application program must:

- Establish a session with the database manager
- Bind each database record type to be accessed to a location in variable storage
- Ready areas containing data to be accessed
- Terminate the database session after access is complete

Each of these housekeeping functions is discussed in the following sections.

Establishing a Database Session

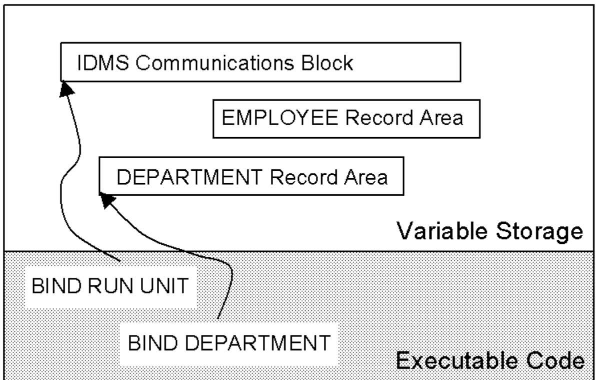
The **BIND RUN UNIT** function initiates a database session, called a run unit, for navigational access to CA IDMS data. A run unit begins with the BIND RUN UNIT function and ends when a FINISH or ROLLBACK function is executed.

In addition to initiating a database session, the BIND RUN UNIT function also establishes addressability to the IDMS communications block, an area in the program's variable storage into which CA IDMS returns information that is pertinent to the services it performs for the program. Among the information returned is a status indicating the success or failure of each executed DML function.

Binding Records

The **BIND record** function establishes addressability to the areas in variable storage that hold or will hold record data. When a record is retrieved from the database, its content is placed into the area to which its record type is bound. Conversely, when a record is stored into the database, the DBMS uses the values that the application has placed into this same area as the record's content. The program allocates the storage areas and then informs the DBMS of their location using the BIND record function.

Suppose, for example, that a program is written to list the departments and employees at Commonweather. As shown in the following figure, the program reserves variable storage for the DEPARTMENT and EMPLOYEE record types and the IDMS Communications block.



Each record accessed by a program must be bound to some area in variable storage.

Readying Areas

The **READY** function informs CA IDMS which areas of the database will be accessed by the application program and in which of the following usage modes:

- **RETRIEVAL**- Records in the area can be retrieved only.
- **UPDATE**- Records in the area can be retrieved and updated.

When running under central version, specifying one of the following qualifiers controls concurrent use of the area by other applications:

- **PROTECTED**- Prevents other applications from updating data.
- **EXCLUSIVE**- Prevents other applications from accessing the area for both retrieval and update operations.

If neither PROTECTED nor EXCLUSIVE is specified, the system assumes that the area is shared and concurrent access of any kind is acceptable.

Terminating a Database Session

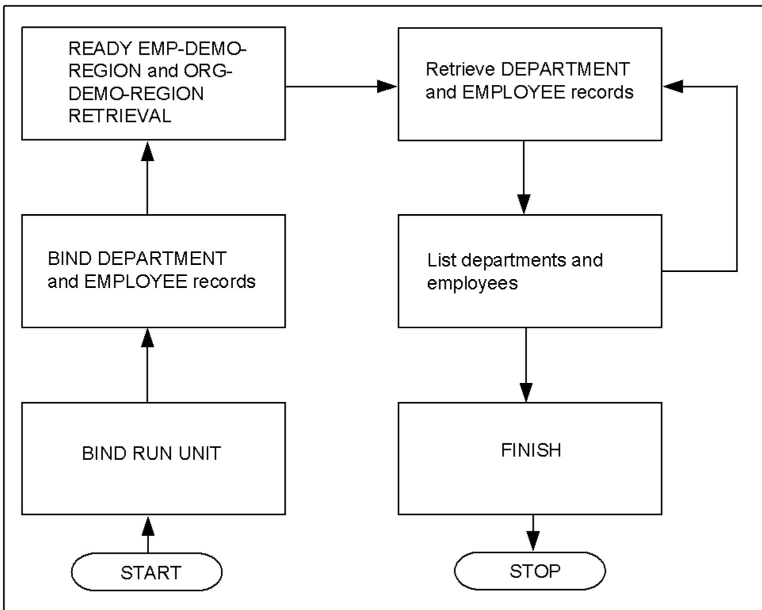
Once the application program has completed its work, it must end the database session. It does so by using either a FINISH or ROLLBACK function.

The **FINISH** function commits (makes permanent) all changes made to the database by the application, releases use of the database areas, and terminates the run unit. A FINISH function is used to end an application's successful use of the database.

The **ROLLBACK** function rolls out (reverses) all changes made to the database and then terminates the run unit. It is used to end an application's use of the database when some type of error is detected.

Function Execution Sequence

The cycle of binding, readying, accessing, and finishing can repeat any number of times during a program's execution. The following figure illustrates the housekeeping functions necessary to read and list the departments and employees at Commonweather.



Retrieving Data

Using navigational DML, you access database records one record at a time using retrieval functions to locate record occurrences and return their contents.

There are several ways of retrieving data using navigational DML. For example, an application program can retrieve every record occurrence in an area or it can retrieve records that participate in a set occurrence or that have a specific key value.

Many retrieval operations rely on an application's current position within the database, that is, its most recently accessed record. For example, when retrieving the next EMPLOYEE record occurrence within an area, the DBMS bases the operation on the EMPLOYEE record occurrence last accessed. This notion of currency is described in the next section followed by a description of the types of retrieval functions that are available and how they are used to access data.

Currency

To enable a program to retrieve data efficiently, CA IDMS saves the database keys of the most recently accessed records, known as currencies. There are four types of currencies maintained by the DBMS:

- **Current of Run Unit**- The most recent record occurrences accessed by the user program.
- **Current of Record Type**- The most recent record occurrence of each record type accessed by the user program.
- **Current of Set**- The most recent record occurrence (owner or member) in each set accessed by the user program.
- **Current of Area**- The most recent record occurrence in each area accessed by the user program.

Current of Run Unit

The record occurrence that was the target of the most recent retrieval or update function is current of run unit. Only one record occurrence is current of run unit at any given time during program execution.

Current of Record Type

The most recently accessed occurrence of each record type is current of that record type. At any given time during program execution, one current record can exist for each record type. For example, when your program successfully retrieves JOB Programmer, that record becomes current of the JOB record type. If you then successfully obtain EMPLOYEE John Done, that record becomes current of the EMPLOYEE record type; currency for the JOB record type remains unchanged.

Current of Set

The most recently accessed record occurrence in each set is current of that set. At any given time during program execution, one current record can exist for each set type.

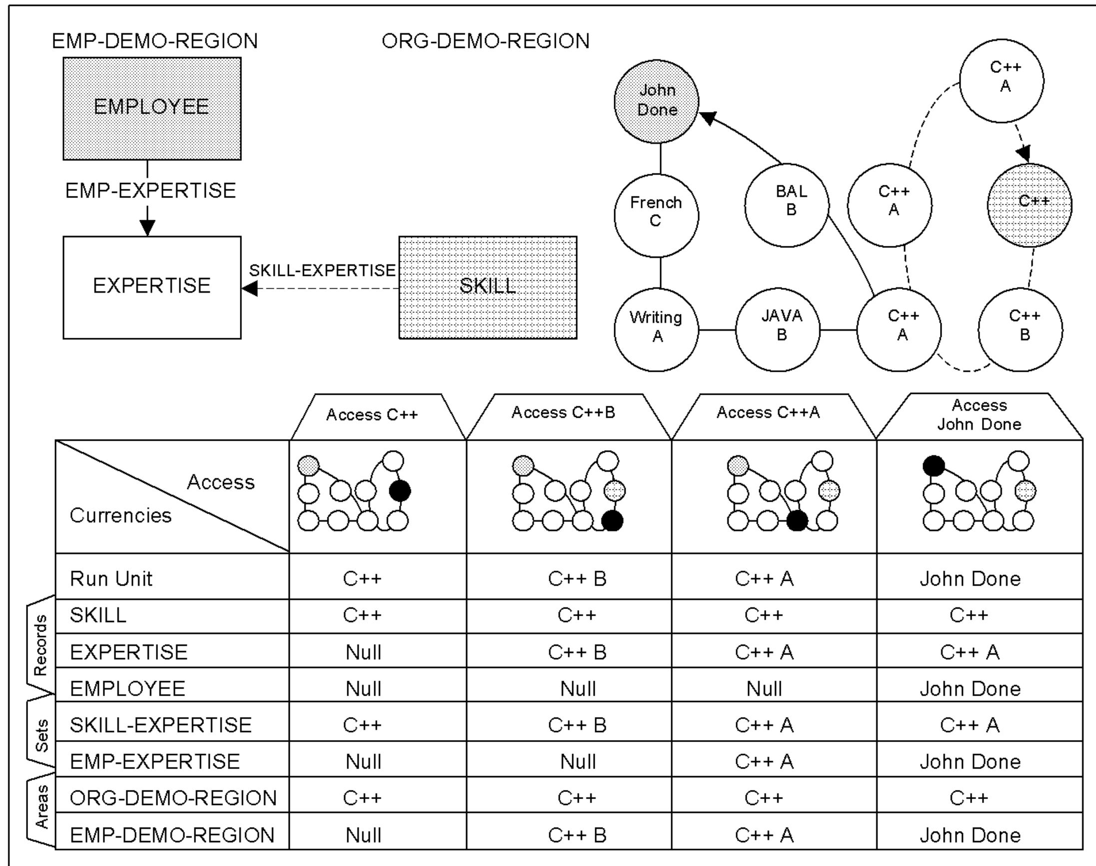
Because a successfully accessed record becomes the current record of all sets in which it participates as either owner or member, a given record occurrence can be the current record of any number of sets.

Current of Area

The most recently accessed record occurrence in each area is current of area for that area. At any given time during program execution, one record can be current for each area.

The following figure illustrates how currencies change as different records are accessed in the sample database. All currencies begin as null.

C++, C++ B, C++ A, and John Done are accessed sequentially. As we access a record, it becomes current of run unit, its record type, its area, and all sets in which it participates. C++ A becomes current of both SKILL-EXPERTISE and EMP-EXPERTISE sets. When we access John Done, it becomes current of the EMP-EXPERTISE set while C++ A remains current of the SKILL-EXPERTISE set.



Access / Currencies	Access C++	Access C++B	Access C++A	Access John Done
Run Unit	C++	C++ B	C++ A	John Done
SKILL	C++	C++	C++	C++
EXPERTISE	Null	C++ B	C++ A	C++ A
EMPLOYEE	Null	Null	Null	John Done
SKILL-EXPERTISE	C++	C++ B	C++ A	C++ A
EMP-EXPERTISE	Null	Null	C++ A	John Done
ORG-DEMO-REGION	C++	C++	C++	C++
EMP-DEMO-REGION	Null	C++ B	C++ A	John Done

Retrieval Functions

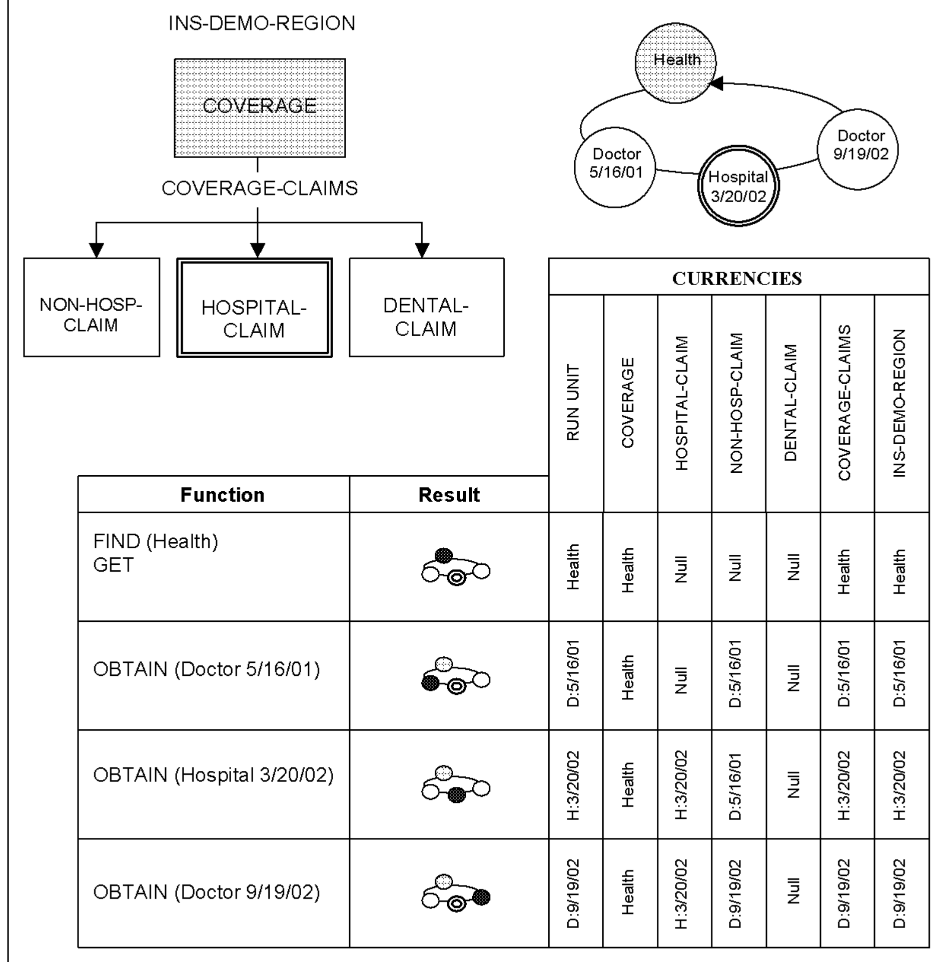
Data retrieval is performed by the following navigational DML functions:

- **FIND**- Locates a record in the database. The located record becomes current of run unit, current of its record type, current of its area, and current of all sets in which it participates.
- **GET**- Retrieves the most recently located record (that is, the record that is current of run unit) by transferring its contents from the database to the program's variable storage.

- **OBTAIN**- Combines the FIND and GET functions so data is located and retrieved in one operation.
- **RETURN**- Retrieves the database key and/or symbolic key values from an index set without accessing the database record.

The following figure illustrates how information on John Done's health coverage at Commonweather is made available to the user program using a combination of OBTAIN, FIND, and GET functions. The following steps are taken:

1. John Done's health coverage record is located and separately transferred to the variable storage area to which the COVERAGE record type is bound.
2. The next record in this occurrence of the COVERAGE-CLAIMS set is located and transferred in a single operation.
3. Step 2 is repeated until all the claim records are processed.



		CURRENCIES						
Function	**Result**	RUN UNIT	COVERAGE	HOSPITAL-CLAIM	NON-HOSP-CLAIM	DENTAL-CLAIM	COVERAGE-CLAIMS	INS-DEMO-REGION
FIND (Health) GET		Health	Health	Null	Null	Null	Health	Health
OBTAIN (Doctor 5/16/01)		D:5/16/01	Health	Null	D:5/16/01	Null	D:5/16/01	D:5/16/01
OBTAIN (Hospital 3/20/02)		H:3/20/02	Health	H:3/20/02	D:5/16/01	Null	H:3/20/02	H:3/20/02
OBTAIN (Doctor 9/19/02)		D:9/19/02	Health	H:3/20/02	D:9/19/02	Null	D:9/19/02	D:9/19/02

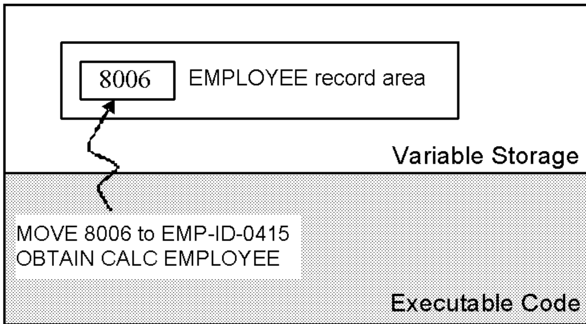
Locating a record through either the FIND or OBTAIN function uses a technique identified by one of the following options:

- **CALC**- Locates a record directly, based on a symbolic key value, the CALC key, in the record.
- **WITHIN SET**- Locates a record based on its logical relationships with other records.
- **WITHIN AREA**- Locates a record directly or relatively, based on its physical location in the database.
- **DB-KEY**- Locates a record directly based on its database key.
- **CURRENT**- Locates a record directly based on existing currencies.

CALC

The FIND/OBTAIN CALC function locates a record that has been stored using the CALC location mode based on the value of its CALC key. Locating records using FIND/OBTAIN CALC is the most common method of entering the database to process a transaction.

The following figure illustrates retrieval of the BILL BALL record using its CALC key. First, we place the value of its CALC key, 8006, in the EMP-ID-0415 field of the EMPLOYEE record type then issue the OBTAIN CALC function, specifying EMPLOYEE as an argument.



WITHIN SET

Once entry is made into the database, the user program can locate additional records by following set linkages. The FIND/OBTAIN WITHIN SET function provides the following options for this purpose:

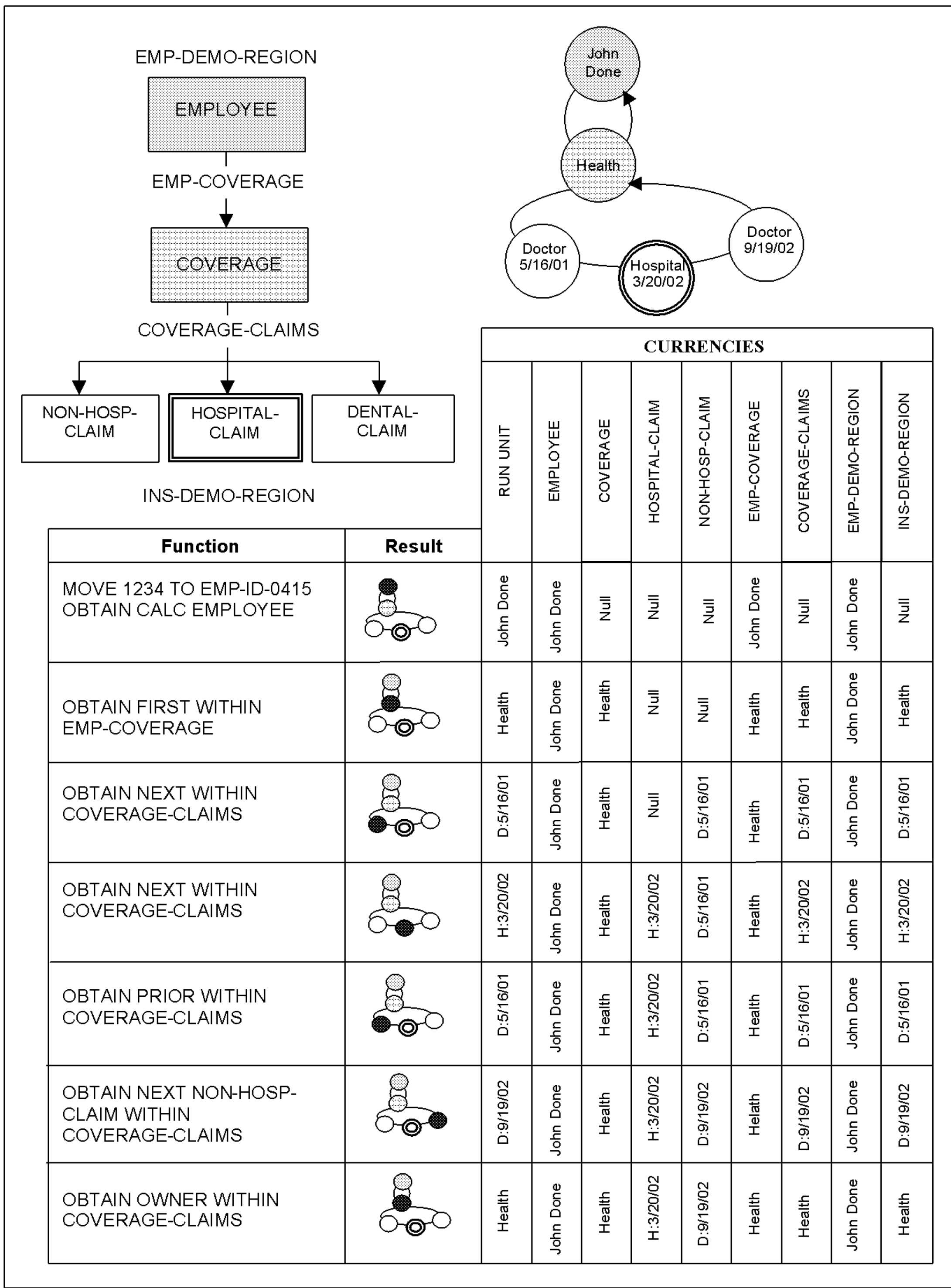
- **FIRST**- Locates the first member record in a set.
- **LAST**- Locates the last member record in a set (prior linkage required).
- **Nth**- Locates the *n*th member record in a set.
- **NEXT**- Locates the next record in a set.
- **PRIOR**- Locates the prior record in a set (prior linkage required).

- **OWNER**- Locates the owner record in a set (owner linkage recommended but not required).
- **USING**- Locates a record in a set by using its sort key value (sorted sets, including index sets, only).

The process of consecutively accessing member records within a set is called **walking a set**.

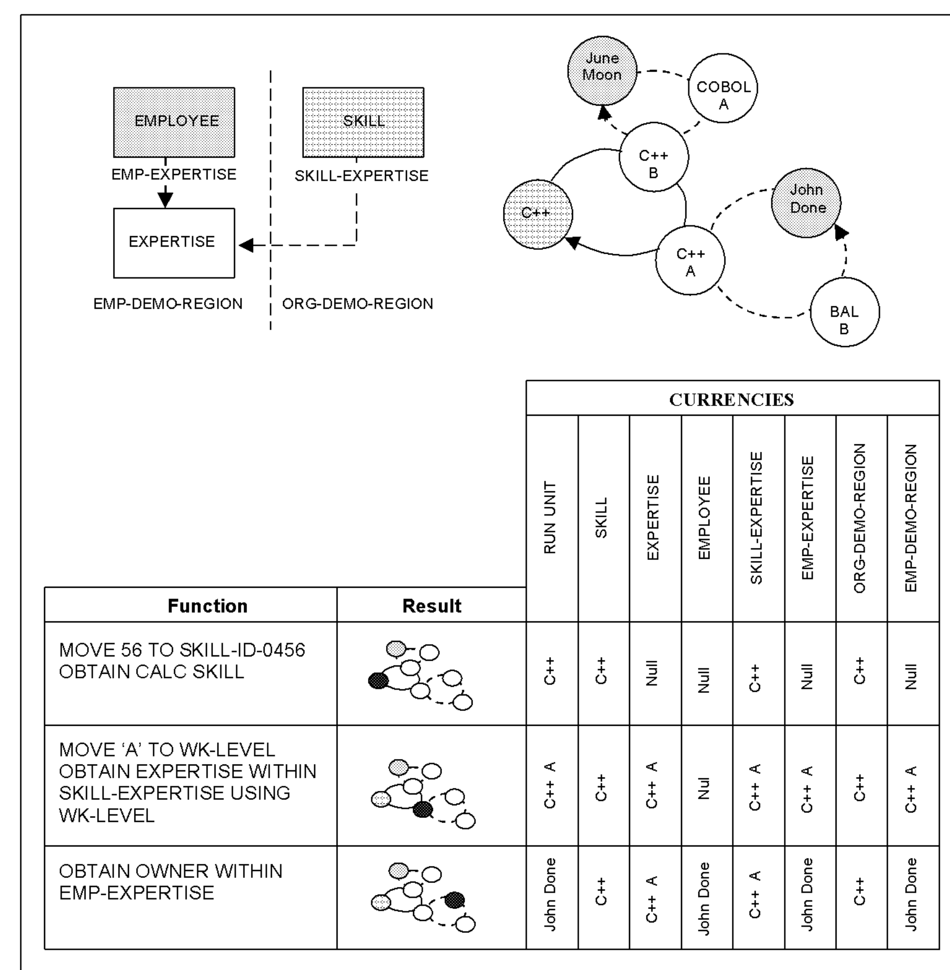
In a set containing multiple record types (for example, HOSPITAL-CLAIM, NON-HOSPITAL-CLAIM, and DENTAL-CLAIM which are all members of the COVERAGE-CLAIMS set), records can be located independently of record type, or the function can be limited to records of a specified record type.

The following figure illustrates various WITHIN SET options. Using our Commonweather example, we enter the database on John Done through the OBTAIN CALC function. We then use the OBTAIN WITHIN SET function to locate the first record in his EMP-COVERAGE set and then to locate various records in its COVERAGE-CLAIMS set.



Function	Result	RUN UNIT	EMPLOYEE	COVERAGE	HOSPITAL-CLAIM	NON-HOSP-CLAIM	EMP-COVERAGE	COVERAGE-CLAIMS	EMP-DEMO-REGION	INS-DEMO-REGION
MOVE 1234 TO EMP-ID-0415 OBTAIN CALC EMPLOYEE		John Done	John Done	Null	Null	Null	John Done	Null	John Done	Null
OBTAIN FIRST WITHIN EMP-COVERAGE		Health	John Done	Health	Null	Null	Health	Health	John Done	Health
OBTAIN NEXT WITHIN COVERAGE-CLAIMS		D:5/16/01	John Done	Health	Null	D:5/16/01	Health	D:5/16/01	John Done	D:5/16/01
OBTAIN NEXT WITHIN COVERAGE-CLAIMS		H:3/20/02	John Done	Health	H:3/20/02	D:5/16/01	Health	H:3/20/02	John Done	H:3/20/02
OBTAIN PRIOR WITHIN COVERAGE-CLAIMS		D:5/16/01	John Done	Health	H:3/20/02	D:5/16/01	Health	D:5/16/01	John Done	D:5/16/01
OBTAIN NEXT NON-HOSP-CLAIM WITHIN COVERAGE-CLAIMS		D:9/19/02	John Done	Health	H:3/20/02	D:9/19/02	Helath	D:9/19/02	John Done	D:9/19/02
OBTAIN OWNER WITHIN COVERAGE-CLAIMS		Health	John Done	Health	H:3/20/02	D:9/19/02	Health	Health	John Done	Health

In the following figure, the FIND/OBTAIN WITHIN SET function allows us to locate an employee that has at least an A skill rating in C++.



Function	Result	RUN UNIT	SKILL	EXPERTISE	EMPLOYEE	SKILL-EXPERTISE	EMP-EXPERTISE	ORG-DEMO-REGION	EMP-DEMO-REGION
		CURRENCIES							
MOVE 56 TO SKILL-ID-0456 OBTAIN CALC SKILL		C++	C++	Null	Null	C++	Null	C++	Null
MOVE ‘A’ TO WK-LEVEL OBTAIN EXPERTISE WITHIN SKILL-EXPERTISE USING WK-LEVEL		C++ A	C++	C++ A	Nul	C++ A	C++ A	C++	C++ A
OBTAIN OWNER WITHIN EMP-EXPERTISE		John Done	C++	C++ A	John Done	C++ A	John Done	C++	John Done

WITHIN AREA

User programs can also navigate the database based on the physical location of records in the area. The FIND/OBTAIN WITHIN AREA function provides the following options for this purpose:

- **FIRST**- Locates the first record in the area.
- **LAST**- Locates the last record in an area.

- **Nth**- Locates the *n*th record in an area.
- **NEXT**- Locates the next record in an area.
- **PRIOR**- Locates the prior record in an area.

The process of consecutively accessing records in an area is called an **area sweep**.

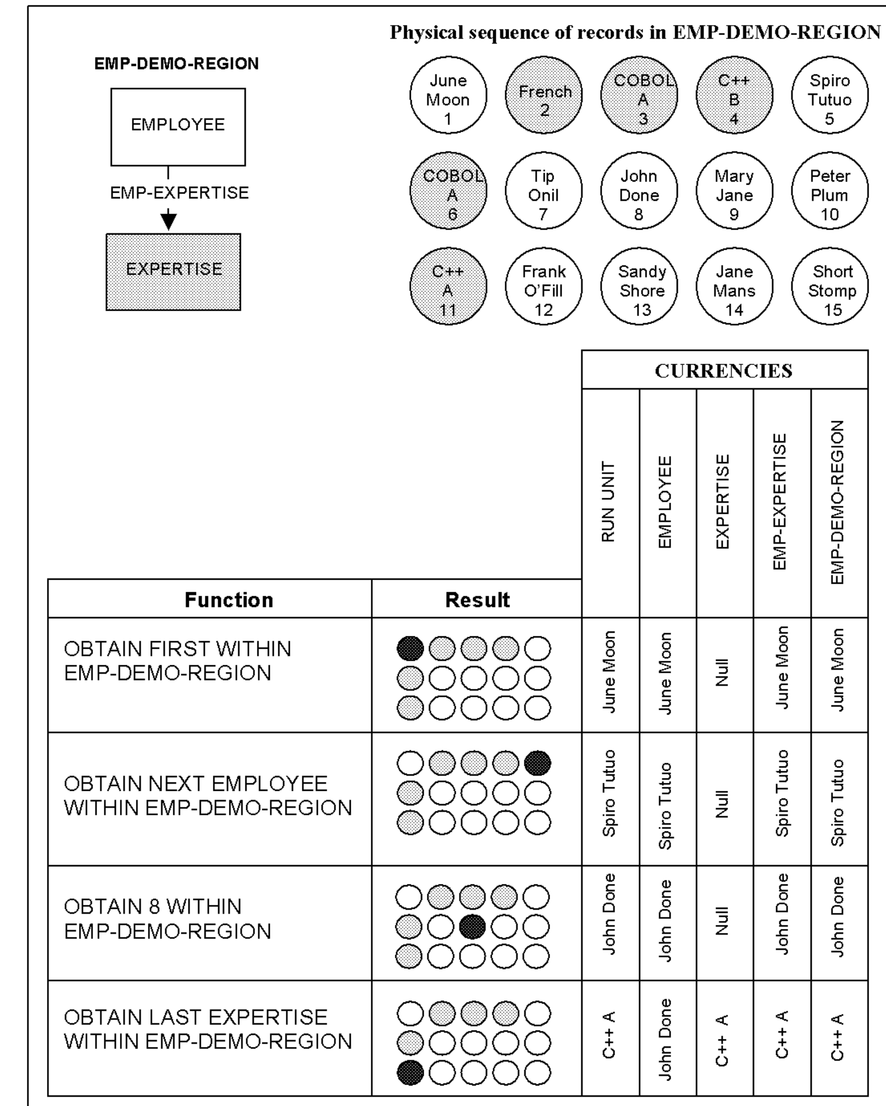
In an area that contains multiple record types, records can be located independently of type, or the function can be limited to records of a specific type.

The FIND/OBTAIN WITHIN AREA function is often the most efficient means of accessing all records of a particular type since each page is processed only once.

For example, we can produce complete reports of all employees like this:

```
OBTAIN FIRST EMPLOYEE WITHIN EMP-DEMO-REGION.
PERFORM REPORT-WRITER.
   ...
Repeat until end of area is reached:
     OBTAIN NEXT EMPLOYEE WITHIN EMP-DEMO-REGION.
     PERFORM REPORT WRITER.
```

The following figure illustrates various OBTAIN WITHIN AREA functions:



DB-KEY and ACCEPT

If the database key of a record is known, the record can be retrieved directly with the FIND/OBTAIN DB-KEY function. If the db-key is not known, the ACCEPT function can be used to retrieve the db-key value before issuing the FIND/OBTAIN DB-KEY.

The ACCEPT function returns a db-key based on some currency. The following ACCEPT function options indicate which currency to use and what db-key value to return:

- **Run-Unit**- Returns the db-key for the current of run unit.
- **Record**- Returns the db-key for the current of a specified record type.

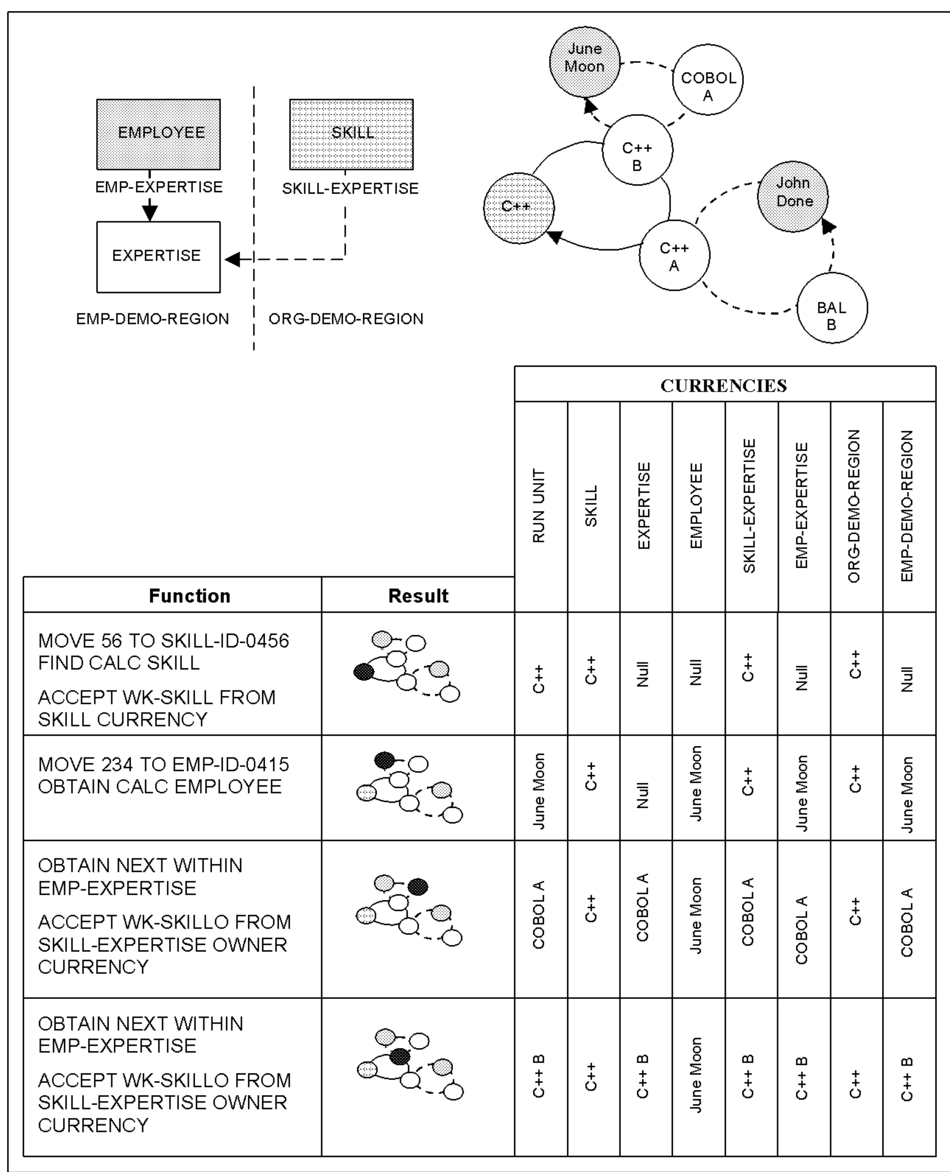
- **Set**- Returns the db-key for the current of a specified set.
  - **NEXT**- Returns the db-key for the record following the current of set.
  - **PRIOR**- Returns the db-key for the record preceding the current of set. (Requires PRIOR pointers.)
  - **OWNER**- Returns the db-key for the owner record of the current of set.
- **Area**- Returns the db-key for the current of a specified area.

Note: Other ACCEPT functions, not discussed here, retrieve run-time statistics and database procedure control areas.

The following figure illustrates the techniques of returning database keys and using them for direct retrieval.

To determine June Moon's expertise rating for C++, we must locate the EXPERTISE record for C++ in her EMP-EXPERTISE set occurrence. This would be easy to do if the skill name were contained as a field in EXPERTISE records; we would CALC to June Moon and walk her EMP-EXPERTISE set examining the skill name in each EXPERTISE record that we accessed. However, to avoid data redundancy, the skill name is not carried in EXPERTISE records, so we will use the ACCEPT function to efficiently accomplish the same thing.

We first determine the db-key of the C++ SKILL occurrence and then walk June Moon's EMP-EXPERTISE set. For each member retrieved, we use the ACCEPT function to retrieve the db-key of the owner in its SKILL-EXPERTISE set and compare the value to that for C++. If they are equal, then we have located June Moon's expertise in C++.



Function	Result	RUN UNIT	SKILL	EXPERTISE	EMPLOYEE	SKILL-EXPERTISE	EMP-EXPERTISE	ORG-DEMO-REGION	EMP-DEMO-REGION
MOVE 56 TO SKILL-ID-0456 FIND CALC SKILL ACCEPT WK-SKILL FROM SKILL CURRENCY		C++	C++	Null	Null	C++	Null	C++	Null
MOVE 234 TO EMP-ID-0415 OBTAIN CALC EMPLOYEE		June Moon	C++	Null	June Moon	C++	June Moon	C++	June Moon
OBTAIN NEXT WITHIN EMP-EXPERTISE ACCEPT WK-SKILLO FROM SKILL-EXPERTISE OWNER CURRENCY		COBOL A	C++	COBOL A	June Moon	COBOL A	COBOL A	C++	COBOL A
OBTAIN NEXT WITHIN EMP-EXPERTISE ACCEPT WK-SKILLO FROM SKILL-EXPERTISE OWNER CURRENCY		C++ B	C++	C++ B	June Moon	C++ B	C++ B	C++	C++ B

CURRENT

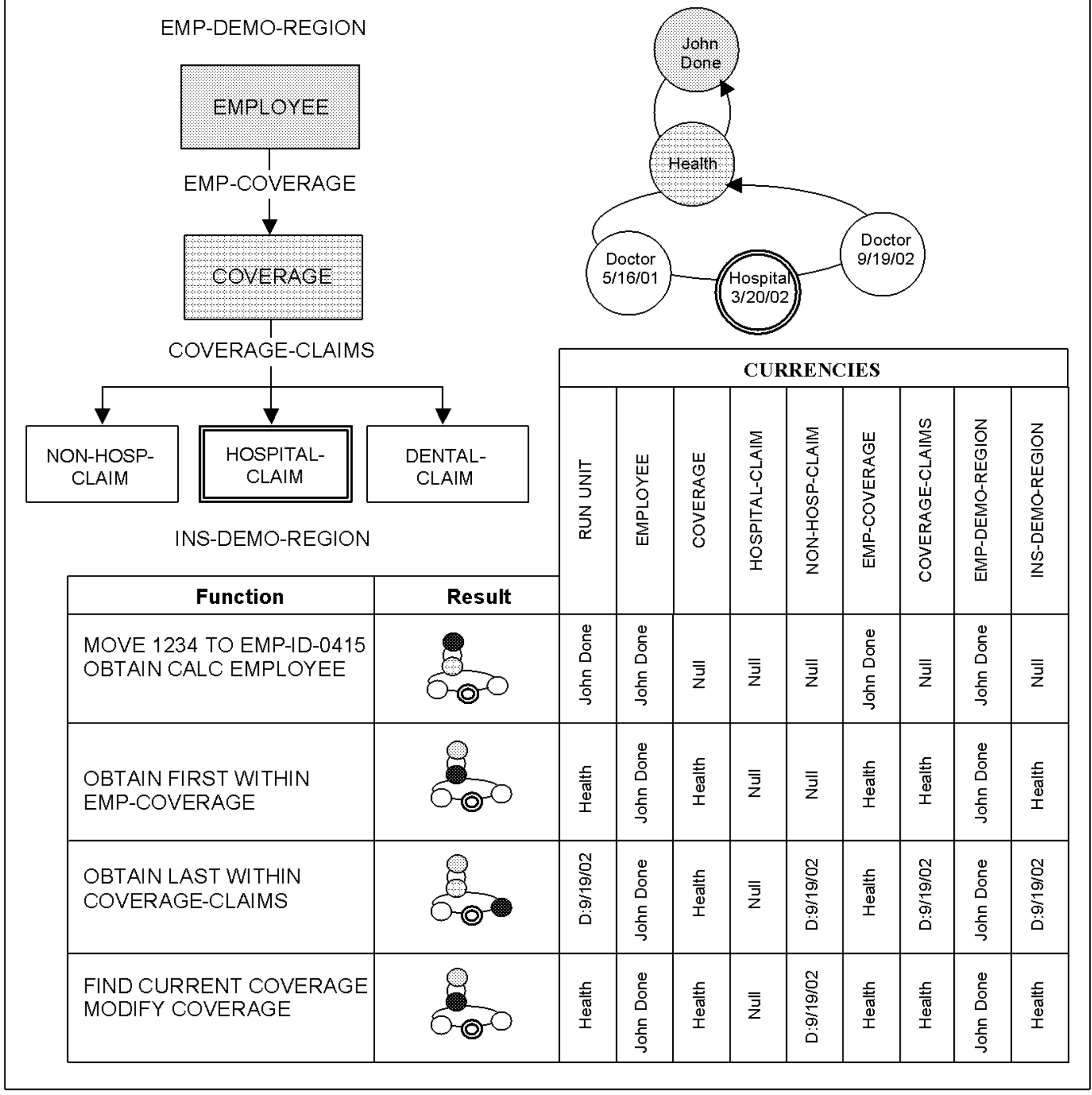
The FIND/OBTAIN CURRENT function uses established currencies to retrieve records within the database. The following currencies can be specified:

- **Run-Unit**- Locates the current of run unit.
- **Record**- Locates the current of record for a specified record type.

- **Set**- Locates the current of set for a specified set type.
- **Area**- Locates the current of area for a specified area.

FIND/OBTAIN CURRENT is frequently used to make a record the current of run unit so it can be modified.

The following figure illustrates repositioning in a database. To update a COVERAGE record after examining its latest claim record, we use the FIND CURRENT function to reposition on the COVERAGE record before issuing the update. We need to do this because, as we shall see later, a record to be updated must be current of run unit.



Function	Result	RUN UNIT	EMPLOYEE	COVERAGE	HOSPITAL-CLAIM	NON-HOSP-CLAIM	EMP-COVERAGE	COVERAGE-CLAIMS	EMP-DEMO-REGION	INS-DEMO-REGION
MOVE 1234 TO EMP-ID-0415 OBTAIN CALC EMPLOYEE		John Done	John Done	Null	Null	Null	John Done	Null	John Done	Null
OBTAIN FIRST WITHIN EMP-COVERAGE		Health	John Done	Health	Null	Null	Health	Health	John Done	Health
OBTAIN LAST WITHIN COVERAGE-CLAIMS		D:9/19/02	John Done	Health	Null	D:9/19/02	Health	D:9/19/02	John Done	D:9/19/02
FIND CURRENT COVERAGE MODIFY COVERAGE		Health	John Done	Health	Null	D:9/19/02	Health	Health	John Done	Health

RETURN

The RETURN function can be used to directly access the index of either user-owned or system-owned index sets.

The structure of an index set differs from a chained set since it is implemented through a pointer array, or index, between the owner record and the member record occurrences. The index contains the db-key and symbolic key of each of the member records and is always located in the same area as the owner of the set, which may be different than the area in which the member records reside.

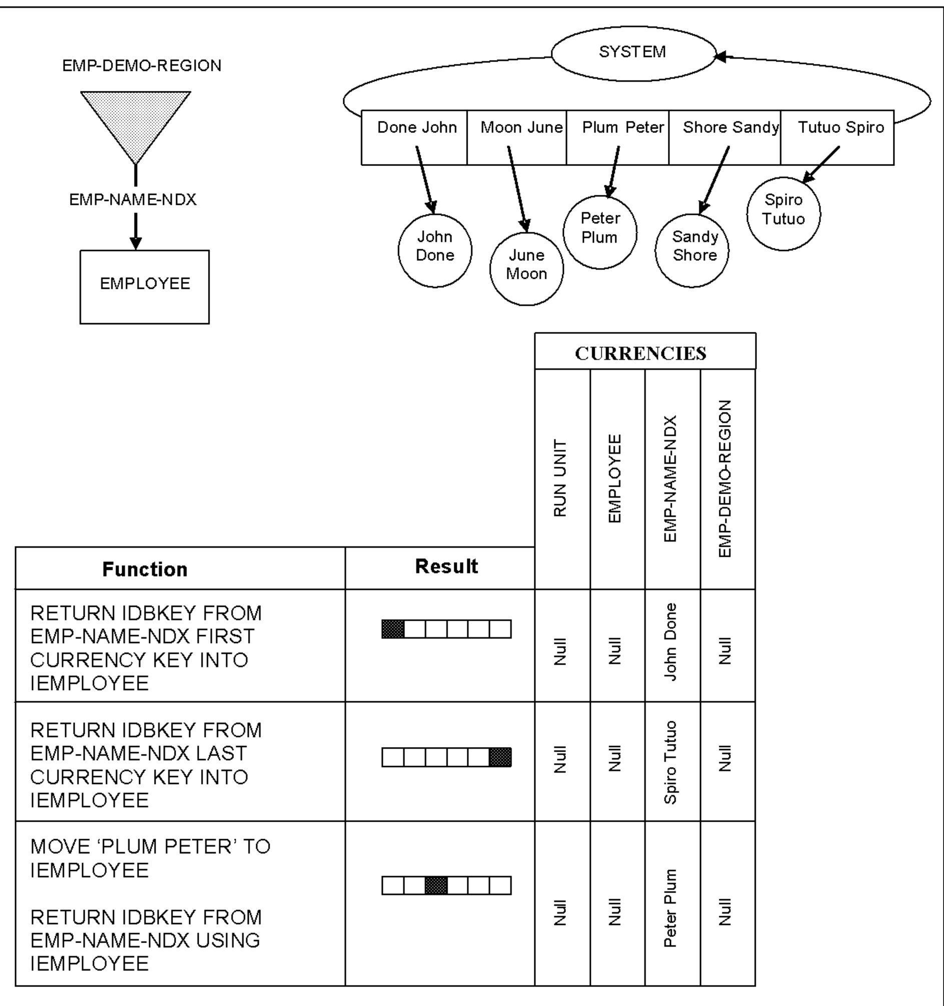
A user program can directly access the index to retrieve the db-key and, optionally, the symbolic key if one is defined, of an index entry using the RETURN function instead of or in addition to using FIND/OBTAIN functions to retrieve the indexed member records from the database.

The RETURN function supports the following options:

- **CURRENCY**- returns the db-key of the current index entry.
- **FIRST CURRENCY**- returns the db-key of the first index entry.
- **LAST CURRENCY**- returns the db-key of the last index entry.
- **NEXT CURRENCY**- returns the db-key for the index entry following current of index.
- **PRIOR CURRENCY**- returns the db-key for the index entry preceding current of index.
- **USING**- returns the db-key for the first index entry matching the specified symbolic key value.

Unlike FIND/OBTAIN requests, RETURN requests affect only the currency of the indexed set and do not affect currency for the run unit or records participating in the set.

In the following figure, the RETURN function is used on the EMP-NAME-NDX indexed set. In the first function, the db-key and symbolic key of the first EMPLOYEE are returned into the variable storage fields IDBKEY and IEMPLOYEE. In the next function, the db-key and symbolic keys of the last EMPLOYEE are returned. In the third function, the db-key of the 'Peter Plum' EMPLOYEE record is returned. Note that in each of the examples, only the index currency is affected.



Function	Result	RUN UNIT	EMPLOYEE	EMP-NAME-NDX	EMP-DEMO-REGION
RETURN IDBKEY FROM EMP-NAME-NDX FIRST CURRENCY KEY INTO IEMPLOYEE		Null	Null	John Done	Null
RETURN IDBKEY FROM EMP-NAME-NDX LAST CURRENCY KEY INTO IEMPLOYEE		Null	Null	Spiro Tutuo	Null
MOVE ‘PLUM PETER’ TO IEMPLOYEE RETURN IDBKEY FROM EMP-NAME-NDX USING IEMPLOYEE		Null	Null	Peter Plum	Null

Updating Data

Once you have located a record and moved it to variable storage, you can update the data and save it back into the database; you can delete it; or you can change the set association of a record. The following functions are used to change the contents of the database:

- STORE
- CONNECT
- MODIFY
- ERASE
- DISCONNECT

STORE

The STORE function creates a new record occurrence. It uses values in the variable storage area to which the record type is bound as the contents of the new record.

In order to use the STORE function, the user program must:

- Set up the data to be stored in the appropriate space in variable storage (that is, the space to which the record type is bound). Values must be supplied for CALC and sort keys.
- Establish correct currencies for all sets whose membership option is Automatic in which the record type participates as a member, if necessary using the FIND function. It is not necessary to establish currency for system-owned index sets, since the DBMS can do this automatically since there is never more than one occurrence of the set.

For each set type in which the record type participates as owner, a new, empty set occurrence is created. The new record occurrence will be automatically connected to each set in which it participates as a member, if the set membership is Automatic. For sets whose membership option is Manual, you must use the CONNECT function to link the new record to an occurrence of the set.

For example, before storing June Moon's EMPLOYEE record, we must first establish currency in the correct DEPT-EMPLOYEE and OFFICE-EMPLOYEE set occurrences because the membership option for both of these is Automatic. The order in which June Moon will be linked within the sets is determined by the set order and discussed under the CONNECT function.

The record, when stored, becomes current of run unit, current of its record type, current of its area, and current of all sets in which it participates.

CONNECT

The CONNECT function explicitly links an existing occurrence of the member record type to other records within a set. This function is used after storing a new record in order to link it into a set whose membership option is Manual. It can also be used to link an existing record into a set after it has been disassociated from another set occurrence.

The user program prepares for a CONNECT function by establishing correct currencies for the member record being connected and for the set occurrence to which it is being connected. If the set order is FIRST, LAST, or SORTED, the current of set can be any record within the set. If the set order is NEXT or PRIOR it may be necessary to establish currency on a specific record before issuing the connect function.

For example, June Moon is joining the MIS department. We will add her EMPLOYEE record to the DEPT-EMPLOYEE set owned by the MIS department. Because the set order is sorted by last and first names, MIS, Peter Plum, Sandy Shore, or John Done can be current of set; June Moon is always linked after John Done because of the relative value of her last name. If the set order was NEXT or PRIOR and a member's position in the set is important, then the program must ensure that the correct record occurrence is current of set.

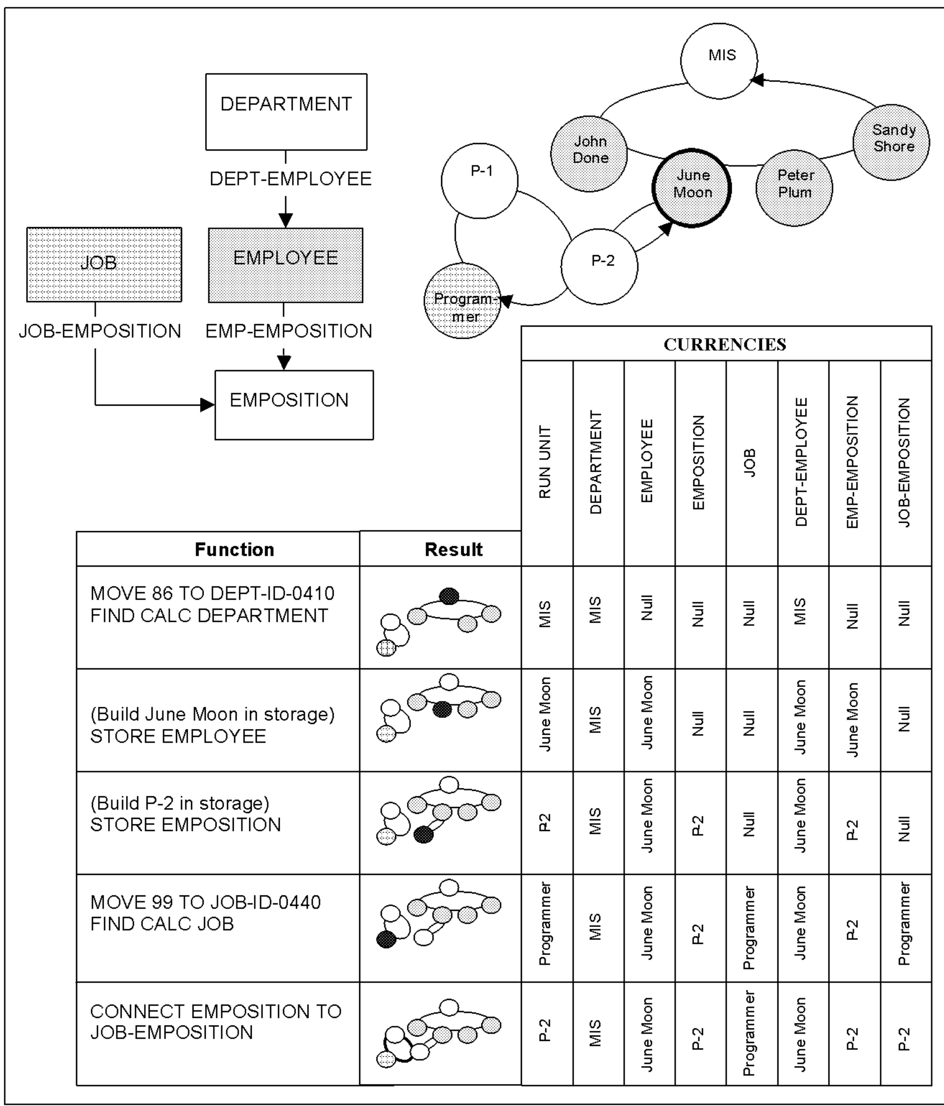
The program must also ensure that a member in the correct set occurrence is current. If Spiro Tutuo, a member of the DEPT-EMPLOYEE set owned by the Sales department, is the current of set when the CONNECT function is issued, then June Moon will appear to be a member of the Sales department rather than the MIS department.

The record, when connected, becomes current of run unit, current of its record type, current of its area, and current of all sets in which it participates.

The following figure illustrates storing and connecting records in our sample database. In this example, we want to add a new employee, June Moon. She will work as a programmer in the MIS department. The example illustrates the following steps:

- Establishing currency in the DEPT-EMPLOYEE set as MIS.
- **Note:** For brevity, we do not show establishing currency in the OFFICE-EMPLOYEE set whose membership option is also automatic.
- Moving the data for June Moon's EMPLOYEE record to its area in variable storage and storing the new EMPLOYEE record occurrence. Since membership in the DEPT-EMPLOYEE and OFFICE-EMPLOYEE sets is Automatic, June Moon is automatically connected into the MIS and Boston set occurrences.
- Moving the data for EMPOSITION to its area within variable storage and storing EMPOSITION. The new EMPOSITION record occurrence is automatically linked into June Moon's set occurrence that was made current as a result of the STORE function.

- Establishing currency in the JOB-EMPOSITION set for Programmer.
- Connecting the new EMPOSITION record into the Programmer occurrence of the JOB-EMPOSITION set.



Function	Result	RUN UNIT	DEPARTMENT	EMPLOYEE	EMPOSITION	JOB	DEPT-EMPLOYEE	EMP-EMPOSITION	JOB-EMPOSITION
		CURRENCIES							
MOVE 86 TO DEPT-ID-0410 FIND CALC DEPARTMENT		MIS	MIS	Null	Null	Null	MIS	Null	Null
(Build June Moon in storage) STORE EMPLOYEE		June Moon	MIS	June Moon	Null	Null	June Moon	June Moon	Null
(Build P-2 in storage) STORE EMPOSITION		P-2	MIS	June Moon	P-2	Null	June Moon	P-2	Null
MOVE 99 TO JOB-ID-0440 FIND CALC JOB		Programmer	MIS	June Moon	P-2	Programmer	June Moon	P-2	Programmer
CONNECT EMPOSITION TO JOB-EMPOSITION		P-2	MIS	June Moon	P-2	Programmer	June Moon	P-2	P-2

MODIFY

The MODIFY function replaces the contents of the record that is current of run unit with the values in the area of variable storage to which its record type is bound. The user program must ensure correct currency by performing some other function (a STORE, GET, OBTAIN, or another MODIFY) on the record prior to issuing the MODIFY function.

The MODIFY function can change the value of any field in a record including CALC, sort, and index key fields. If the value of a sort or index key field is changed, the record is repositioned in the set or index to reflect the new value.

The following diagram illustrates the steps needed to locate and update June Moon's EMPLOYEE record to change her status from temporary (TP) to permanent (P).

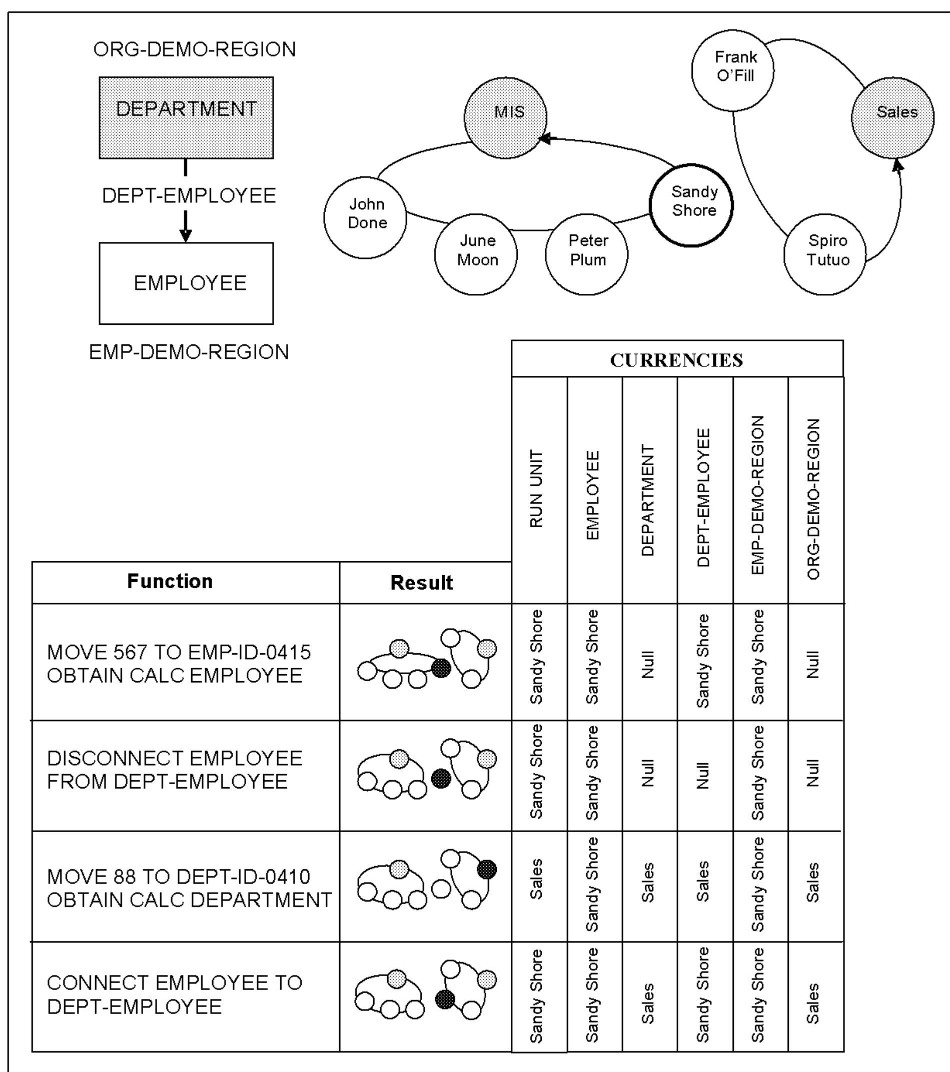
Step	Database	Procedure	Variable Storage
1. Identify the June Moon record	June Moon 234 TP	MOVE 234 TO EMP-ID-0415	234
2. Locate and transfer the record to the area in variable storage to which the EMPLOYEE record type is bound.	June Moon 234 TP	OBTAIN CALC EMPLOYEE	June Moon 234 TP
3. Change the status field in the record area in variable storage.	June Moon 234 TP	MOVE 'P' to STATUS-0415	June Moon 234 P
4. Replace the record on the database with the contents of variable storage.	June Moon 234 P	MODIFY EMPLOYEE	June Moon 234 P

ERASE and DISCONNECT

The ERASE function deletes a record occurrence from the database. The DISCONNECT function removes a member record occurrence from a set without deleting it from the database. You can only issue a DISCONNECT if the set's membership option is Optional rather than Mandatory.

At Commonweather, the EMPLOYEE record is Optional in the DEPT-EMPLOYEE and OFFICE-EMPLOYEE sets to allow for transfers. For example, if Sandy Shore transfers from the MIS to the Sales department we disconnect her record occurrence from the MIS occurrence of the DEPT-EMPLOYEE set and connect it to the Sales occurrence. If the set membership option was Mandatory instead of Optional, we would have to erase Sandy Shore's EMPLOYEE record and re-store it in order to effect the transfer.

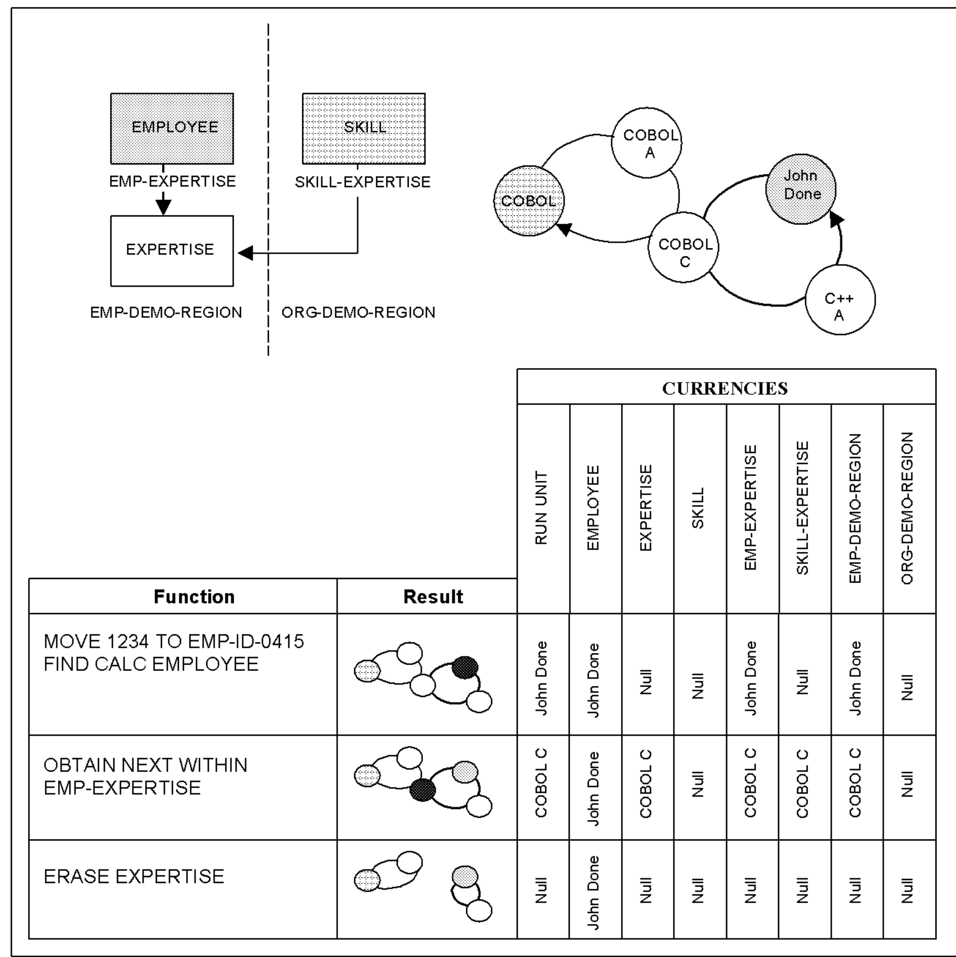
Prior to a DISCONNECT, you must set the current of record type to the record about to be disconnected. After the DISCONNECT, the current of set for the set from which the record was disconnected becomes null, but other currencies associated with the set (such as next and prior) remain unchanged.



Function	Result	RUN UNIT	EMPLOYEE	DEPARTMENT	DEPT-EMPLOYEE	EMP-DEMO-REGION	ORG-DEMO-REGION
MOVE 567 TO EMP-ID-0415 OBTAIN CALC EMPLOYEE		Sandy Shore	Sandy Shore	Null	Sandy Shore	Sandy Shore	Null
DISCONNECT EMPLOYEE FROM DEPT-EMPLOYEE		Sandy Shore	Sandy Shore	Null	Null	Sandy Shore	Null
MOVE 88 TO DEPT-ID-0410 OBTAIN CALC DEPARTMENT		Sales	Sandy Shore	Sales	Sales	Sandy Shore	Sales
CONNECT EMPLOYEE TO DEPT-EMPLOYEE		Sandy Shore	Sandy Shore	Sales	Sandy Shore	Sandy Shore	Sales

The ERASE function disconnects the target record from all sets in which it participates as a member prior to deleting the record from the database. The record must be current of run unit at the time of the ERASE.

By erasing John Done's COBOL expertise in the following figure, we implicitly disconnect it from the two sets in which it is a member: EMP-EXPERTISE and SKILL-EXPERTISE. Because membership is Mandatory in the EMP-EXPERTISE set, we cannot simply disconnect the record without erasing it.



		CURRENCIES							
Function	**Result**	RUN UNIT	EMPLOYEE	EXPERTISE	SKILL	EMP-EXPERTISE	SKILL-EXPERTISE	EMP-DEMO-REGION	ORG-DEMO-REGION
MOVE 1234 TO EMP-ID-0415 FIND CALC EMPLOYEE		John Done	John Done	Null	Null	John Done	Null	John Done	Null
OBTAIN NEXT WITHIN EMP-EXPERTISE		COBOL C	John Done	COBOL C	Null	COBOL C	COBOL C	COBOL C	Null
ERASE EXPERTISE		Null	John Done	Null	Null	Null	Null	Null	Null

Once a record is erased, all currencies for the record and the sets in which it participates become null. (Issuing FIND NEXT WITHIN SET, FIND PRIOR WITHIN SET, FIND NEXT WITHIN AREA, and FIND PRIOR WITHIN AREA functions still locate the correct records.)

A record can be deleted using the ERASE function only if all sets it owns are empty. In our example, we were able to erase John Done's COBOL expertise because it did not own any sets. If we attempt to erase the COBOL SKILL record that owns several EXPERTISE records, an error condition results.

Owners of sets containing members can be deleted using the ERASE function by specifying one of the following options:

- **ERASE ALL**- Erases the owner record and all members.
- **ERASE PERMANENT**- Erases the owner record and all mandatory members. Optional members are disconnected.
-
- **ERASE SELECTIVE**- Erases the owner, all mandatory members, and all optional members that are not connected as members to other sets. Optional members that do have other set linkages are disconnected from the set whose owner is being erased (but not from their other sets).

If a deleted member record is also an owner in another set, its members are treated in a like manner. All members must be accounted for before deleting an owner.

Testing Set Membership

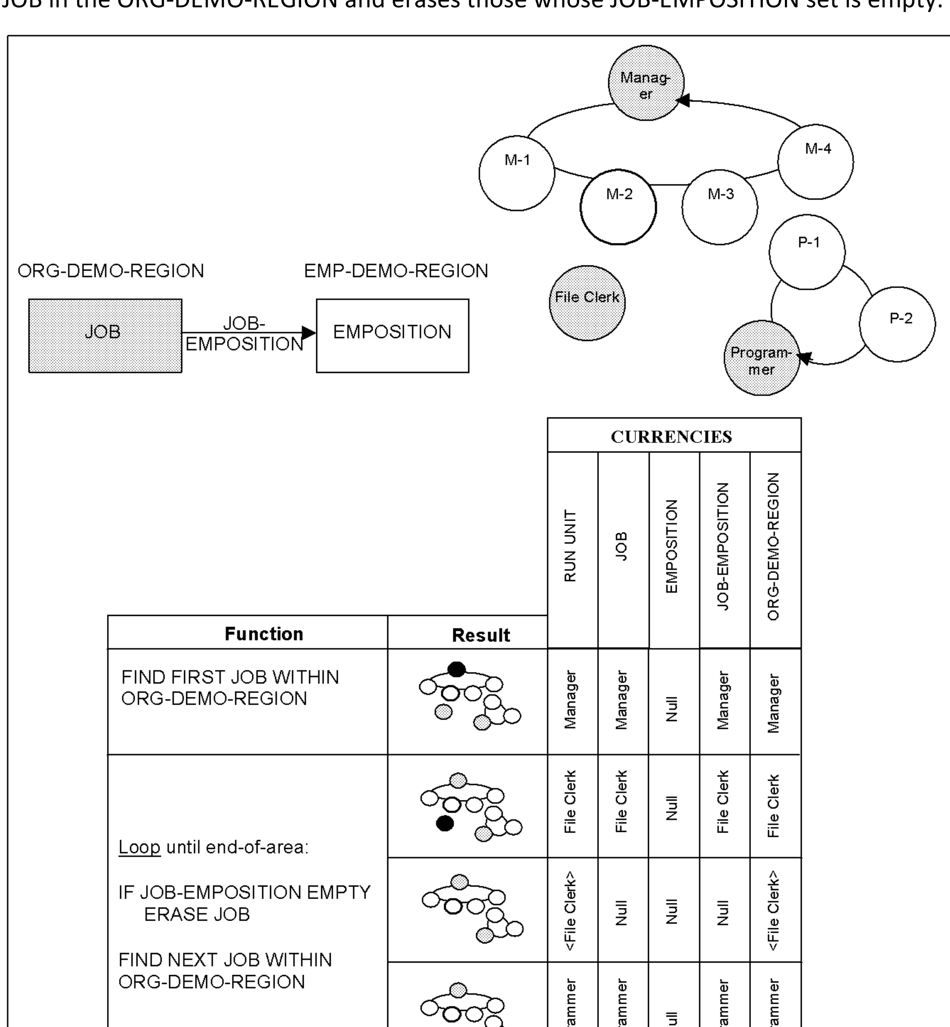
Using the IF function, users can test the following conditions associated with set membership:

- Whether a set occurrence is empty, that is whether there are member records in the set occurrence.
- Whether a record occurrence is a member of an occurrence of the set.

The IF function allows specification of the action to be taken when the result of the test is true. The following options of the IF function determine the condition to be tested:

- **EMPTY**- The test is true if the set occurrence identified by the current of set is empty. This test is typically made when positioned on an occurrence of the set's owner record type.
- **NOT EMPTY**- The test is true if the set occurrence identified by the current of set is not empty.
- **MEMBER**- The test is true if the current of run unit is a member in the specified set. This test is made when positioned on an occurrence of the set's member record type.
- **NOT MEMBER**- The test is true if the current of run unit is not a member in the specified set.

The following figure illustrates tests for an empty set condition. The routine locates each JOB in the ORG-DEMO-REGION and erases those whose JOB-EMPOSITION set is empty.



Chapter 5: Writing a Navigational DML Program

This chapter provides an overview of how to write a program to access a CA IDMS database using navigational DML. It first describes what is meant by DML and how it is used to request the execution of CA IDMS functions. It then discusses common considerations such as how to identify the operating mode in which the program will execute and the subschema it will use, how to copy data descriptions from the IDD, and how to check for and handle error conditions. The chapter then describes the DML statements used to initiate and terminate a run unit and perform other housekeeping functions. It also outlines considerations in the use of subschemas and in copying information from the dictionary into your program.

All examples in this chapter are based on the COBOL language. For full descriptions of the DML statements and for other languages, see the language-specific *CA IDMS DML Reference* Guide.

This section contains the following topics:

Navigational Data Manipulation Language (DML) (see page 93)
Common Considerations (see page 94)
Housekeeping Statements (see page 99)
Subschema Considerations (see page 102)
Copying Record Definitions and Their Synonyms (see page 105)

Navigational Data Manipulation Language (DML)

Application programs initiate CA IDMS functions by transferring control to the Database Management System (DBMS) to execute the function. The transfer of control is accomplished using the CALL mechanism available in most high-level languages. CA IDMS provides a **data manipulation language (DML)** statement for each CA IDMS function so that users do not have to code detailed host language calling sequences.

DML allows users to code statements patterned after the COBOL, PL/I, or Assembler programming language of the program in which the DML is embedded. Prior to compilation, the program is preprocessed through a language-specific precompiler to convert the DML statements into appropriate calling sequences. DML is also supported in the Advantage CA-ADS application development environment for which no preprocessing is necessary.

For example, to store a SCHEDULE record the COBOL programmer codes the following statement:

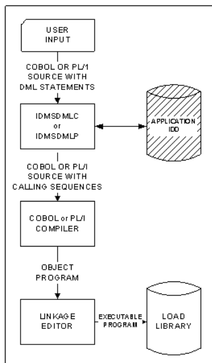
```
STORE SCHEDULE
```

The COBOL precompiler, IDMSDMLC, converts this statement into the following calling sequence:

```
CALL 'IDMS' USING SUBSCHEMA-CTRL
          IDBMSCOM (42)
          SR1007
```

In addition to converting DML statements into calling sequences, the DML precompiler also checks the syntax and logic of the statements, issues diagnostics, and copies source code and data descriptions from the CA IDMS Integrated Data Dictionary (IDD).

The following figure illustrates the compilation of an application using a precompiler to convert DML statements into calling sequences. The IDD supplies descriptions of the records and sets being accessed by the program and common routines associated with CA IDMS processing. The precompiler can also copy file definitions, map descriptors for interfacing with 3270 screens, non-IDMS record definitions, and source routines that have been stored in the IDD.



Common Considerations

Every program that uses navigational DML to access a CA IDMS database must do the following:

1. Identify the program's operating mode, the environment in which it will execute. Common operating modes are BATCH, BATCH-AUTOSTATUS, and IDMS-DC although many other environments are supported.
2. Identify the subschema, the program's view of the database to be accessed. The subschema must include all record types, set types, and areas required by the program.

3. Include descriptions of the IDMS communications block and the database records to be accessed. These descriptions are copied from the IDD into the program by the precompiler.
4. Detect and handle error conditions encountered during the execution of a DML function.

Identifying the Operating Mode

The program identifies the operating mode in which it will execute through a DML statement called a precompiler-directive that varies based on the language in which the program is written. COBOL programs identify the operating mode in a special section of the ENVIRONMENT DIVISION called the IDMS-CONTROL SECTION, as illustrated in the following example that identifies this as a batch application:

```
IDMS-CONTROL SECTION.

PROTOCOL MODE IS BATCH.
```

The operating mode affects the form and content of the calling sequences produced by the DML precompiler.

DEBUG can be specified as an option of the operating mode. If specified, each DML statement in the program is identified by a sequence number that can be used to determine the last DML statement that was executed when an error occurs. This can be very useful in debugging navigational applications.

Identifying the Subschema

A subschema is a program's view of the database and typically includes a subset of the records, record elements, sets, and areas defined in the schema describing the database to be accessed. A subschema can also limit the types of DML functions that can be issued by programs that use it.

The subschema to be used by the program is identified through a precompiler-directive statement that varies based on the language in which the program is written. COBOL programs identify the subschema in a special section of the DATA DIVISION called the SCHEMA SECTION, as illustrated below:

```
SCHEMA SECTION.

DB EMPSS01 WITHIN EMPSCHM VERSION 100.
```

In this example, EMPSS01 is the name of the subschema. Version 100 of EMPSCHM is the schema under which it is defined.

Including Data Descriptions

Descriptions of the following structures must be included in the program in order to reserve space for them in variable storage.

- **IDMS Communications Block**- CA IDMS uses the IDMS communications block to post status information back to the application program concerning requested database services. The description of the IDMS communications block is generated as a record named SUBSCHEMA-CTRL (in COBOL) and is often referred to as the subschema control block.
- **Subschema Names (COBOL only)**- COBOL programs require the generation of a name literal for the subschema, and each record, set, and area included in the subschema.
- **Subschema Records**- The description of each database record to be accessed by the program must be included. Only records contained in the subschema may be accessed and the generated description reflects the subschema's view of the record rather than that of the schema.

The programmer controls how and where the data descriptions are generated using precompiler-directive statements. For example, the COBOL programmer can automatically generate all required descriptors at the end of the WORKING-STORAGE section by extending the PROTOCOL statement as follows:

```
IDMS-CONTROL SECTION.
PROTOCOL MODE IS BATCH DEBUG
    IDMS-RECORDS WITHIN WORKING-STORAGE.
```

Alternatively, the COBOL programmer can generate the descriptions at the end of the LINKAGE SECTION by specifying:

```
IDMS-RECORDS WITHIN LINKAGE.
```

To have complete control over where structure descriptions are generated, the COBOL programmer can specify:

```
IDMS-RECORDS MANUAL.
```

The programmer then inserts a COPY statement, as shown below, at the point where the descriptions are to be generated:

```
COPY IDMS SUBSCHEMA-DESCRIPTION.
```

Additional forms of the COPY IDMS statement permit COBOL programmers to generate descriptions for the IDMS communications block, subschema names and subschema records separately, and control the level numbers that are generated.

See Copying Record Definitions and Their Synonyms for more information on copying information from IDD.

IDMS Communications Block

The IDMS communications block is the main interface block between your program and the DBMS. Whenever your program issues a call to the DBMS for a database operation, the DBMS returns information about the outcome of the requested service into the IDMS communications block. In particular, it contains the following fields (described using COBOL field names):

Field	Description
PROGRAM-NAME	Name of the program; supplied by the program
ERROR-STATUS	4-digit code indicating the outcome of the last database service; all zeros indicates successful completion
DBKEY	Database key of current of run unit
RECORD-NAME	Last record type successfully accessed
AREA-NAME	Area name of last record type successfully accessed
ERROR-SET	Name of set last involved in an error condition
ERROR-RECORD	Last record type involved in an error condition
ERROR-AREA	Name of area last involved in an error condition
PAGE-INFO	Page information for current of run unit
DIRECT-DBKEY	Suggested database key for storing a record direct
DML-SEQUENCE	Number of the DML statement last executed if DEBUG was specified in the operating mode

Error Handling

CA IDMS reports the completion status of a requested DML function by placing a 4-digit alphanumeric code in the ERROR-STATUS field of the IDMS communications block. The ERROR-STATUS field should be examined following every executable DML command to determine whether the request was successful or not.

The following table identifies some of the most common error status values that can be returned and the corresponding condition names that are generated as part of the IDMS communications block record description (SUBSCHEMA-CTRL) for COBOL. For PL/I, these names may optionally be generated as named constants.

Code	Explanation	Level-88 Name
0000	Request completed successfully	DB-STATUS-OK
0326	Record not found on a FIND or OBTAIN using a CALC key, db-key, index, or sort key	DB-REC-NOT-FOUND
0307	End of set or end of area was encountered on a FIND or OBTAIN NEXT or PRIOR in a set or area.	DB-END-OF-SET

For COBOL and PL/I programs, CA IDMS provides an error checking routine called IDMS-STATUS that can be copied into the program. IDMS-STATUS checks the completion status of the latest DML request that was issued. If the function did not complete successfully, IDMS-STATUS performs a user-supplied routine called IDMS-ABORT (COBOL only), displays status information, and aborts the program. The status information, retrieved from the IDMS communications block, includes PROGRAM-NAME, ERROR-STATUS, ERROR-RECORD, ERROR-SET, ERROR-AREA, RECORD-NAME, AREA-NAME, DBKEY, page group, dbkey format and DML-SEQUENCE.

CA IDMS delivers IDMS-STATUS in 2 flavors. The new flavor includes DBKEY, page group and dbkey format and is defined in the dictionaries as versions 11 and higher. The old flavor does not expand any additional call to the internal formatting routine and thus does not display DBKEY, page group or dbkey format. The old flavor is defined as versions 1 to 5. The higher versions always take precedence. If you prefer to use an older version, you can drop the new versions 11 and higher from the dictionary which causes the old versions to be copied into the program instead. Another option is to select the requested version in the program using the VERSION clause of the COPY IDMS statement (INCLUDE IDMS statement in PL/I).

The COBOL example below shows how the IDMS-STATUS routine is copied into the program using the COPY IDMS statement. The example also shows the user-supplied IDMS-ABORT routine that must be coded by the COBOL programmer.

```
COPY IDMS IDMS-STATUS.
IDMS-ABORT SECTION.
IDMS-ABORT-EXIT. EXIT.
```

After every executable DML function, the program should do the following:

- Check the ERROR-STATUS field for any expected error codes. COBOL condition names (level 88s) have been provided with the standard SUBSCHEMA-CTRL definition for the most common ERROR-STATUS values.
- PERFORM IDMS-STATUS to check for any unexpected error codes.

For example, after issuing an OBTAIN CALC request, the program should check for an ERROR-STATUS value of '0326' (DB-REC-NOT-FOUND) and otherwise perform the IDMS-STATUS routine as follows:

```
OBTAIN CALC STUDENT.
IF DB-REC-NOT-FOUND
   ...
   ...
ELSE PERFORM IDMS-STATUS.
```

CA IDMS provides a set of protocols (operating modes) for COBOL programs that automatically perform the IDMS-STATUS routine as part of expanding the DML statement. These modes are referred to as "autostatus" protocols and have names such as BATCH-AUTOSTATUS. When using an autostatus protocol, expected ERROR-STATUS values can be specified as ON parameters in the DML statement. The following example illustrates the use of the ON clause to check for an '0326' ERROR-STATUS value when issuing an OBTAIN CALC request.

```
OBTAIN CALC STUDENT
ON DB-REC-NOT-FOUND
 ...
 ...
```

Housekeeping Statements

Housekeeping chores consist of initiating a run unit, binding records to storage areas, readying areas, and terminating the run unit. These functions were introduced in "Housekeeping Functions" in chapter 3. This section describes how to code DML statements that request the execution of these functions. For a full description of these statements and the possible error status values that may result, see the appropriate language-specific *CA IDMS DML Reference Guide*.

BIND RUN-UNIT Statement

The first database function executed within your program must be a BIND RUN-UNIT in order to establish a session with the DBMS. You can code this in one of two ways:

- By coding a BIND RUN-UNIT statement.
- By coding a COPY IDMS SUBSCHEMA-BINDS statement.

The COPY IDMS SUBSCHEMA-BINDS causes the precompiler to generate the BIND RUN-UNIT statement followed by BIND RECORD statements for every subschema record whose description is included in the program.

In deciding which statement to use, consider the following:

- If coding the BIND RUN-UNIT statement explicitly, you should first move the name of the program to the PROGRAM-NAME field within the IDMS communications block. The precompiler does this for you if you are using COPY IDMS SUBSCHEMA-BINDS.
- COPY IDMS SUBSCHEMA-BINDS should only be used if an autostatus protocol is in effect since otherwise error checking is not performed after each of the generated DML statements.
- Optional clauses on the BIND RUN-UNIT statement allow the specification of the target database to access, the subschema to use, and the dictionary from which to load the subschema. These options are not available when using COPY IDMS SUBSCHEMA-BINDS.

The following COBOL example binds a run unit to the EMPDEMO database:

```
MOVE 'MYPROG' TO PROGRAM-NAME.
BIND RUN-UNIT DBNAME 'EMPDEMO'.
PERFORM IDMS-STATUS.
```

If your program serially initiates and terminates multiple run units, you should re-initialize the ERROR-STATUS field to '1400' before starting each subsequent run unit.

BIND RECORD Statement

Before a record can be accessed, you must bind it to a specific location in variable storage by issuing a BIND RECORD function. To do this, you either:

- Code a BIND RECORD statement for each record to be accessed.
- Code a COPY IDMS SUBSCHEMA-BINDS statement. See the preceding section for more information on this statement.

The following example binds the DEPARTMENT and EMPLOYEE records to their respective locations in variable storage.

```
BIND DEPARTMENT.
PERFORM IDMS-STATUS.
BIND EMPLOYEE.
PERFORM IDMS-STATUS.
```

READY Statement

Each database area containing records to be accessed must be readied. The following example readies both the EMP-DEMO-REGION and the ORG-DEMO-REGION areas:

```
READY EMP-DEMO-REGION
   USAGE-MODE IS PROTECTED UPDATE.
PERFORM IDMS-STATUS.
READY ORG-DEMO-REGION
   USAGE-MODE IS RETRIEVAL.
PERFORM IDMS-STATUS.
```

The specified usage mode allows records within the EMP-DEMO-REGION to be updated (stored, modified, and erased) but prevents them from being updated by other programs running concurrently; records within the ORG-DEMO-REGION can only be retrieved (no updating will be allowed).

You can code a single ready statement if all areas within the subschema are to be readied with the same mode. The following statement readies all areas in a shared update mode:

```
READY USAGE-MODE IS UPDATE.
PERFORM IDMS-STATUS.
```

It is possible to define default usage modes for areas within a subschema. A program using such a subschema need not code a READY statement. If the program does code a READY statement, it must ready every area that it will access unless the FORCE option was specified for the default usage mode. Areas using the default usage mode combined with the FORCE option are automatically readied even if the run-unit already issued READY for other areas.

Note: Your program should ready all areas that it intends to access before issuing any other DML request (other than BIND RUN-UNIT and BIND RECORD). This avoids deadlocks between programs that ready areas in conflicting ways such as shared update and protected update.

Termination Statements

After a program has completed its database access activities successfully, it must issue the FINISH function to "close" the database and commit any changes that it has made. The following example shows how to code the FINISH DML statement in COBOL:

```
FINISH.
PERFORM IDMS-STATUS.
```

It is also possible to commit changes without terminating the run unit. This is often done at the completion of a logical unit of work to free up records that have been updated so that other programs can access them, while keeping the run unit open so that more work can be done. To do this, a COMMIT DML statement is used instead of a FINISH, as shown in the following example that commits changes but maintains the run unit's currencies:

```
COMMIT.
PERFORM IDMS-STATUS.
```

An option on the COMMIT statement nullifies currencies in addition to committing database changes.

If an error is encountered during program execution that prevents successful completion of the logical unit of work, the program should issue a ROLLBACK DML statement instead of a FINISH as shown in the following example in which database changes are rolled back and the run unit terminated:

```
ROLLBACK.
PERFORM IDMS-STATUS.
```

An option on the ROLLBACK statement permits the run unit to continue after the database changes have been rolled back.

Note: Database changes are only rolled back automatically when the run unit is executing under central version. In local mode, the database must be manually recovered.

For more information on the use of these statements and their impact on throughput and database integrity, see Run Units, Locks, and Database Transactions.

Subschema Considerations

A **subschema** is a program's view of the database; it typically defines a subset of the records and record elements contained in the schema. The following rules apply to subschema usage:

- Any number of subschemas can be associated with a single schema.
- Any number of programs can share a subschema.
- A program can have only one subschema.

Comparing Subschema and Schema

The table below compares the features and characteristics of subschemas and schemas.

Subschema	Schema
One or more per database	One per database
A program's view of the database (subset of records and record elements)	Complete database description (all records and record elements)
Source description resides in the DDLDML area of the dictionary	Source description resides in the DDLDML area of the dictionary
Source description used at DML program compile time	Source description not used at DML program compile time
A load module resides in the DDLDCLOD area of the dictionary or in a load (core-image) library	No load module
Load module used at run time	Not used at run time

Subschema Access Restrictions

The subschema may place restrictions on the DML statements that can be used to access database records. For example, you may be able to retrieve a record but not modify or erase it.

DBA-designated access restrictions, defined in the subschema, control program access to the database. Restrictions can be placed on:

- **Areas**-Access restrictions placed on areas prevent programs from readying them in specified usage modes (see Area Usage Modes).

  For example, a subschema with an update access restriction on the ORG-DEMO-REGION area **can prevent** programs from readying that area in any update mode.

- **Records**-Access restrictions placed on records prohibit programs from performing one or more of the following DML functions against the specified record types:
  - STORE .li ERASE
  - CONNECT .li FIND
  - MODIFY .li GET
  - DISCONNECT .li KEEP

For example, ERASE IS NOT ALLOWED for the OFFICE record type prohibits a program using the subschema from erasing OFFICE record occurrences.

Note: The DML OBTAIN statement is a combination of FIND and GET; access restrictions on either FIND or GET will affect the use of OBTAIN.

- **Sets**-Access restrictions placed on sets prohibit programs from performing one or more of the following DML functions against record occurrences in the specified set:

  - CONNECT .li FIND
  - DISCONNECT .li KEEP

  For example, DISCONNECT IS NOT ALLOWED for the JOB-EMPOSITION set prohibits a program from disconnecting EMPOSITION occurrences from the JOB-EMPOSITION set.

If your program issues a DML statement that is prohibited in the subschema, the DBMS returns a status of *nn*10 in the ERROR-STATUS field in the IDMS communications block. The IDMSRPTS utility (discussed below) produces listings of any access restrictions that apply to a given subschema.

Program Registration

The DBA can specify in the subschema that each program that is to use the subschema must be defined in the dictionary before compilation under one of the precompilers. If program registration is in effect, you should ensure that the name listed in the PROGRAM-ID statement (for COBOL) matches the program name registered with IDD.

IDMSRPTS Utility

The IDMSRPTS utility produces listings that describe the database definition (that is, the schema and all associated subschemas). These reports are useful in all phases of program development; they provide the following information:

- Names of all records, sets, and areas included in the subschema
- Names, attributes, and positions of all elements included in each subschema record
- Storage mode of each record
- Access restrictions
- Set characteristics

IDMSRPTS Parameters

The table below lists the parameters of the IDMSRPTS utility that are most useful to applications programmers.

Parameter	Requested information
RECDES	All records and record elements defined in the schema
SETDES	Set name, owner, membership options, and linkage options for all sets defined in the schema
SUBREC	All records and record elements defined in the subschema; access restrictions placed on records
SUBSET	Set name, owner, membership options, and linkage options for all sets defined in the subschema; access restrictions placed on sets
SUBAREA	Usage modes applicable to subschema areas, default usage modes; access restrictions placed on areas

For more information on the IDMSRPTS utility, see *CA IDMS Utilities Guide*.

Copying Record Definitions and Their Synonyms

Typically, you copy subschema records into variable storage using the primary name that the record is known by in the schema. Synonyms are alternative names for existing dictionary entities. A given file, record, or element can have multiple names through the use of IDD synonyms. This allows all programs that use an entity such as a record to access the same entity definition but refer to it using a name that complies to the language in which the program is written.

Uses of Synonyms

Synonyms are typically used for the following reasons:

- **To allow you to copy schema-owned records into a program whose subschema is not associated with that schema**. If a record has been copied into a schema, it can be copied only into a program that uses a subschema associated with that schema.

  However, if your program uses a subschema that is not associated with that schema, you cannot copy the record definition into the program unless you do so through one of its synonyms. One way to do that is to specify a version number on the COPY IDMS statement:

  ```
  COPY IDMS RECORD EMPLOYEE VERSION 100.
  ```

- **To allow different programming languages to access the same record definition**. For example, in Assembler the EMPLOYEE record can be defined as EMPLOYE.

  Note: The precompiler for PL/I automatically automatically converts hyphens to underscores. For example, if you define a record called NON-HOSP-CLAIM and copy its description into a PL/I program using the following statement:

  ```
  INCLUDE IDMS (NON-HOSP-CLAIM);
  ```

  The precompiler converts the hyphens to underscores:

  ```
  DMLP      INCLUDE IDMS (NON-HOSP-CLAIM);
          DECLARE 1  NON_HOSP_CLAIM,
               2  CLAIM_TYPECHARACTER (2);
  ```

Terminology for Using Synonyms

You should be familiar with the following terms:

- **Schema-owned** refers to any record that is defined in a schema.
- **IDD-defined** refers to any record defined using the DDDL compiler that has not been included in a schema.
- **Mode** refers to the operating mode of your program (that is, BATCH, IDMS-DC, DC-BATCH, CICS, and so on). IDD-defined records can be assigned one of these modes, a mode of NON-MODESPECIFIC, or no mode attribute at all.
- **Language attribute** refers to the optional attribute that can be included in IDD-defined records and synonyms. For example, LANGUAGE IS COBOL, LANGUAGE IS PL/I, or LANGUAGE IS DC.

How the Precompiler Performs COPY IDMS

When the precompiler selects which record or synonym to copy into your program, it first checks to see if you have specified a VERSION clause in the COPY command. If no VERSION clause is given, a two-fold search is undertaken, first for a record associated with the subschema and then, if the first test fails, for an IDD-defined record.

To determine if the specified record is associated with the subschema, the precompiler performs the following steps:

1. **Forms a table of records defined in the subschema and their synonyms**. This table contains all records copied into the subschema and, for every record copied, the names of its synonyms *not* copied into another subschema.
2. **Searches this table to match the name of the record in the COPY statement**. If a match is found, that record is copied in; if no match is found, the search continues as described below.

If you specify a VERSION clause or if the test listed above fails, the precompiler assumes that the record is an IDD-defined record and performs the following steps:

1. **Identifies a candidate record.** A record is a candidate if it has a synonym whose name matches the name specified in the COPY IDMS statement.
2. **Checks the builder code.** The candidate record is tested for being either schema-owned (builder code of S) or a subschema view (builder code of V). If either is true, the record is rejected and another candidate is examined.
3. **Checks the VERSION clause.** If a VERSION clause is specified on the COPY IDMS statement, the candidate record's version is compared to that specified. If they are not equal, the record is rejected and another candidate is examined.
4. **Checks the language attribute.** If the candidate record has a language attribute and it is different than that of the compiler being used (for example, PL/I), the record is rejected and another candidate is examined.
5. **Checks the mode.** The mode associated with the candidate record is compared with the operating mode specified in the program. If they match, the record remains a candidate. If there is no record with a match on mode and the candidate has a mode of NON-MODESPECIFIC, it remains a candidate. If there is no record with a matching mode or a mode of NON-MODESPECIFIC and the candidate has no mode associated with it, it remains a candidate. Otherwise, the record is rejected and another candidate is examined.
6. **Rechecks the VERSION clause.** If a VERSION clause is specified on the COPY IDMS statement, the candidate record is chosen as the one to be copied into the program. If no VERSION clause is specified, the candidate with the highest version meeting all the above criteria is chosen.

Chapter 6: Navigational DML Programming Techniques

This chapter discusses programming techniques used to access the database in navigational DML programs. Functionally similar DML statements are presented together; sample code that demonstrates typical usage of each statement is included. The navigational DML functions are divided into these categories:

- **Retrieving Records**-Retrieving information from the database by using navigational DML statements
- **Saving DB-Key and Address Information**-Saving db-keys and bind addresses
- **Checking For Set Membership**-The two forms of the DML IF statement, used to obtain set membership information without performing any I/O
- **Updating the Database**-Modifying, storing, erasing, connecting, and disconnecting database records
- **Accessing Bill-of-Materials Structures**-Storing and retrieving records related as in a bill-of-materials structure
- **Locking Records**-Restricting access to database records

This chapter also contains information on the following topics of relevance to the navigational DML programmer:

- How page information can be used to make db-keys unique within the scope of a run unit
- Currency and how it is used and updated by various DML statements
- How to collect run-time statistics

This section contains the following topics:

DB-Keys and Page Information (see page 110)
Run Unit Currencies (see page 112)
Retrieving Records (see page 115)
Saving db-key, Page Information, and Bind Addresses (see page 134)
Checking for Set Membership (see page 140)
Updating the Database (see page 145)
Accessing Bill-of-Materials Structures (see page 159)
Locking Records (see page 166)
Implicit Record Locks (see page 168)
Collecting Database Statistics (see page 171)

DB-Keys and Page Information

Database Keys

Each database record occurrence is identified by a database key (db-key). The db-key is a 4-byte identifier that consists of:

- A page number, which identifies the page on which the record occurrence is stored
- A line number, which identifies the record's location on the page

The DBMS assigns a db-key to a record occurrence when the occurrence is stored in the database; that db-key remains unchanged until the occurrence is erased or the database is unloaded and subsequently reloaded.

Page Information

Page numbers are used to identify pages within a database; however, a page number is not necessarily unique across all areas accessible to a CA IDMS run-time system or even across all areas accessible to your run unit. If a page number is not unique, you can qualify it with additional information so that it uniquely identifies a page. The additional qualifying information is a 4-byte identifier that consists of:

- A 2-byte page group
- A 2-byte db-key radix

A page group is a number assigned to an area by the DBA for the purpose of making the area's page range unique to the CA IDMS run-time system. The db-key radix indicates the number of bits within the 4-byte db-key that contain a record's line number. The db-key radix is calculated by CA IDMS based on the maximum number of record occurrences that can be stored on a page of the area.

For more information on page groups and maximum records per page, see the *CA IDMS* Database *Administration Guide*.

Qualifying Db-keys

Normally all areas accessed by a run unit have the same page information and so the db-key of a record occurrence uniquely identifies it from all other record occurrences accessible to the run unit. A run unit, however, can access areas with different page groups or db-key radixes if it accesses a database defined to allow mixed page group binds. When this happens, a db-key must be qualified either by record type or page information so that it uniquely identifies a record occurrence.

In order to permit qualification, either record type or page information can be specified when retrieving a record occurrence through its db-key. Whenever a record occurrence is retrieved, its record type, db-key and associated page information are returned to the application program in the IDMS communications block. You can save these for later use in retrieval commands. It is also possible to determine the page information associated with a specific record type by issuing an ACCEPT Page-Info command.

For more information on retrieving a record by db-key, see Accessing a Record by Its db-key.

For more information on saving qualifying information, see Saving Page Information.

For more information on mixed page group binds and accessing areas with different page groups or maximum records per page, see the *CA IDMS Database Administration Guide*.

Page Information and Record Types

For the duration of a run unit, the page information for all occurrences of a given record type is the same. Similarly, the page information for all record types within an area or all record types associated with a set is the same.

Using Page Information to Interpret Db-keys

The format of a db-key value depends on its db-key radix. The db-key radix specifies the number of bits within a db-key that are reserved for a record occurrence's line number. Since the db-key radix is part of the page information associated with a db-key, you can use page information to interpret a 4-byte db-key value. You can use this when displaying db-keys for error reporting purposes or when establishing a target page for storing records whose location mode is direct.

Given a db-key, you can separate its associated page number by dividing the db-key by 2 raised to the power of the db-key radix. For example, if the db-key is 4, you divide the db-key value by 2**4. The resulting value is the page number of the db-key. To separate the line number, you multiply the page number by 2 raised to the power of the db-key radix and subtract this value from the db-key value. The result is the line number of the db-key. You can use the following two formulas to calculate the page and line numbers from a db-key value:

- Page-number = db-key value / (2**db-key radix)
- Line-number = db-key value - (page number*(2**db-key radix))

Run Unit Currencies

During the execution of your application program, the DBMS uses **currency** to keep track of the database location (db-key) of the most recently accessed record occurrences for the run unit, record type, set, and area. By keeping track of the most recently accessed records, currency enables you to navigate the database with a minimum of effort. Currency values determine which record occurrences are affected by DML functions requested by an application program. Upon successful execution of a DML statement, the DBMS automatically updates currency values, as appropriate.

A record occurrence can be:

- Current of run unit
- Current of record type
- Current of set
- Current of area

Current of Run Unit

The record occurrence that was the object of the most recent successful FIND, OBTAIN, CONNECT, STORE, MODIFY, DISCONNECT, or ERASE function is current of run unit. Only one current record of run unit exists at any given time during program execution. That record's db-key, record type and qualifying page information are placed in the DBKEY, RECORD-NAME and PAGE-INFO fields of the IDMS communications block.

For more information on the IDMS communications block, see IDMS Communications Block.

Current of Record Type

The most recently accessed occurrence of each record type is current of that record type. At any given time during program execution, one current record can exist for each record type defined in the program's subschema. For example, when your program successfully retrieves JOB 2215, that record becomes current of the JOB record type. If you then successfully obtain EMPLOYEE 466, that record becomes current of the EMPLOYEE record type; currency for the JOB record type remains unchanged.

Current of Set

The most recently accessed record occurrence in each set is current of set for that set. At any given time during program execution, one current record can exist for each set defined in the program's subschema.

Because a successfully accessed record becomes the current record of all sets in which it participates as either owner or member, a given record occurrence can be the current record of any number of sets.

Current of Area

The most recently accessed record occurrence in each area is current of area for that area. At any given time during program execution, one current record can exist for each area defined to the program's subschema.

When Currency is Established

At the beginning of a program, all currencies are null. Currency is established by the DML FIND, OBTAIN, RETURN, or STORE function. Currency is updated following each successful execution of a FIND, OBTAIN, CONNECT, DISCONNECT, ERASE, RETURN, MODIFY, or STORE statement.

How the DBMS Uses Currency

The DBMS uses currency to:

- Establish a starting point for the execution of a DML retrieval statement by using the current position in the database with respect to run unit, record, set, or area
- Establish proper set occurrences for STORE, CONNECT, and DISCONNECT functions
- Determine the target record for a MODIFY or ERASE statement
- Determine the physical placement in the database of records stored with a location mode of VIA
- Provide the basis for saving the db-keys of related records for subsequent use by the program
- Prevent a record that is current of record, set, or area from being updated by another application

Use and Updating of Currency by DML Verbs

The table below outlines the currency required to execute each DML verb and the changes to currency following the successful execution of that verb. The bullet symbol (&bul.) indicates currency used in command execution.

Note: BIND and READY do not use or update currency, but both verbs must be issued before any database access is attempted.

DML verb	Run unit	Record	Set	Area	Currency updated by successful execution
ACCEPT*	&bul.	&bul.	&bul.	&bul.	None
IF*	&bul.		&bul.		None

DML verb	Run unit	Record	Set	Area	Currency updated by successful execution
FIND/OBTAIN DB-KEY					All
FIND/OBTAIN CURRENT*	&bul.	&bul.	&bul.	&bul.	All
FIND/OBTAIN WITHIN SET1			&bul.		All
FIND/OBTAIN WITHIN AREA				**	All
FIND/OBTAIN OWNER			&bul.		All
FIND/OBTAIN CALC					All
FIND/OBTAIN DUPLICATE		&bul.			All
FIND/OBTAIN USING SORT KEY2			&bul.		All
GET	&bul.				None
RETURN3			&bul.		Set
STORE			***		All
MODIFY	&bul.				None4
ERASE	&bul.				Nullifies currencies of all record types and sets involved
CONNECT		&bul.	&bul.		Run unit, set
DISCONNECT		&bul.			Nullifies currency of object set; updates current of run unit and area
KEEP*	&bul.	&bul.	&bul.	&bul.	None
COMMIT					None
COMMIT ALL					Nullifies all currencies
ROLLBACK					Nullifies all currencies
ROLLBACK CONTINUE					Nullifies all currencies
FINISH					Nullifies all currencies

* Uses only one currency as determined by command format.
** Required for NEXT and PRIOR formats only.
*** All in which record type participates as an automatic member.
1 Currency is not required if the statement specifies FIRST, LAST or *sequence-number* for a system-owned indexed set.
2 Currency is not required for a system-owned indexed set.
3 Currency is not required if the statement specifies FIRST, LAST, or USING *index-key*.
4 Except in the case of a sorted set.

Retrieving Records

In the navigational environment, you can use the following navigational DML statements to retrieve database records:

- **FIND** locates a record occurrence in the database.
- **GET** moves the data associated with a record occurrence from the page buffers to program variable storage.
- **OBTAIN** locates a record occurrence in the database and moves the data associated with that occurrence to program variable storage (the equivalent of a FIND followed by a GET).
- **RETURN** retrieves the db-key and the symbolic key for an indexed record without retrieving the record itself.

The DML retrieval functions listed below are discussed on the following pages:

- Accessing CALC records
- Walking a set
- Accessing a sorted set
- Performing an area sweep
- Accessing owner records
- Reestablishing run-unit currency
- Accessing a record by its db-key
- Accessing indexed records
- Moving contents of a record occurrence

Accessing CALC Records

To access a record occurrence based on its CALC-key value, perform the following steps:

1. Move the CALC-key value(s) to the CALC-key field(s) in the database record in variable storage.
2. Issue the FIND/OBTAIN CALC command.
3. Check the ERROR-STATUS field for the value 0326 (DB-REC-NOT-FOUND).
4. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value other than 0326.

Example of Retrieving CALC Records

The program excerpt below shows retrieval of CALC records.

The MOVE statement initializes the CALC-key field before the database access is performed. If the DBMS returns an ERROR-STATUS of 0326 (condition DB-REC-NOT-FOUND), the program prints a message and goes on to the next input record.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SWITCHES.

  05 EOF-SW              PIC X    VALUE 'N'.
    88 END-OF-FILE                VALUE 'Y'.
PROCEDURE DIVISION.
       .
       .
   READ EMP-FILE-IN
      AT END MOVE 'Y' to EOF-SW.
   IF NOT END-OF-FILE
      PERFORM A400-GET-EMP-REC THRU A400-EXIT
        UNTIL END-OF-FILE.
   FINISH.
   GOBACK.
A400-GET-EMP-REC.
  *** INITIALIZE CALC KEY ***
   MOVE EMP-ID-IN TO EMP-ID-0415.
  *** RETRIEVE RECORD ***
   OBTAIN CALC EMPLOYEE.
    *** CHECK FOR ERROR-STATUS = 0326 ***
   IF DB-REC-NOT-FOUND THEN
     DISPLAY 'EMPLOYEE ID: ' EMP-ID-IN ' NOT FOUND'
    *** CHECK FOR ERROR-STATUS = 0000 ***
```

```
  ELSE IF DB-STATUS-OK
    PERFORM B100-WRITE-EMP-REPORT
  ELSE
    PERFORM IDMS-STATUS.
   READ EMP-FILE-IN
     AT END MOVE 'Y' to EOF-SW.
A400-EXIT.
  EXIT.
```

Retrieving Records with Duplicate CALC-keys

Record types may be defined to allow duplicate CALC-keys. To retrieve all records that have the same CALC-key value, retrieve the first by using the FIND/OBTAIN CALC statement shown above and then use the DUPLICATE option of the FIND/OBTAIN CALC statement to retrieve the additional records. An '0326' error status signals when no more records with the same key exist.

Walking a Set

To access a record occurrence based on its logical position within a set, perform the following steps:

1. Establish the current of set for the specified set type (for example, by issuing an OBTAIN CALC).
2. Issue the FIND/OBTAIN WITHIN SET command.
3. Check the ERROR-STATUS field for the value 0307 (DB-END-OF-SET).
4. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value other than 0307.

Example of Walking a Set

The program excerpt below shows the procedure for retrieving all member records in a set.

The program enters the database on the CALC-key field DEPT-ID-0410 and establishes currency on the DEPARTMENT record. It then walks the DEPT-EMPLOYEE set until the DBMS returns an ERROR-STATUS of 0307 (DB-END-OF-SET).

```
WORKING-STORAGE SECTION.
01  SWITCHES.

   05 EOF-SW              PIC X    VALUE 'N'.
      88 END-OF-FILE              VALUE 'Y'.
PROCEDURE DIVISION.
        .
    READ DEPT-RECORD-IN
        AT END MOVE 'Y' TO EOF-SW.
    PERFORM A300-GET-DEPT-SET THRU A300-EXIT
                UNTIL EOF-SW = 'Y'.
    FINISH.
    GOBACK.
A300-GET-DEPT-SET.
    MOVE DEPT-ID-IN TO DEPT-ID-0410.
    OBTAIN CALC DEPARTMENT.
    IF DB-REC-NOT-FOUND
      DISPLAY 'DEPT: ' DEPT-ID-IN ' NOT FOUND'
      GO TO A300-GET-NEXT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
    MOVE DEPT-NAME-0410 TO DEPT-NAME-OUT.
    PERFORM U0900-WRITE-LINE.
A300-SET-WALK.
     *** RETRIEVE NEXT EMPLOYEE IN SET ***
    OBTAIN NEXT EMPLOYEE WITHIN DEPT-EMPLOYEE.
     *** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET
      GO TO A300-GET-NEXT
     *** CHECK FOR ERROR-STATUS = 0000 ***
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
    MOVE EMP-NAME-0415 TO EMP-NAME-OUT.
    MOVE EMP-ID-0415 TO EMP-ID-OUT.
    PERFORM U0900-WRITE-LINE.
    GO TO A300-SET-WALK.
A300-GET-NEXT.
    READ DEPT-RECORD-IN
        AT END MOVE 'Y' TO EOF-SW.
A300-EXIT.
    EXIT.
```

Accessing a Sorted Set

To access a record occurrence in a sorted set based on its sort key, use the FIND/OBTAIN WITHIN SET USING SORT KEY statement. The elements that make up a sort key need not be adjacent to one another (that is, they can be contiguous or noncontiguous).

To access a record that has either a **single** or a **contiguous** sort key, perform the following steps:

1. Establish the current of set for the specified set type.
2. Initialize the sort-key field of the database record in program variable storage with the sort-key value; for example:

   ```
   MOVE 77 TO SORT-KEY-1.
   ```

3. Issue the FIND/OBTAIN WITHIN SET USING SORT KEY statement; for example:

   ```
   OBTAIN RECORD-B WITHIN A-B USING SORT-KEY-1.
   ```

4. Check the ERROR-STATUS field for the value 0326 (DB-REC-NOT-FOUND).
5. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value other than 0326.

Sorted Set with a Noncontiguous Sort Key

To access a record that has a **noncontiguous** sort key, perform the following steps:

1. Establish a work field in program variable storage that consists of the record's multiple sort-key elements stored as contiguous data items:

   Subschema Record

   ```
   02  RECORD-B.
     05 SORT-KEY-1           PIC 9(2).
     05 NOT-A-KEY-1          PIC X(8).
     05 SORT-KEY-2           PIC 9(5).
     05 NOT-A-KEY-2          PIC XXX.
     05 SORT-KEY-3           PIC X(15).
     05 NOT-A-KEY-3          PIC 9(5)V99.
   ```

   Work Record

   ```
   02  SORT-RECORD-B
     05 S-KEY-1            PIC 9(2).
     05 S-KEY-2            PIC 9(5).
     05 S-KEY-3            PIC X(15).
   ```

2. Move the sort key values into the work record; for example:

```
MOVE 77         TO S-KEY-1.
MOVE 12345      TO S-KEY-2.
MOVE 'PROGRAMMER' TO S-KEY-3.
```

3. Establish the current of set for the specified set type.

4. Issue the FIND/OBTAIN statement, using the work record:

```
OBTAIN RECORD-B WITHIN A-B
         USING SORT-RECORD-B.
```

5. Check the ERROR-STATUS field for the value 0326 (DB-REC-NOT-FOUND).

6. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value other than 0326.

'Batch programmers'. Sorted sets can be processed more efficiently by sorting the input transactions in the same order as the set before program execution.

Example of Retrieval Using a Sort Key

The program excerpt below retrieves an EMPLOYEE record through its sort key.

This example retrieves insurance records for all specified employees. It enters the database through the EMP-NAME-NDX set using the sort key, which is composed of the employee's last name and first name. This example eliminates the need to initialize the sort key elements in the record by using the input file as the sort-control element.

```
DATA DIVISION.
FILE SECTION.

FD  SORTED-EMP-FILE-IN.
01  INS-INQ-EMP-REC-IN.
    02  EMP-SORT-NAME.
      04  LAST-IN             PIC X(15).
      04  FIRST-IN            PIC X(10).
WORKING-STORAGE SECTION.
01  SWITCHES.
   05 EOF-SW               PIC X    VALUE 'N'.
     88 END-OF-FILE                 VALUE 'Y'.
PROCEDURE DIVISION.
        .
    READ INS-INQ-EMP-REC-IN
        AT END MOVE 'Y' TO EOF-SW.
    PERFORM A300-GET-EMP-NDX THRU A300-EXIT
              UNTIL EOF-SW = 'Y'.
    FINISH.
    GOBACK.
A300-GET-EMP-NDX.
     *** RETRIEVE EMPLOYEE USING SORT KEY ***
```

```
 OBTAIN EMPLOYEE WITHIN EMP-NAME-NDX
      USING EMP-SORT-NAME.
  *** CHECK FOR ERROR-STATUS = 0326 ***
 IF DB-REC-NOT-FOUND
   THEN DISPLAY
   'EMPLOYEE ' INS-INQ-EMP-REC ' NOT FOUND'
   GO TO A300-GET-NEXT
  *** CHECK FOR ERROR-STATUS = 0000 ***
 ELSE IF DB-STATUS-OK
   NEXT SENTENCE
 ELSE
   PERFORM IDMS-STATUS.
 PERFORM A400-GET-INS-INFO.
A300-GET-NEXT.
 READ INS-INQ-EMP-REC-IN
     AT END MOVE 'Y' TO EOF-SW.
A300-EXIT.
 EXIT.
A400-GET-INS-INFO.
  *** RETRIEVE ALL INSURANCE CLAIM RECORDS THROUGH THE ***
  *** EMP-COVERAGE AND COVERAGE-CLAIMS SETS          ***
```

Sorted Set Considerations

You should be aware of the following considerations related to the processing of sorted sets:

- The selected record occurrence has a key value equal to the value of the sort-control element. If more than one occurrence contains a sort key equal to the key value in variable storage, the first such record is selected.
- The search for the specified record begins with the owner of the current of set unless the CURRENT option is specified. When CURRENT is specified, the search begins with the currencies already established for the specified set.

  Note: If duplicates are allowed, iterative use of CURRENT continually returns the same occurrence; in this case, use OBTAIN NEXT WITHIN SET.

- The search always proceeds in the next direction. The next of set is the member record with the next higher sort-key value (next lower for descending sets) than the requested value; the prior of set is the member record with the next lower value (higher for descending sets).

Generic Key Searches

If a member occurrence with the requested sort-key value is not found, the current of set is nullified but the next and prior of set are maintained.

You can use this feature to perform generic key searches. For example, to retrieve all employees whose last names start with the letter **N** or greater, you can establish the appropriate currency by issuing the following statements:

```
MOVE 'N          ' TO EMP-LAST-NAME-0415.
FIND EMPLOYEE WITHIN EMP-LNAME-NDX USING EMP-LAST-NAME-0415.
IF ERROR-STATUS = '0326'
  NEXT SENTENCE
ELSE
  PERFORM IDMS-STATUS.
```

To return the first record containing the partial key value followed by characters other than blanks, you issue this statement:

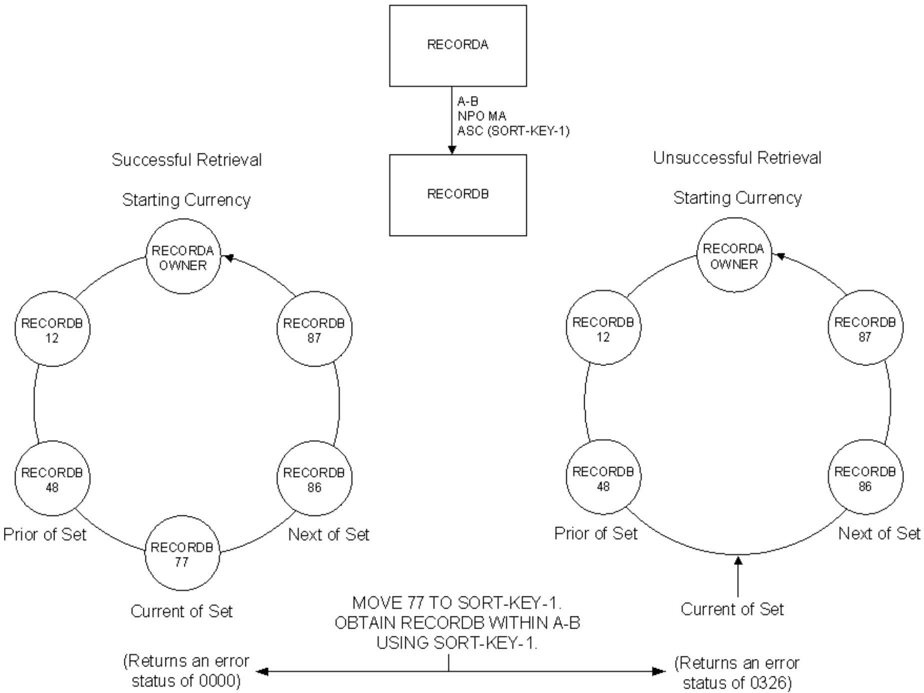
```
OBTAIN NEXT EMPLOYEE WITHIN EMP-LNAME-NDX.
```

Continue to issue this OBTAIN until all records within the range you want have been returned.

Example of Retrieving Occurrences of Sorted Sets

The figure below shows the currencies maintained by successful and unsuccessful retrieval within a sorted set.

Following successful retrieval within the A-B set, member occurrence 77 is established as current. Following unsuccessful retrieval, a status of 0326 is returned and current of set is nullified, but the next and prior of set are maintained; this enables you to continue accessing that set by using the FIND/OBTAIN WITHIN SET command.



Performing an Area Sweep

To access a record occurrence based on its physical position within an area, perform the following steps to establish the correct starting position:

1. Issue the FIND/OBTAIN FIRST/LAST/*n*th WITHIN *area-name* statement.
2. Check the ERROR-STATUS field for the value 0307 (DB-END-OF-SET).
3. Perform the IDMS-STATUS routine if a value other than 0307 is returned.

Accessing Subsequent Records

To retrieve subsequent record occurrences within an area, perform the following steps:

1. Issue the FIND/OBTAIN NEXT/PRIOR WITHIN *area-name* statement.
2. Check the ERROR-STATUS field for the value 0307 (DB-END-OF-SET).
3. Perform the IDMS-STATUS routine if a value other than 0307 is returned.

Relative Db-key Values

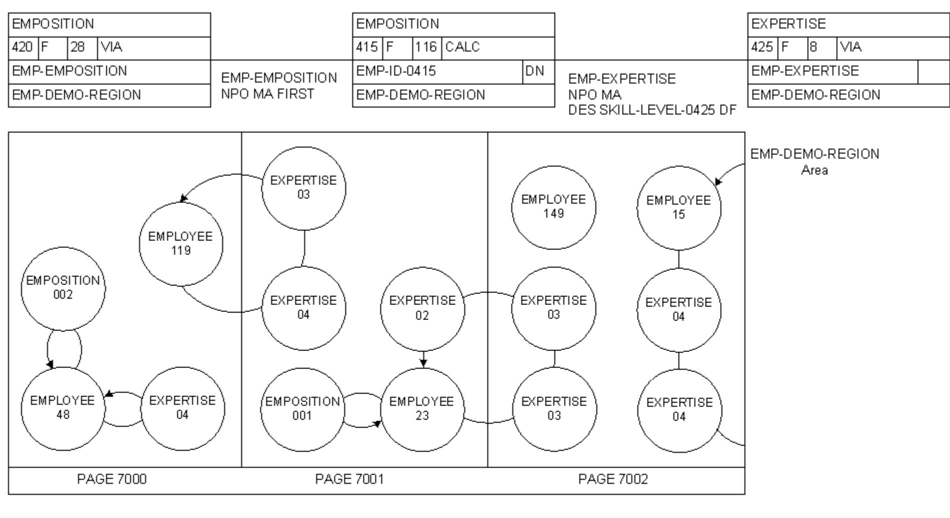
The first record occurrence in an area is the one with the lowest db-key; the last record has the highest db-key. The next record occurrence in an area is the one with the next higher db-key relative to the current record of the named area; the prior record is the one with the next lower db-key relative to the current of area.

Accessing Multiple Record Types

When accessing multiple records types while sweeping an area, be sure that the correct record occurrence is current of area before issuing the next FIND...WITHIN AREA. The easiest way to do this is by issuing the FIND CURRENT *record-name* statement each time before reissuing the OBTAIN NEXT WITHIN AREA statement. Failure to reestablish area currency ca CAuse your program to loop or skip records during retrieval.

The figure below shows retrieval of records within an area that contains multiple record types.

In this example, a sweep of the EMP-DEMO-REGION is performed, retrieving sequentially each EMPLOYEE record and all records in the associated EMPLOYEE-EXPERTISE set. The first command retrieves EMPLOYEE 119. Subsequent OBTAIN WITHIN SET statements retrieve the associated EXPERTISE records and establish currency on EXPERTISE 03. The FIND CURRENT statement is used to reestablish the proper position before retrieving EMPLOYEE 48. If FIND CURRENT EMPLOYEE is not specified, an attempt to retrieve the next EMPLOYEE record in the area would return EMPLOYEE 23.



	RUN UNIT	EMPLOYEE	EXPERTISE	EMP-EXPERTISE	EMP-DEMO-REGION
OBTAIN FIRST EMPLOYEE WITHIN EMP-DEMO-REGION.	119	119		119	119
OBTAIN FIRST EXPERTISE WITHIN EMP-EXPERTISE.	04	119	04	04	04
OBTAIN NEXT EXPERTISE WITHIN EMP-EXPERTISE.	03	119	03	03	03
FIND CURRENT EMPLOYEE.	119	119	03	119	119
OBTAIN NEXT EMPLOYEE WITHIN EMP-DEMO-REGION.	48	48	03	48	48

Area Sweep of the EMP-DEMO-REGION

The program excerpt below shows a program that sequentially retrieves all occurrences of the EMPLOYEE record in the EMP-DEMO-REGION.

```
 A000-MAIN-LINE.
      .
 A400-GET-FIRST.

*** RETRIEVE FIRST EMPLOYEE IN AREA ***
    OBTAIN FIRST EMPLOYEE WITHIN EMP-DEMO-REGION.
*** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET
      DISPLAY 'AREA EMPTY'
      FINISH
      GOBACK
    ELSE IF DB-STATUS-OK
      PERFORM A400-AREA-LOOP THRU A400-EXIT
         UNTIL DB-END-OF-SET
    ELSE
      PERFORM IDMS-STATUS.
    FINISH.
    GOBACK.
 A400-AREA-LOOP.
    DISPLAY 'EMPLOYEE: ' EMP-ID-0415
         'FIRST NAME: ' EMP-FIRST-NAME-0415
         'LAST NAME: ' EMP-LAST-NAME-0415.
*** RETRIEVE NEXT EMPLOYEE IN AREA ***
    OBTAIN NEXT EMPLOYEE WITHIN EMP-DEMO-REGION.
*** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET
      GO TO A400-EXIT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
 A400-EXIT.
    EXIT.
```

Accessing Owner Records

To access the owner record of the current record of set, perform the following steps:

1. Establish the current of set for the specified set type (for example, by issuing an OBTAIN CALC).
2. If the set is defined with either the optional or the manual set membership option, issue the IF MEMBER statement to determine set membership status.
3. Issue the FIND/OBTAIN OWNER command.
4. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

How FIND/OBTAIN OWNER Works

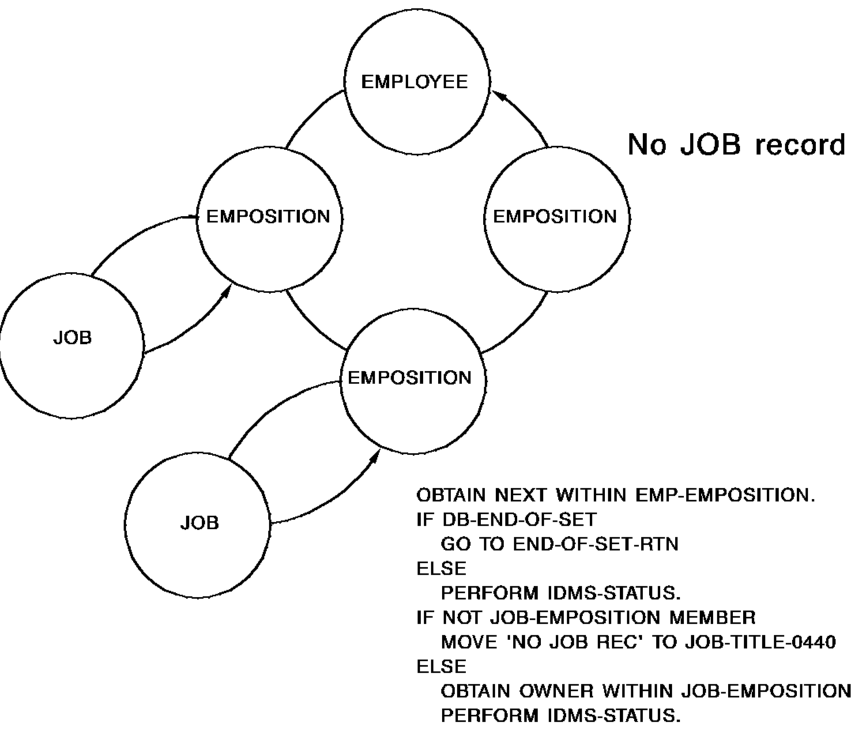
FIND/OBTAIN OWNER uses the *current of set* and locates the owner of that set occurrence.

Checking for Set Membership

Since an optional member may have been disconnected from the set and a manual member may never have been connected to the set, you cannot assume that such a record is actually connected to an occurrence of the set. Failure to check membership may result in obtaining the owner of another record, the one that is the current of set. If a member record is declared with either the optional or the manual set membership option, you should use the IF statement (explained in Checking for Set Membership) to determine whether the current record is presently connected to the specified set before issuing the FIND/OBTAIN OWNER command.

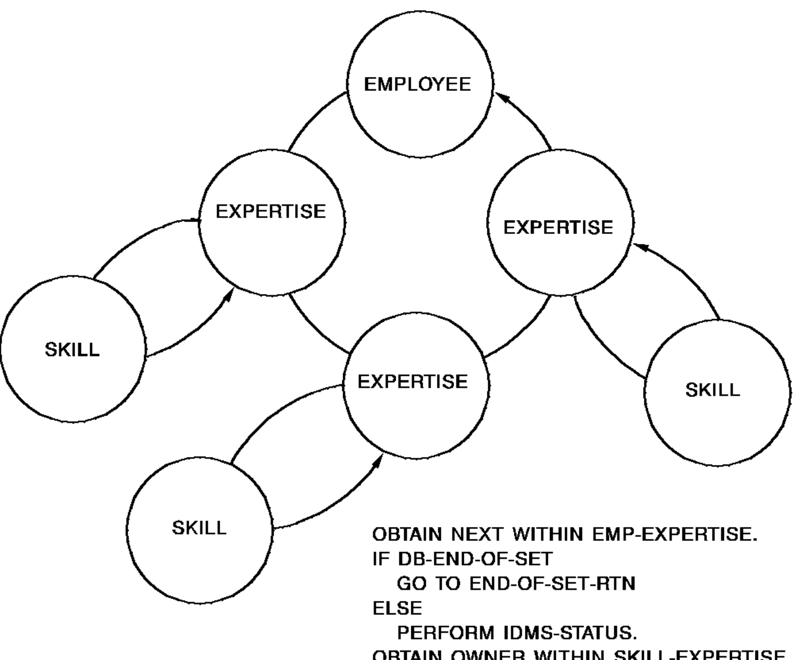
OWNER Retrieval in Optional or Manual Sets

The program excerpt and the figure below illustrate OWNER retrieval for sets with either the optional or the manual membership option. Records defined to sets with either the optional or the manual option may not be connected to a set occurrence; you can use the DML IF statement to test for set membership.



Owner Retrieval for Mandatory Automatic Sets

A member record declared as a *mandatory automatic* member of a set must be connected to an owner record. Such records are connected to a set occurrence when they are stored and cannot be disconnected. Therefore, you need not test for set membership prior to obtaining the set owner. The following program excerpt and figure illustrate OWNER retrieval for mandatory automatic sets.



Accessing a Record by Its db-key

The DBMS assigns a db-key to each record occurrence in the database. This key identifies the database page and line number where the record is located. The db-key can be qualified by record type or page information to ensure that it identifies a unique record occurrence. While always allowed, qualification is necessary only under the following circumstances:

- the subschema includes areas with different page information values
- the page information associated with the current of run unit is different than that of the record to be retrieved

For more information about qualifying db-keys, see DB-Keys and Page Information.

Steps to Access a Record by its Db-key

To access a record directly by using its db-key, perform the following steps:

1. Save the db-key of the record to be retrieved in a field defined as a binary fullword (COBOL PIC S9(8) COMP SYNC). Optionally save its record type or page information to use to qualify the db-key. For more information, see Saving a db-key.
2. Perform processing as required.
3. Issue the FIND/OBTAIN DB-KEY command using the saved db-key and qualifying information.
4. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

When to Use Access by Db-key

Using a record's db-key provides for the most efficient form of database retrieval. For example, if you know that your program will need to use a record more than once, it is best to save the record's db-key and reaccess the record by using FIND/OBTAIN DB-KEY. Any subschema record can be accessed by its db-key, regardless of location mode. Currency is not used to determine the target record of the FIND/OBTAIN DB-KEY statement; the record is identified by its db-key and, optionally, by its record type or page information.

'Native VSAM users'. The FIND/OBTAIN DB-KEY statement cannot be used to access records in a native VSAM key-sequenced data set (KSDS).

Example of Record Access by Db-key and Page-info

The program excerpt below shows using a db-key and page-info to reestablish currency.

Note: This application walks the DEPT-EMPLOYEE set, printing a report of all employees and their managers. After accessing the manager's EMPLOYEE record, the FIND DB-KEY statement is used to reestablish the correct EMPLOYEE record as current of the DEPT-EMPLOYEE set.

```
 WORKING-STORAGE SECTION.
 01  SAVED-DBKEYS.
    05 SAVE-EMP-DBKEY        PIC S9(8) COMP SYNC.
 PROCEDURE DIVISION.
      .
 A200-GET-EMP-MANAGER.
*** RETRIEVE EMPLOYEES SEQUENTIALLY WITHIN SET ***
    OBTAIN NEXT WITHIN DEPT-EMPLOYEE.
*** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET
      GO TO A200-EXIT
*** CHECK FOR ERROR-STATUS = 0000 ***
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
```

```
      PERFORM IDMS-STATUS.
*** SAVE EMPLOYEES' DB-KEY ***
  MOVE DBKEY TO SAVE-EMP-DBKEY.
    PERFORM IDMS-STATUS.
    MOVE EMP-ID-0415 TO EMP-ID-OUT.
    MOVE EMP-FIRST-NAME-0415 TO EMP-FIRST-OUT.
    MOVE EMP-LAST-NAME-0415 TO EMP-LAST-OUT.
    IF REPORTS-TO IS EMPTY
      DISPLAY 'EMPLOYEE ' EMP-ID-0415 'HAS NO MANAGER'
      GO TO A200-EXIT.
    FIND FIRST WITHIN REPORTS-TO.
    PERFORM IDMS-STATUS.
*** ACCESS MANAGER'S EMPLOYEE RECORD ***
    OBTAIN OWNER WITHIN MANAGES.
    PERFORM IDMS-STATUS.
    MOVE EMP-FIRST-NAME-0415 TO MANAGER-FIRST-OUT.
    MOVE EMP-LAST-NAME-0415 TO MANAGER-LAST-OUT.
*** REESTABLISH EMPLOYEE CURRENCY TO      ***
*** CONTINUE WALKING THE DEPT-EMPLOYEE SET ***
    FIND EMPLOYEE DB-KEY IS SAVE-EMP-DBKEY.
    PERFORM IDMS-STATUS.
 A200-EXIT.
    EXIT.
```

Example of Record Access by Db-key and Page-info

The program excerpt below shows using a db-key and page-info to reestablish currency.

Note: Use this coding technique when the subschema includes areas that have mixed page groups.

This application walks the DEPT-EMPLOYEE set, printing a report of all employees and their managers. After accessing the manager's EMPLOYEE record, the FIND DB-KEY statement is used to reestablish the correct EMPLOYEE record as current of the DEPT-EMPLOYEE set.

```
 WORKING-STORAGE SECTION.
 01  SAVED-DBKEY-PAGEINFO.

    05 SAVE-EMP-DBKEY        PIC S9(8) COMP SYNC.
    05 SAVE-EMP-PAGEINFO     PIC S9(8) COMP SYNC.
 PROCEDURE DIVISION.
      .
```

```
A200-GET-EMP-MANAGER.
*** RETRIEVE EMPLOYEES SEQUENTIALLY WITHIN SET ***
   OBTAIN NEXT WITHIN DEPT-EMPLOYEE.
*** CHECK FOR ERROR-STATUS = 0307 ***
   IF DB-END-OF-SET
     GO TO A200-EXIT
*** CHECK FOR ERROR-STATUS = 0000 ***
   ELSE IF DB-STATUS-OK
     NEXT SENTENCE
   ELSE
     PERFORM IDMS-STATUS.
*** SAVE EMPLOYEES' DB-KEY and PAGE-INFO ***
     MOVE DBKEY TO SAVE-EMP-DBKEY.
     MOVE PAGE-INFO TO SAVE-EMP-PAGEINFO.
   PERFORM IDMS-STATUS.
   MOVE EMP-ID-0415 TO EMP-ID-OUT.
   MOVE EMP-FIRST-NAME-0415 TO EMP-FIRST-OUT.
   MOVE EMP-LAST-NAME-0415 TO EMP-LAST-OUT.
   IF REPORTS-TO IS EMPTY
     DISPLAY 'EMPLOYEE ' EMP-ID-0415 'HAS NO MANAGER'
     GO TO A200-EXIT.
   FIND FIRST WITHIN REPORTS-TO.
   PERFORM IDMS-STATUS.
*** ACCESS MANAGER'S EMPLOYEE RECORD ***
   OBTAIN OWNER WITHIN MANAGES.
   PERFORM IDMS-STATUS.
   MOVE EMP-FIRST-NAME-0415 TO MANAGER-FIRST-OUT.
   MOVE EMP-LAST-NAME-0415 TO MANAGER-LAST-OUT.
*** REESTABLISH EMPLOYEE CURRENCY TO       ***
*** CONTINUE WALKING THE DEPT-EMPLOYEE SET ***
   FIND DB-KEY IS SAVE-EMP-DBKEY PAGE-INFO SAVE-EMP-PAGEINFO.
   PERFORM IDMS-STATUS.
A200-EXIT.
   EXIT.
```

Accessing Indexed Records

Indexes provide an efficient means of accessing member record occurrences. You can retrieve member records in indexed sets as if they were member records in nonindexed sets.

The table below lists the retrieval statements that you can use with indexed records.

Retrieval statement	Restrictions
FIND/OBTAIN CURRENT	WITHIN SET

Retrieval statement	Restrictions
FIND/OBTAIN RECORD	WITHIN SET
FIND/OBTAIN USING SORT KEY	Sorted indexed sets only
FIND/OBTAIN OWNER	OBTAIN not allowed for system-owned indexes
RETURN	Db-key and symbolic key only

Example of Accessing an Indexed Record

The program excerpt below shows retrieval of all records in the EMP-NAME-NDX (a system-owned indexed set).

The EMP-NAME-NDX set is sorted in ascending order on EMP-LAST-NAME and EMP-FIRST-NAME; this program produces an alphabetical list of all employees.

```
 PROCEDURE DIVISION.
 A000-MAIN-LINE.
 .
 .
 .
     MOVE 'Y' to FIRST-TIME-SW.
     PERFORM A000-GET-NDX-SET THRU A000-EXIT
        UNTIL DB-END-OF-SET.
     PERFORM END-PROCESSING.
     GOBACK.

 A000-GET-NDX-SET.
*** SEQUENTIALLY RETRIEVE EMPLOYEES INDEXED BY LAST NAME ***
   IF FIRST-TIME-SW = 'Y'
     MOVE 'N' TO FIRST-TIME-SW
     OBTAIN FIRST EMPLOYEE WITHIN EMP-NAME-NDX
   ELSE
     OBTAIN NEXT EMPLOYEE WITHIN EMP-NAME-NDX.
*** CHECK FOR ERROR-STATUS = 0307 ***
   IF DB-END-OF-SET
     GO TO A000-EXIT
*** CHECK FOR ERROR-STATUS = 0000 ***
   ELSE IF DB-STATUS-OK
     NEXT SENTENCE
   ELSE
     PERFORM IDMS-STATUS.
   DISPLAY EMP-ID-0415
        EMP-LAST-NAME-0415
        EMP-FIRST-NAME-0415.
 A000-EXIT.
   EXIT.
```

Retrieving the Key without the Record

To retrieve the db-key and symbolic key of an indexed record without retrieving the record itself, perform the following steps:

1. Initialize variable storage fields, as required.
2. Issue the RETURN statement.
3. If you are issuing the RETURN statement iteratively, check for an ERROR-STATUS of 1707; if you are doing a keyed search, check for an ERROR-STATUS of 1726.
4. Perform the IDMS-STATUS routine if 1726, 1707, or 0000 is not returned.

Using the RETURN Statement

The RETURN statement establishes currency in the indexed set and moves the record's symbolic key into the data fields within the record in program variable storage. Alternatively, you can move the record's symbolic key into some other specified variable-storage location.

Example of Using RETURN

The program excerpt below uses the RETURN statement to establish indexed set currency.

This program establishes currency in the EMP-NAME-NDX set by using the RETURN statement to perform a generic-key search. It checks for the ERROR-STATUS 1726 (record not found), and retrieves all employees whose last name begins with the letter N or greater.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  INDEX-ITEMS.
     03 DB-KEY-V          PIC S9(8) COMP SYNC.
     03 INDEX-START-POINT PIC X(30)   VALUE 'N'.
 PROCEDURE DIVISION.
 A000-MAIN-LINE.
 .
 .
 .
     MOVE INDEX-START-POINT TO INDEX-KEY-VALUE.
     RETURN DB-KEY-V FROM EMP-NAME-NDX
         USING INDEX-START-POINT.
*** IF NO MATCHING EMPLOYEE, OBTAIN NEXT IN SET ***
     IF ERROR-STATUS = '1726' THEN
       OBTAIN NEXT EMPLOYEE WITHIN EMP-NAME-NDX.
     IF DB-STATUS-OK
       NEXT SENTENCE
```

```
  ELSE
    PERFORM IDMS-STATUS.
  PERFORM A000-GET-NDX-SET THRU A000-EXIT
     UNTIL DB-END-OF-SET.
  FINISH.
  GOBACK.
A000-GET-NDX-SET.
  DISPLAY  EMP-ID-0415
        EMP-LAST-NAME-0415
        EMP-FIRST-NAME-0415.
  OBTAIN NEXT EMPLOYEE WITHIN EMP-NAME-NDX.
*** CHECK FOR ERROR-STATUS = 0307 ***
  IF DB-END-OF-SET
    GO TO A000-EXIT
*** CHECK FOR ERROR-STATUS = 0000 ***
  ELSE IF DB-STATUS-OK
    NEXT SENTENCE
  ELSE
    PERFORM IDMS-STATUS.
A000-EXIT.
  EXIT.
```

Saving db-key, Page Information, and Bind Addresses

Retrieving a record by using its db-key is the most efficient form of retrieval. If you know that you will use a record later in your program, you should save its db-key in order to reaccess the record by using db-key retrieval. In certain circumstances a db-key used to access a record may require qualification by record type or page information. You can save page information when saving a db-key or by issuing a DML request.

For information about direct access to a record, see Accessing a Record by Its db-key.

For more information about qualifying db-keys, see DB-Keys and Page Information.

ACCEPT statements (also called save statements) transfer db-keys, page information, and storage-addresses from the DBMS to program variable storage. These statements are an efficient means of obtaining information at run time because they usually cause no database I/O.

Saving a db-key, page information and a bind address are explained below.

Saving a db-key

You can retrieve a db-key using one of these methods:

- **Accepting the db-key of a current record**. You can retrieve the db-key of the record that is current of run unit, record type, set, or area using the ACCEPT DB-KEY FROM CURRENCY statement.

  Note: You can also retrieve the db-key of the record that is current of run unit from the DBKEY field of the IDMS communications block. You can retrieve its page information from the PAGE-INFO field of the IDMS communications block.

- **Accepting a db-key relative to the current record**. You can use an ACCEPT DB-KEY RELATIVE TO CURRENCY statement to retrieve the db-key of the NEXT, PRIOR, or OWNER record relative to the current record of set.

Steps in Saving a db-key

To save a db-key, perform the following steps:

1. Establish the appropriate currency for the required record.
2. Perform one of the following steps:
   - **If the required record is current of run unit**, move the DBKEY field in the IDMS communications block to a variable storage field defined as a binary fullword (COBOL PIC S9(8) COMP SYNC).
   - **If the required record is not current of run unit**, issue either the ACCEPT DBKEY FROM CURRENCY or the ACCEPT DBKEY RELATIVE TO CURRENCY statement, storing the saved db-key in a variable storage field defined as a binary fullword (COBOL PIC S9(8) COMP SYNC).
3. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

Important! You should not save db-keys or page information outside of the program because these values can change if the database is unloaded and reloaded, if record occurrences are erased or if an area is assigned to a different page group.

Example of Using db-keys

The program excerpt below shows a program that compares db-keys. The first db-key is acquired from the IDMS communications block, the second by using an ACCEPT DB-KEY statement.

This application compares the db-key of each JOB record with JOB owner db-keys in EMPOSITION records in the JOB-EMPOSITION set. When the db-keys match, the program accesses the EMPOSITION information by issuing a GET statement.

```
WORKING-STORAGE SECTION.
01  JOB-DBKEY         PIC S9(8)  COMP.
01  MATCH-DBKEY       PIC S9(8)  COMP.
.
.
.
PROCEDURE DIVISION.
.
.
.
    PERFORM A100-GET-EMP-JOB THRU A100-EXIT
        UNTIL END-OF-FILE.
.
.
.
A100-GET-EMP-JOB.
    MOVE GETEMP-ID-IN TO EMP-ID-0415.
    OBTAIN CALC EMPLOYEE.
*** CHECK FOR ERROR-STATUS = 0326 ***
    IF DB-REC-NOT-FOUND
      DISPLAY 'EMP NOT FOUND: ' GETEMP-ID-IN
      GO TO A100-GET-NEXT
*** CHECK FOR ERROR-STATUS = 0000 ***
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
    MOVE GETJOB-ID-IN TO JOB-ID-0440.
    OBTAIN CALC JOB.
*** CHECK FOR ERROR-STATUS = 0326 ***
    IF DB-REC-NOT-FOUND
      DISPLAY 'JOB NOT FOUND: ' GETJOB-ID-IN
      GO TO A100-GET-NEXT
*** CHECK FOR ERROR-STATUS = 0000 ***
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
```

```
*** SAVE JOB DB-KEY ***
    MOVE DBKEY TO JOB-DBKEY.
    IF EMP-EMPOSITION IS EMPTY
       DISPLAY 'EMP-EMPOSITION IS EMPTY FOR: ' GETEMP-ID-IN
       GO TO A100-GET-NEXT
    ELSE
       PERFORM A200-LOOP THRU A200-EXIT.

 A100-GET-NEXT.
    READ GET-FILE-IN AT END MOVE 'Y' TO EOF-SW.
 A100-EXIT.
    EXIT.
 A200-LOOP.
    FIND NEXT WITHIN EMP-EMPOSITION.
*** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET
      GO TO A200-EXIT
*** CHECK FOR ERROR-STATUS = 0000 ***
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
*** ACCESS DB-KEY OF OWNER IN JOB-EMPOSITION SET ***
    ACCEPT MATCH-DBKEY FROM JOB-EMPOSITION
        OWNER CURRENCY.
    IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
*** IF DB-KEYS ARE NOT EQUAL, LOOP AND TRY AGAIN ***
    IF JOB-DBKEY NOT = MATCH-DBKEY
     THEN
       GO TO A200-LOOP
    ELSE
      NEXT SENTENCE.
*** IF DB-KEYS ARE EQUAL, ACCESS THE EMPOSITION DATA ***
    GET EMPOSITION.
    IF NOT DB-STATUS-OK
     PERFORM IDMS-STATUS
    ELSE NEXT SENTENCE.
    PERFORM A300-PRINT-DATA.
 A200-EXIT.
    EXIT.
 .
 .
 .
```

Inferring Information

For indexed sets and chained sets with prior pointers, the ACCEPT DB-KEY RELATIVE TO CURRENCY statement can also be used to infer information, as shown in the program excerpt below.

This application erases all DEPARTMENT records that contain less than two EMPLOYEE records. The first ACCEPT statement tests for zero EMPLOYEE records; the second ACCEPT statement tests for one.

```
WORKING-STORAGE SECTION.
01  SAVED-DBKEYS.
    05 NEXT-DEPT-EMP-DBKEY     PIC S9(8) COMP SYNC.
    05 PRIOR-DEPT-EMP-DBKEY    PIC S9(8) COMP SYNC.
PROCEDURE DIVISION.
A100-LEAN-AND-FAST.
    OBTAIN FIRST DEPARTMENT WITHIN ORG-DEMO-REGION.
*** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET THEN
      GO TO EMPTY-AREA
    ELSE
      PERFORM IDMS-STATUS.
    PERFORM A200-ACCEPT-AND-TEST THRU A200-EXIT
            UNTIL DB-END-OF-SET.
    FINISH.
    GOBACK.
A200-ACCEPT-AND-TEST.
*** RETRIEVE NEXT DB-KEY ***
    ACCEPT NEXT-DEPT-EMP-DBKEY FROM
      DEPT-EMPLOYEE NEXT CURRENCY.
      PERFORM IDMS-STATUS.
*** CHECK FOR EMPTY SET ***
*** IF DB-KEYS ARE THE SAME, THE SET IS EMPTY ***
    IF NEXT-DEPT-EMP-DBKEY = DBKEY THEN
      ERASE DEPARTMENT PERMANENT
      PERFORM IDMS-STATUS
      GO TO A200-GET-NEXT.
*** CHECK FOR ONE-MEMBER SET ***
    ACCEPT PRIOR-DEPT-EMP-DBKEY FROM
        DEPT-EMPLOYEE PRIOR CURRENCY.
```

```
    PERFORM IDMS-STATUS.
*** IF DB-KEYS ARE THE SAME, THE SET HAS ONE MEMBER ***
    IF NEXT-DEPT-EMP-DBKEY =
           PRIOR-DEPT-EMP-DBKEY THEN
      ERASE DEPARTMENT PERMANENT
      PERFORM IDMS-STATUS
      GO TO A200-GET-NEXT
    ELSE
      GO TO A200-GET-NEXT.
 A200-GET-NEXT.
    OBTAIN NEXT DEPARTMENT WITHIN ORG-DEMO-REGION.
*** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET THEN
      GO TO A200-EXIT
    ELSE
      PERFORM IDMS-STATUS.
 A200-EXIT.
    EXIT.
```

Saving Page Information

You can retrieve page information using one of these methods:

- Moving the page information of the record that is current of run unit from the PAGE-INFO field of the IDMS communications block.
- Accepting page information for a record type by using an ACCEPT PAGE-INFO statement.

Steps in Saving Page Information

To save page information, perform the following steps:

1. If the required record is current of run unit, move the PAGE-INFO field of the IDMS communications block to a variable storage field.
2. If you know the record type for which page information is desired:
   - Issue the ACCEPT PAGE-INFO statement, storing the output in a variable storage field.
   - Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

The variable storage field used to hold page information can either be defined as a binary fullword field (COBOL PIC S9(8) COMP SYNC) or as a group item consisting of two contiguous binary halfwords. In COBOL this might look as:

```
01 <group-field-name>.

   02 <page-group-field-name>  PIC S9(4) COMP SYNC.
   02 <dbkey-radix-field-name> PIC S9(4) COMP SYNC.
```

The latter definition enables the components of the page information to be accessed independently.

Important! You should not save page information outside of the program because the value can change if the database is unloaded and reloaded or if an area is assigned to a different page group.

Example of Using Page Information

The following example retrieves the page information for the DEPARTMENT record and uses the db-key radix to separate a db-key's page and line numbers.

Saving a Record's BIND Address

To access a database record from a subprogram, you may need to know its storage address. You can use the ACCEPT BIND ADDRESS statement to acquire the storage address of a record that was bound in the calling program.

Because the ACCEPT BIND ADDRESS statement returns a storage address, it is typically used with subroutines.

For more information on the ACCEPT BIND ADDRESS statement, see the language-specific *CA IDMS DML Reference Guide*.

Checking for Set Membership

When accessing the database, you may find it necessary to obtain information about an owner or member record's set-membership status. To obtain set-specific information for a record occurrence, use the IF statement. By using the IF statement, you can determine:

- If the current occurrence of a specified set contains any member record occurrences (Is it empty?)
- If the current record of run unit participates as a member in a specified set defined with either the optional or the manual set membership option (Is it currently connected to an occurrence of the specified set?)

Each IF statement contains a conditional phrase and an imperative statement that specifies further action based on the outcome of the evaluation.

'Native VSAM users'. The IF statement is not allowed for sets defined with member records that are stored in native VSAM data sets.

Use of the IF EMPTY statement and the IF MEMBER statement are discussed below.

Using the IF EMPTY Statement

After you have retrieved the owner record in a set, you can issue the IF EMPTY statement to determine if the set owns any member record occurrences. This allows you to control processing based on whether the set is empty.

Steps in Determining if a Set Is Empty

To determine if a set is empty, perform the following steps:

1. Establish currency for the set.
2. Issue the IF EMPTY statement.
3. Perform further processing as specified.

Note: If the set contains a member record, the first record in the set is always accessed during the processing of an IF EMPTY DML statement, in order to determine whether or not it is logically deleted. This can result in additional I/Os particularly if the member records are not stored VIA the set being tested.

How to Avoid an OBTAIN

You can also use the IF EMPTY statement to eliminate the need for using an OBTAIN FIRST WITHIN SET statement to walk a set, as illustrated in the program excerpt below.

Because the IF EMPTY statement determines that the set is not empty, you can be assured that the program can read at least one EMPLOYEE record before an ERROR-STATUS of 0307 (DB-END-OF-SET) is returned.

```
.
.
.
  IF DEPT-EMPLOYEE IS EMPTY
    MOVE NO-EMP-MESSAGE TO TITLE-OUT
  ELSE
    PERFORM A100-DEPT-EMP-WALK THRU A100-EXIT
      UNTIL DB-END-OF-SET.
A100-DEPT-EMP-WALK.
  OBTAIN NEXT WITHIN DEPT-EMPLOYEE.
.
.
.
```

Using the IF MEMBER Statement

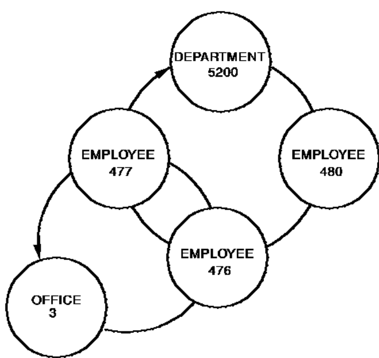
If the current record of run unit participates as a member in a set defined with either the optional or the manual set membership option, you cannot assume that that record occurrence is also current of set. For example, an optional record may never have been connected to the set, or a manual record may have been disconnected from the set.

For more information about optional and manual set membership, see Set Membership Options.

Failure to Test for Set Membership

The figure below shows the invalid conclusion that can result from not testing for set membership.

Since EMPLOYEE 480 is not currently connected to an occurrence of the OFFICE-EMPLOYEE set, the OBTAIN OWNER statement retrieves the owner of the current record of set (OFFICE 3). This leads to the invalid assumption that EMPLOYEE 480 works in OFFICE 3.



	R. U. curr.	Record currencies			Set currencies		Area currencies	
	RUN UNIT	DEPARTMENT	EMPLOYEE	OFFICE	DEPT-EMPLOYEE	OFFICE-EMPLOYEE	ORG-DEMO-REGION	EMP-DEMO-REGION
PREVIOUSLY ESTABLISHED CURRENCIES	5200	5200	477	3	5200	477	477	5200
OBTAIN FIRST WITHIN DEPT-EMPLOYEE.	480	5200	480	3	480	477	480	5200
OBTAIN OWNER WITHIN OFFICE-EMPLOYEE.	3	5200	480	3	480	3	480	3

Steps in Testing for Set Membership

You can issue the IF MEMBER statement to ensure that a record occurrence currently participates as a member of a specified set.

To determine if a record participates as a member in a set, perform the following steps:

1. Establish run unit currency for the specified member record.
2. Issue the IF MEMBER statement.
3. Perform further processing, as specified

The program excerpt below uses the IF EMPTY and the IF MEMBER statements to facilitate database navigation.

The IF EMPTY statement determines if the DEPT-EMPLOYEE set is empty; the IF MEMBER statement determines whether the current of run unit (EMPLOYEE) participates as a member in the OFFICE-EMPLOYEE set.

```
 PROCEDURE DIVISION.
 .
 .
 .
 A100-EMP-DEPT-OFF.
    MOVE DEPT-ID-IN TO DEPT-ID-0410.
    OBTAIN CALC DEPARTMENT
    IF DB-REC-NOT-FOUND
      GO TO GET-NEXT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
*** TEST TO SEE IF SET IS EMPTY ***
    IF DEPT-EMPLOYEE IS EMPTY
       MOVE NO-EMP-MESSAGE TO TITLE-OUT
    ELSE
       PERFORM A100-WALK THRU A100-EXIT
          UNTIL DB-END-OF-SET.
 A100-WALK.
    OBTAIN NEXT WITHIN DEPT-EMPLOYEE.
*** CHECK FOR ERROR-STATUS = 0307 ***
    IF DB-END-OF-SET
      GO TO A100-EXIT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
*** TEST TO SEE IF EMPLOYEE IS CURRENTLY CONNECTED TO THE SET ***
    IF NOT OFFICE-EMPLOYEE MEMBER
       MOVE NO-OFF-MESSAGE TO TITLE-OUT
    ELSE
       OBTAIN OWNER WITHIN OFFICE-EMPLOYEE
       PERFORM IDMS-STATUS
       MOVE OFFICE-ADDRESS-0450 TO ADDRESS-OUT.
    PERFORM U100-PRINT.
  A100-EXIT.
    EXIT.
```

Updating the Database

DML modification statements update record occurrences in the database. By using these statements, which are discussed separately on the following pages, you can:

- Store a new record occurrence in the database
- Modify the contents of an existing record
- Erase a record from the database
- Connect a member record to a set
- Disconnect a member record from a set

Storing Records

To add a record occurrence to the database, perform the following steps:

1. Specify a subschema that includes:
   - All sets in which the stored record is defined as an automatic member
   - The owner record of each of the required automatic sets

   Note: Sets for which the stored record is defined as a manual member need not be defined in the subschema because the STORE statement does not access those sets. (An automatic member is connected automatically to the selected set occurrence when the record is stored; a manual member is not connected automatically to the selected set occurrence.)

2. Ready all affected areas in one of the update usage modes (for more information, see Area Usage Modes).

   Areas should be readied whether they are affected explicitly or implicitly (for example, as owner of a mandatory automatic set whose members are being stored).

3. Initialize the following variable storage fields:
   - All CALC, index, sort-key, and data fields
   - If the record being stored has a location mode of DIRECT, initialize the contents of the DIRECT-DBKEY field in the IDMS communications block with a suggested db-key value or a null db-key value of -1.
   - If the record is to be stored in a native VSAM relative-record data set (RRDS), initialize the contents of the DIRECT-DBKEY field with the relative record number that represents the location within the data set where the record is to be stored.

4. Establish currency for all set occurrences in which the stored record will participate as an *automatic* member. Depending on the set order, the stored record occurrence is positioned as follows:

   ■ **If the named record is defined as a member of a set that is ordered FIRST or LAST**, the record that is current of set establishes the set occurrence to which the new record will be connected.

   ■ **If the named record is defined as a member of a set that is ordered NEXT or PRIOR**, the record that is current of set establishes the set occurrence into which the new record will be connected *and* determines its position within the set.

   ■ **If the named record is defined as a member of a sorted set**, the record that is current of set establishes the set occurrence into which the new record will be connected. The DBMS compares the sort key of the new record with the sort key of the current record of set to determine if the new record can be inserted into the set by movement in the next direction. If it can, the current of set remains positioned at the record that is current of set and the new record is inserted. If it cannot, the DBMS finds the owner of the current of set (not necessarily the current occurrence of the owner record type) and moves as far forward in the next direction as is necessary to determine the logical insertion point for the new record.

   If the record being stored has a location mode of VIA, currency must be established for that VIA set, regardless of whether the record being stored is an automatic or manual member of that set. Current of the VIA set provides the suggested page for the record being stored.

5. Issue the STORE command.

6. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

What STORE Does

STORE performs the following functions:

■ Acquires space and assigns a database key for a new record occurrence in the database

■ Transfers the value of the appropriate elements from program variable storage to the space acquired for the record occurrence in the database

■ Connects the new record occurrence to all sets for which it is defined as an automatic member

The program excerpt below shows storing records in the database.

The program establishes the proper DEPARTMENT and OFFICE currencies and stores the new EMPLOYEE record.

```
 PROCEDURE DIVISION.
        .

    READ NEW-EMP-FILE-IN.
        AT END MOVE 'Y' TO EOF-SW.
    PERFORM A300-STORE-EMP THRU A300-EXIT
                  UNTIL END-OF-FILE.
    FINISH.
    GOBACK.
 A300-STORE-EMP.
    MOVE DEPT-ID-IN TO DEPT-ID-0410.
*** ESTABLISH CORRECT DEPARTMENT CURRENCY ***
    FIND CALC DEPARTMENT.
*** CHECK FOR ERROR-STATUS = 0326 ***
    IF DB-REC-NOT-FOUND
      THEN DISPLAY
      'DEPARTMENT ' DEPT-ID-IN ' NOT FOUND'
      'FOR NEW EMPLOYEE ID ' EMP-ID-IN
      GO TO A300-GET-NEXT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
    MOVE OFFICE-CODE-IN TO OFFICE-CODE-0450.
*** ESTABLISH CORRECT OFFICE CURRENCY ***
    FIND CALC OFFICE.
*** CHECK FOR ERROR-STATUS = 0326 ***
    IF DB-REC-NOT-FOUND
      THEN DISPLAY
      'OFFICE ' OFFICE CODE-IN ' NOT FOUND'
      'FOR NEW EMPLOYEE ID ' NEW-EMP-ID
      GO TO A300-GET-NEXT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
```

```
    PERFORM B300-INITIALIZE-EMPLOYEE.
*** STORE EMPLOYEE RECORD ***
    STORE EMPLOYEE.
    PERFORM IDMS-STATUS.
    PERFORM U500-WRITE-NEW-EMP-REPORT.
 A300-GET-NEXT.
    READ NEW-EMP-FILE-IN
        AT END MOVE 'Y' TO EOF-SW.
 A300-EXIT.
    EXIT.
```

Modifying Records

To change a record occurrence in the database, perform the following steps:

1. Ready all affected areas in one of the update usage modes (for more information, see Area Usage Modes).

   Areas should be readied whether they are affected explicitly or implicitly (for example, as owner or member of a set whose members' sort keys are being modified).

2. Establish the specified record as current of run unit by issuing either a FIND or an OBTAIN statement.

3. Change the variable-storage fields of the record to be modified.

   When using FIND, be sure to initialize *all* the appropriate values of the record to be modified. The best practice, however, is to use the OBTAIN statement to ensure that all the elements in the modified record are present in variable storage.

4. Issue the MODIFY command.

5. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

CALC and Sort Key Considerations

The following special considerations apply to the modification of CALC- and sort-keys:

- If modification of a CALC- or sort-key will violate a duplicates-not-allowed option, the record is not modified and an error condition results.
- If a CALC-key is modified, successful execution of the MODIFY statement enables the record to be accessed on the basis of its new CALC-key value. The db-key of the specified record is not changed.

- If a sort-key is to be modified, the sorted set in which the specified record participates must be included in the subschema invoked by the program. A record occurrence that is a member of a set not defined in the subschema can be modified only *if the undefined set is not sorted.*
- If any of the modified elements in the specified record are defined as sort keys for any set occurrence in which that record is currently a member, the DBMS tests that set occurrence to ensure that set order is maintained. If necessary, the DBMS disconnects the specified record and reconnects it in the set occurrence to maintain the set order specified in the schema.

Native VSAM Considerations

The length of a record in an entry-sequenced data set (ESDS) cannot be changed even in the case of variable-length records.

The prime key for a key-sequenced data set (KSDS) cannot be modified.

Example of Modifying Records

The program excerpt below modifies records in the database.

The program retrieves the specified EMPLOYEE record and modifies the address and phone number. This program issues a COMMIT statement after every 100 updates. COMMIT releases all implicit exclusive locks and writes a checkpoint to the journal file.

```
 WORKING-STORAGE SECTION.
 01  COMMIT-COUNTER            PIC S9(4) COMP VALUE +0.
 PROCEDURE DIVISION.
        .
    READ NEW-EMP-ADDRESS-FILE-IN.

       AT END MOVE 'Y' TO EOF-SW.
    PERFORM A300-CHANGE-ADDRESS THRU A300-EXIT
               UNTIL END-OF-FILE.
    FINISH.
    GOBACK.
 A300-CHANGE-ADDRESS.
    MOVE EMP-ID-IN TO EMP-ID-0415.
*** RETRIEVE EMPLOYEE RECORD ***
    OBTAIN CALC EMPLOYEE.
*** CHECK FOR ERROR-STATUS = 0326 ***
    IF DB-REC-NOT-FOUND
      THEN DISPLAY
      'EMPLOYEE ' EMP-ID-IN ' NOT FOUND'
      GO TO A300-GET-NEXT
```

```
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
    PERFORM U500-WRITE-OLD-ADDRESS.
*** CHANGE DATA AND ISSUE THE MODIFY STATEMENT ***
    MOVE NEW-ADDRESS-IN TO EMP-ADDRESS-0415.
    MOVE NEW-PHONE-IN   TO EMP-PHONE-0415.
    MODIFY EMPLOYEE.
    PERFORM IDMS-STATUS.
    ADD 1 TO COMMIT-COUNTER.
    IF COMMIT-COUNTER > 100 THEN
      COMMIT
      PERFORM IDMS-STATUS
      MOVE 0 TO COMMIT-COUNTER.
    PERFORM U0510-WRITE-NEW-ADDRESS.
 A300-GET-NEXT.
    READ NEW-EMP-ADDRESS-FILE-IN
        AT END MOVE 'Y' TO EOF-SW.
 A300-EXIT.
    EXIT.
```

Erasing Records

To delete a record occurrence from the database, perform the following steps:

1. Specify a subschema that includes the following:
   - All sets in which the specified record participates as owner either directly or indirectly (for example, as owner of a set with a member that is owner of another set)
   - All member record types in the sets specified above
2. Ready all affected areas in one of the update usage modes (for more information, see Area Usage Modes).

   Areas should be readied whether they are affected explicitly or implicitly (for example, as owner of a set whose members are being erased).
3. Establish the specified record as current of run unit.
4. Issue the ERASE command.
5. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

What ERASE Does

The ERASE statement performs the following functions:

- Disconnects the specified record from all set occurrences in which it participates as a member and logically or physically deletes the record from the database

- Optionally erases all records that are mandatory members of set occurrences owned by the specified record
- Optionally disconnects or erases all records that are optional members of set occurrences owned by the specified record

ERASE is a two-step procedure that first cancels the existing membership of the named record in specific set occurrences and then releases for reuse the space occupied by the named record and its db-key. Erased records are unavailable for further processing by any DML statement.

Currencies after an ERASE

Following successful execution of an ERASE statement:

- Currency is *nullified* for all record types involved in the erase, both explicitly and implicitly.
- Currency is *preserved* for run unit and area.
- Next, prior, and owner currencies are *preserved* for sets from which the last record occurrence was erased.

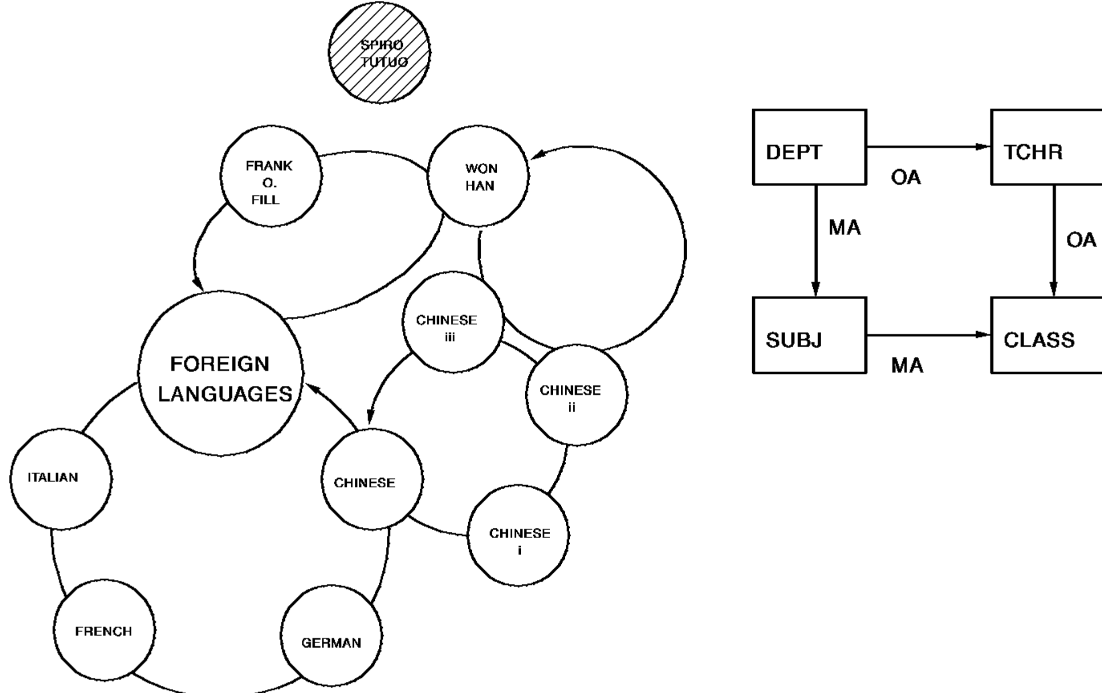
  The preserved currencies enable you to retrieve the next or prior records within the area or the next, prior, or owner records within a set in which the erased record participated.

ERASE Statement with No Options

To issue the ERASE statement with no options:

- The record must be current of run unit.
- All sets in which the record participates as owner must be empty. (An error condition will result if this version of the ERASE statement is attempted against an owner record that has any member occurrences).

In the illustration below, an ERASE TCHR statement with no options disconnects the shaded occurrence (SPIRO TUTUO) from membership in the DEPT-TCHR set and then erases the record occurrences. This statement executes without error because the TCHR-CLASS set owned by record occurrence SPIRO TUTUO is an empty set (he doesn't have any classes).



ERASE Options

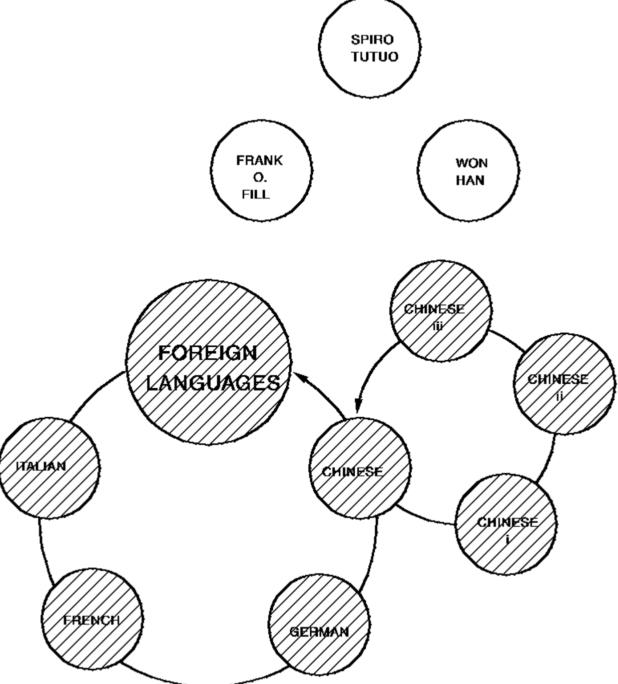
You can qualify the ERASE statement with these options to specify how the ERASE statement affects member occurrences:

- PERMANENT
- SELECTIVE
- ALL

ERASE PERMANENT

ERASE PERMANENT erases the specified record and all mandatory member record occurrences owned by the specified record. Optional member records are disconnected. If any of the erased mandatory members are themselves the owners of any set occurrences, they are erased as if they were directly the object record of an ERASE PERMANENT statement (that is, all mandatory members of such sets are also erased). This process continues until all direct and indirect members have been processed.

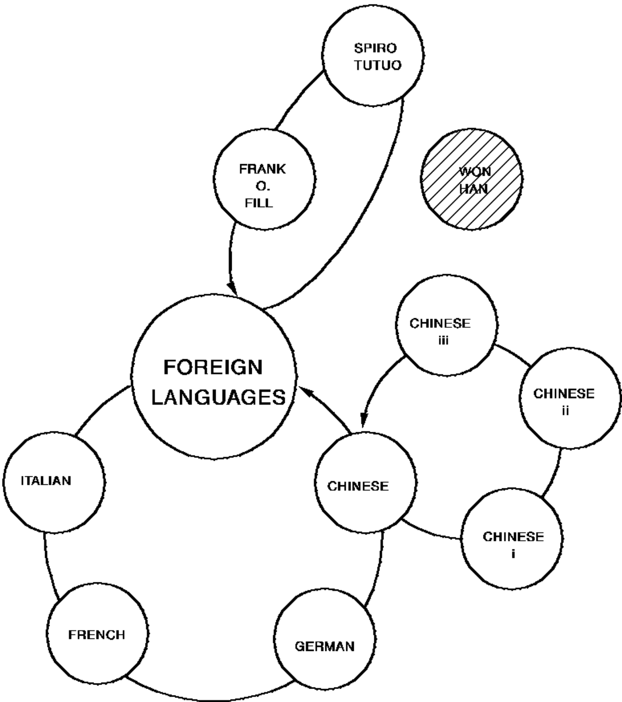
In the illustration below, currency has been set on the FOREIGN LANGUAGES occurrence of the DEPT record, and an ERASE DEPT PERMANENT statement has been issued. All subjects are erased because they are mandatory members of the DEPT-SUBJ set. All classes are also erased because they are mandatory members of the SUBJ-CLASS set. However, since membership in DEPT-TCHR is optional, members of the set owned by FOREIGN LANGUAGES are disconnected, not erased.



ERASE SELECTIVE

ERASE SELECTIVE erases the specified record and all mandatory member record occurrences owned by the specified record. Optional member records are erased if they do not *currently participate* as members in other set occurrences. All erased member records that are themselves the owners of any set occurrences are treated as if they were the object of an ERASE SELECTIVE statement.

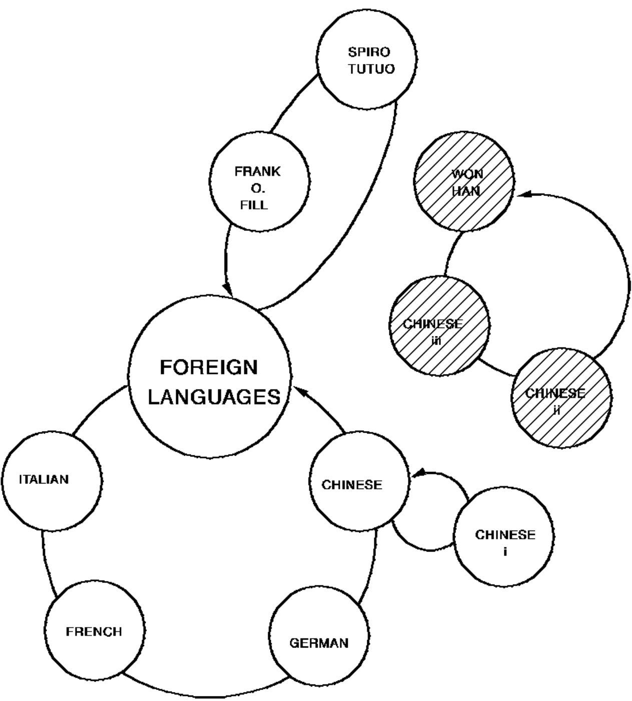
In the illustration below, currency has been set on the WON HAN occurrence of the TCHR record, and an ERASE TCHR SELECTIVE statement has been issued. Since WON HAN was the owner of two occurrences of the TCHR-CLASS set, an ERASE statement without an option would fail. The SELECTIVE option prevents these occurrences from being erased because they currently participate in another set (SUBJ-CLASS). This means, in effect, that the department still offers the classes even though the teacher is gone.



ERASE ALL

ERASE ALL erases the specified record and all mandatory and optional member record occurrences owned by the specified record. All erased member records that are themselves the owners of any set occurrences are treated as if they were the object record of an ERASE ALL statement.

In the illustration below, currency has been set on the WON HAN occurrence of the TCHR record, and an ERASE TCHR ALL statement has been issued. Since WON HAN was the owner of two occurrences of the TCHR-CLASS set, the ERASE ALL statement erases these member occurrences. This means, in effect, that when the teacher leaves the department, his classes are dropped.



Connecting Records to a Set

To connect a record to a set or to reconnect a record that has been disconnected from a set, perform the following steps:

1. Ready all affected areas in one of the update usage modes (for more information, see Area Usage Modes).

   Areas should be readied whether they are affected explicitly or implicitly (for example, as owner of a set whose members are being connected).

2. Establish the following currencies:

   - The specified record must be current of its record type.
   - The occurrence of the set into which the specified record will be connected must be current of set. If set order is NEXT or PRIOR, current of set also determines the position at which the specified record will be connected within the set.

3. Issue the CONNECT command; CONNECT establishes the specified record occurrence as a member of a set occurrence.
4. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

The specified record must previously have been either stored (manual membership) or disconnected (optional membership).

'Native VSAM users'. The CONNECT statement is not valid since all sets in native VSAM files must be defined as mandatory automatic.

Disconnecting Records from a Set

To cancel the membership of a record occurrence in a set occurrence defined with the optional set membership option, perform the following steps:

1. Ready all affected areas in one of the update usage modes (for more information, see Area Usage Modes).

   Areas should be readied whether they are affected explicitly or implicitly (for example, as owner of a set whose members are being disconnected).
2. Establish the following currencies:
   - The specified record must be current of its record type.
   - The specified record must currently participate as a member in an occurrence of the named set.
3. Issue the DISCONNECT statement.
4. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

Accessing a Disconnected Record

Following successful execution of the DISCONNECT statement, you cannot access the record through the set for which membership was canceled. You can still access the record in the following ways:

- Through an area sweep
- By using its db-key
- Through any other sets in which it still participates
- If it has a location mode of CALC, by using its CALC key

Currencies after a DISCONNECT

Although a successfully executed DISCONNECT statement nullifies currency in the specified set, the DBMS maintains next, prior (if specified), and owner currencies so you can still issue the OBTAIN NEXT, PRIOR, or OWNER WITHIN SET statements.

'Native VSAM users'. The DISCONNECT statement is not valid because all sets in native VSAM files must be defined as mandatory automatic.

Example of Disconnecting and Connecting Records

The program excerpt below disconnects and subsequently reconnects EMPLOYEE records in the DEPT-EMPLOYEE set.

Employees have been transferred to another department. The program ensures that both the new and the old departments exist before disconnecting the EMPLOYEE record from the old DEPT-EMPLOYEE set and connecting it to the new DEPT-EMPLOYEE set.

```
 DATA DIVISION.
 FILE SECTION.
 FD  DEPT-TRANSFER-FILE.
 01  TRANS-EMP-REC-IN.
     02  NEW-DEPT-ID-IN        PIC 9(4).
     02  OLD-DEPT-ID-IN        PIC 9(4).
     02  EMP-ID-IN             PIC 9(4).
 WORKING-STORAGE SECTION.
 01  SWITCHES.
     05 EOF-SW                 PIC X    VALUE 'N'.
       88 END-OF-FILE                   VALUE 'Y'.
 01  CONNECT-DBKEY             PIC S9(8) COMP SYNC.
 PROCEDURE DIVISION.
        .
     READ DEPT-TRANSFER-FILE
        AT END MOVE 'Y' TO EOF-SW.
     PERFORM A300-DISCONNECT-EMP THRU A300-EXIT
                UNTIL END-OF-FILE.
     FINISH.
     GOBACK.
 A300-DISCONNECT-EMP.
     MOVE NEW-DEPT-ID-IN TO DEPT-ID-0410.
     FIND CALC DEPARTMENT.
*** IF ERROR-STATUS = 0326, NEW DEPT ID IS INVALID ***
     IF DB-REC-NOT-FOUND
       DISPLAY
       'NEW DEPARTMENT ' NEW-DEPT-ID-IN ' NOT FOUND'
```

```
        'FOR EMPLOYEE ID ' EMP-ID-IN
        GO TO A300-GET-NEXT
     ELSE IF DB-STATUS-OK
        NEXT SENTENCE
     ELSE
        PERFORM IDMS-STATUS.
*** SAVE NEW DEPT DB-KEY TO REOBTAIN RECORD LATER ***
     MOVE DBKEY TO CONNECT-DBKEY.
     PERFORM IDMS-STATUS.

     MOVE OLD-DEPT-ID-IN TO DEPT-ID-0410.
     FIND CALC DEPARTMENT.
*** IF ERROR-STATUS = 0326, OLD DEPT ID IS INVALID ***
     IF DB-REC-NOT-FOUND
         DISPLAY
        'OLD DEPARTMENT ' OLD-DEPT-ID-IN ' NOT FOUND'
        'FOR EMPLOYEE ID ' EMP-ID-IN '
         GO TO A300-GET-NEXT
     ELSE IF DB-STATUS-OK
        NEXT SENTENCE
     ELSE
        PERFORM IDMS-STATUS.
     MOVE EMP-ID-IN TO EMP-ID-0415.
     OBTAIN CALC EMPLOYEE.

*** IF ERROR-STATUS = 0326, EMP ID IS INVALID ***
     IF DB-REC-NOT-FOUND
         DISPLAY
        'EMPLOYEE ' EMP-ID-IN ' NOT FOUND'
        'FOR OLD DEPARTMENT ' OLD-DEPT-ID-IN
        '*** NEW DEPARTMENT ' NEW-DEPT-ID-IN
        GO TO A300-GET-NEXT
     ELSE IF DB-STATUS-OK
        NEXT SENTENCE
     ELSE
        PERFORM IDMS-STATUS.
*** CHECK IF EMPLOYEE IS A MEMBER IN DEPT-EMPLOYEE SET ***
     IF NOT DEPT-EMPLOYEE MEMBER
         DISPLAY
        'EMPLOYEE ' EMP-ID-IN
        'NOT CONNECTED TO DEPARTMENT ' OLD-DEPT-ID-IN
         GO TO A300-GET-NEXT.
     DISCONNECT EMPLOYEE FROM DEPT-EMPLOYEE.
     PERFORM IDMS-STATUS.
*** REACCESS NEW DEPARTMENT USING ITS DB-KEY ***
     FIND DEPARTMENT DB-KEY IS CONNECT-DBKEY.
     PERFORM IDMS-STATUS.
     CONNECT EMPLOYEE TO DEPT-EMPLOYEE.
     PERFORM IDMS-STATUS.
```

Accessing Bill-of-Materials Structures

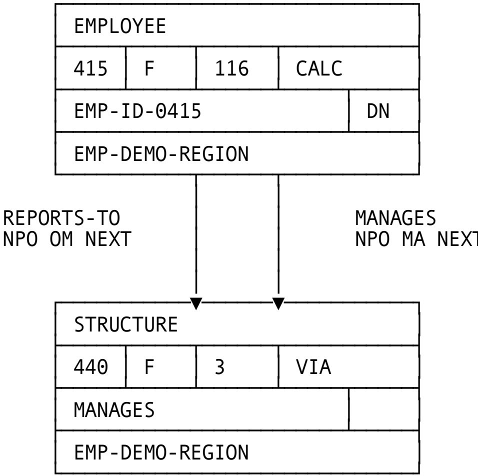
A bill-of-materials structure is a relationship between record occurrences *of the same type*. This structure is derived from the manufacturing environment where it is used to demonstrate relationships between parts: a part can be a component of another part and a part can contain other parts as its components.

This structure is typically represented as a many-to-many relationship (that is, by using two sets and a junction record).

Example of a Bill-of-Materials Structure

In the EMPLOYEE database, a bill-of-materials structure signifies relationships between managers and subordinates: an employee can *manage* other employees through the MANAGES set, and can also be *managed by* other employees through the REPORTS-TO set.

The figure below shows this bill-of-materials structure. The STRUCTURE record serves as the junction record between employees and their managers. Note that one set is defined with the automatic set membership option and the other is defined with the manual set membership option.



Storing a Bill-of-Materials Structure

To store a bill-of-materials structure in the database, perform the following steps:

1. Establish currency at the owner record in the automatic set:

   ```
   MOVE 15 TO EMP-ID-0415.
   OBTAIN CALC EMPLOYEE.
   ```

2. Initialize and store the junction record:

   ```
   PERFORM A100-INITIALIZE-STRUCTURE.
   STORE STRUCTURE.
   ```

   The DBMS automatically connects the STRUCTURE record to the automatic set (MANAGES); EMPLOYEE 15 is now defined as the manager in the bill-of-materials structure.

3. Set run-unit currency at the owner record in the manual set:

   ```
   MOVE 467 TO EMP-ID-0415.
   OBTAIN CALC EMPLOYEE.
   ```

4. Connect the junction record to the manual set:

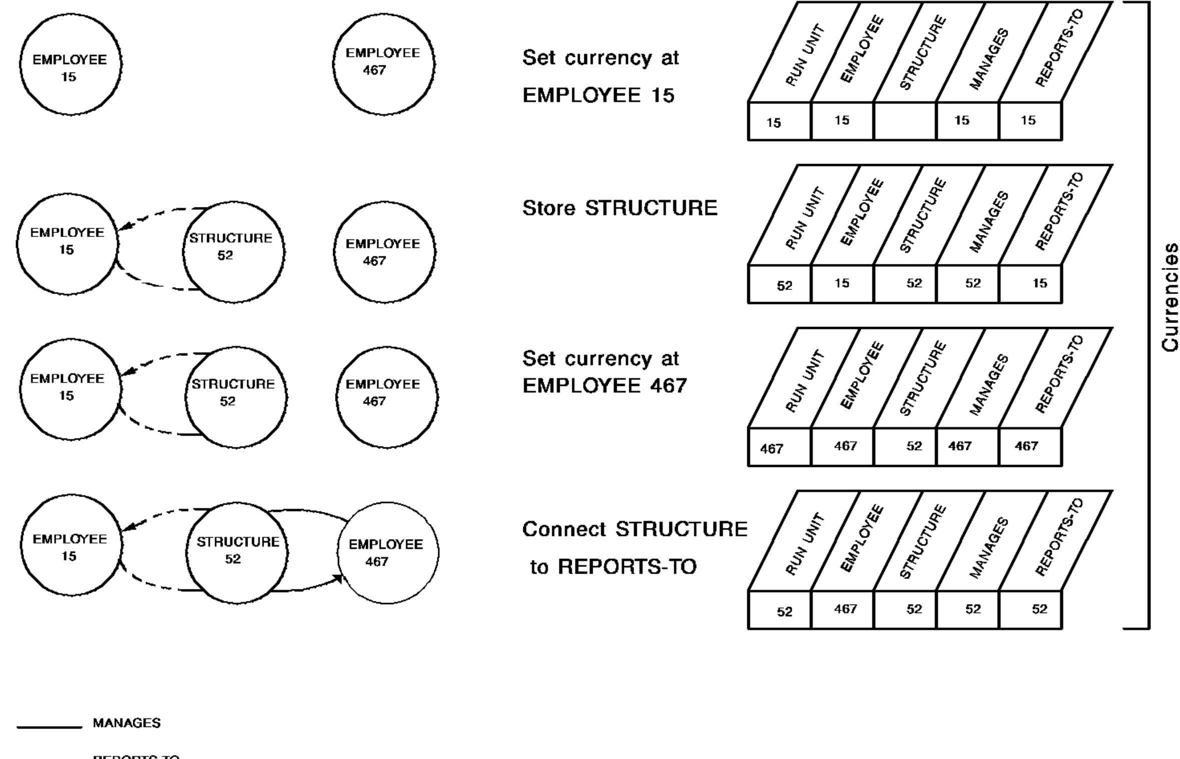
   ```
   CONNECT STRUCTURE TO REPORTS-TO.
   ```

   The bill-of-materials structure is now complete; EMPLOYEE 467 reports to EMPLOYEE 15.

Example of Storing a Bill-of-Materials Structure

The figure below shows the steps and currencies involved in storing an occurrence of a bill-of-materials structure.

To define EMPLOYEE 15 as the manager of EMPLOYEE 467, store and connect a STRUCTURE record as a member of the two EMPLOYEE records: EMPLOYEE 15 in the MANAGES set and EMPLOYEE 467 in the REPORTS-TO set.



Retrieving a Bill-of-Materials Structure

A bill-of-materials structure can contain a variable number of levels. Tracing all records under a given record (for example, finding a manager and all subordinates, and all of their subordinates, and so on) is called an **explosion** of the structure for that record. Tracing all records above a given record (for example, finding an employee and manager, and the manager's manager, and so on) is called an **implosion** of the structure for that record.

To perform a multilevel explosion or implosion, you must maintain a stack of db-keys in order to reestablish the appropriate currencies.

Steps to Retrieve One Bill-of-Materials Level

To retrieve a manager and one level of employees, perform the following steps:

1. Retrieve the manager's EMPLOYEE record:

```
MOVE 15 TO EMP-ID-0415.
OBTAIN CALC EMPLOYEE.
```

2. Retrieve the first STRUCTURE record in the MANAGES set:

```
FIND NEXT STRUCTURE WITHIN MANAGES.
```

3. Because REPORTS-TO is defined as OM, you must test for set membership:

```
IF NOT REPORTS-TO MEMBER
  GO TO A100-EXIT.
```

4. Retrieve the owner EMPLOYEE record in the REPORTS-TO set:

```
OBTAIN OWNER WITHIN REPORTS-TO.
```

5. FIND the current STRUCTURE record to reestablish the original currency within the MANAGES set:

```
FIND CURRENT STRUCTURE.
```

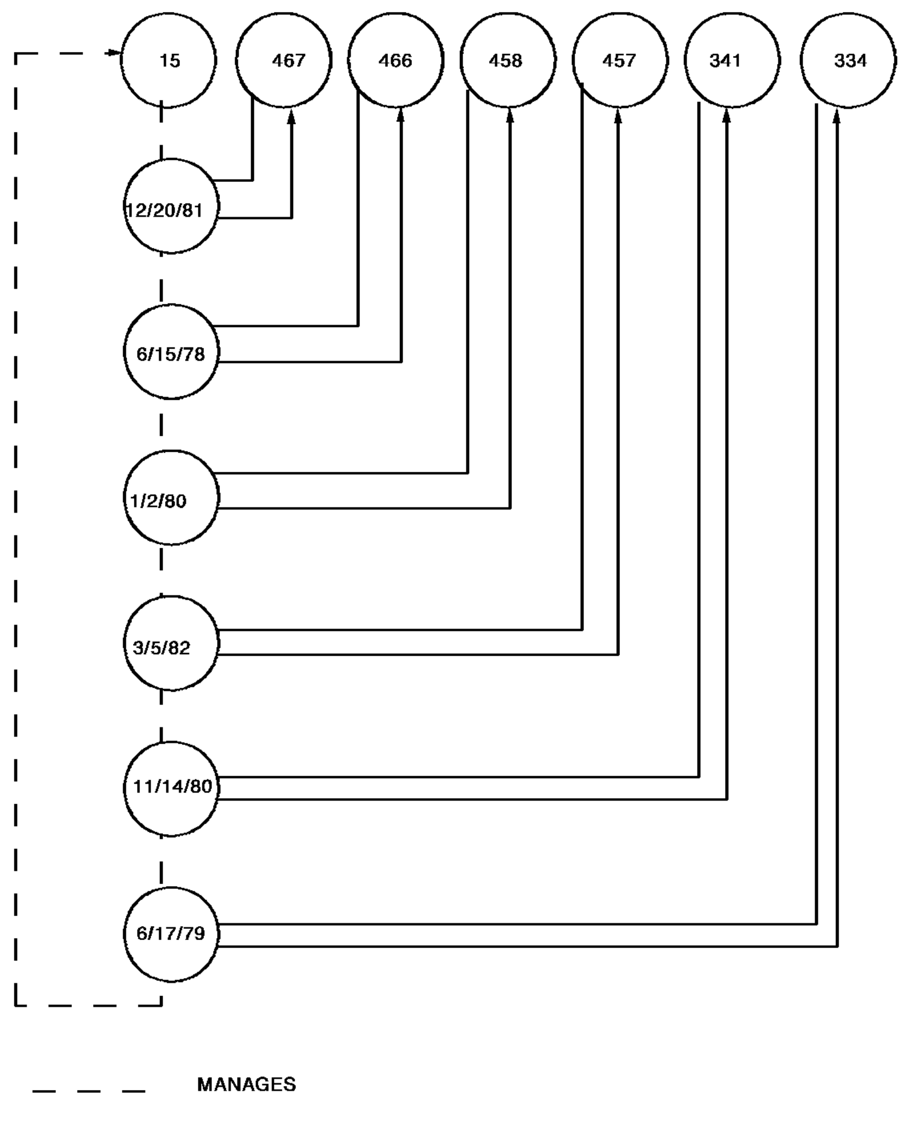
6. Retrieve the next STRUCTURE record in the MANAGES set:

   FIND NEXT STRUCTURE WITHIN MANAGES.

   Perform steps 3 through 6 iteratively until step 6 returns a status of 0307 (DB-END-OF-SET).

Example of One Bill-of-Materials Level

The figure below shows the relationship between manager and employees by showing all the employees managed by employee 15.



Steps to Retrieve Additional Levels

To retrieve an EMPLOYEE record, its manager's EMPLOYEE record, its manager's manager, and so on, perform the following steps:

1. Retrieve the specified EMPLOYEE record:

   ```
   MOVE 91 TO EMP-ID-0415.
   OBTAIN CALC EMPLOYEE.
   ```

2. Retrieve the STRUCTURE record in the REPORTS-TO set:

   ```
   FIND NEXT STRUCTURE WITHIN REPORTS-TO.
   IF ERROR-STATUS = '0307'
     GO TO A100-EXIT.
   ```

3. Optionally, test the junction record for predetermined criteria (for example, if you only want managers for a specific project). Testing for selection criteria in the junction record can prevent looping if there are any circular structures defined.

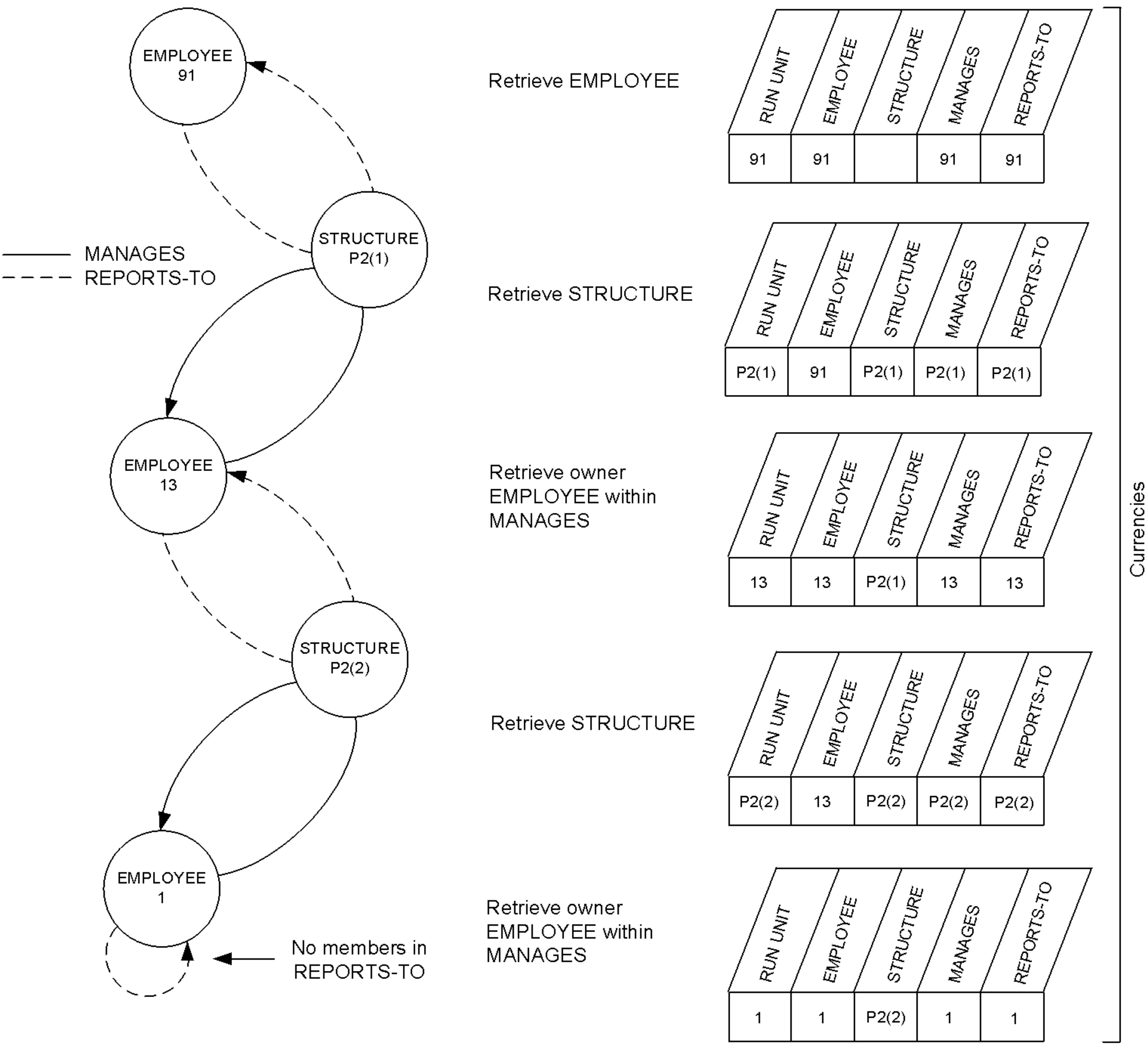
4. Retrieve the owner EMPLOYEE record in the MANAGES set:

   ```
   OBTAIN OWNER EMPLOYEE WITHIN MANAGES.
   ```

   Perform steps 2 through 4 iteratively until step 2 returns a status of 0307, indicating that the REPORTS-TO set is empty.

Example of Retrieving Additional Levels

The figure below shows the relationship between an employee and all managers on the P2 project by showing the hierarchy of managers above EMPLOYEE 91 on the project.



Locking Records

You can explicitly place a shared or exclusive lock on a record that is current of run unit, record, set, or area. You should place explicit locks on records for the following reasons:

- To ensure later access to a specified record occurrence by preventing other run units from modifying or deleting it in the interim
- To ensure exclusive access to a specified record occurrence (by preventing other run units from accessing the occurrence in any way)

To ensure later access, place a **share** lock on the record. To ensure exclusive access, place an **exclusive** lock on the record.

Steps in Locking Records

To place an explicit lock on a record, perform the following steps:

1. Establish the appropriate run unit, record, set, or area currency.
2. Issue the KEEP statement.
3. Perform the IDMS-STATUS routine if the DBMS returns a nonzero value.

Alternatively, you can use the KEEP option of the FIND/OBTAIN statement to place locks on records as they are retrieved.

How Long Explicit Locks are Held

The DBMS maintains explicit record locks until the next COMMIT, FINISH, or ROLLBACK statement.

For more information on shared and exclusive locks, see Record Locks.

Example of Using KEEP to Lock a Record

The program excerpt below shows the use of the KEEP statement in a program that connects and disconnects records.

The program places an explicit shared lock on the new DEPARTMENT record occurrence to prevent other run units from modifying it and to guarantee access later in the program.

```
 A300-DISCONNECT-EMP.

    MOVE NEW-DEPT-ID-IN TO DEPT-ID-0410.
    FIND CALC DEPARTMENT.
*** IF ERROR-STATUS = 0326, NEW DEPT ID IS INVALID ***
    IF DB-REC-NOT-FOUND
       DISPLAY
      'NEW DEPARTMENT ' NEW-DEPT-ID-IN ' NOT FOUND'
      'FOR EMPLOYEE ID ' EMP-ID-IN
      GO TO A300-GET-NEXT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
*** LOCK NEW DEPARTMENT TO ENSURE THAT ***
*** OTHER RUN UNITS DO NOT MODIFY IT   ***
   KEEP CURRENT DEPARTMENT.
*** SAVE NEW DEPT DB-KEY TO REOBTAIN RECORD LATER ***
    MOVE DBKEY TO CONNECT-DBKEY.
    PERFORM IDMS-STATUS.

    MOVE OLD-DEPT-ID-IN TO DEPT-ID-0410.
    FIND CALC DEPARTMENT.
*** IF ERROR-STATUS = 0326, OLD DEPT ID IS INVALID ***
    IF DB-REC-NOT-FOUND
       DISPLAY
      'OLD DEPARTMENT ' OLD-DEPT-ID-IN ' NOT FOUND'
      'FOR EMPLOYEE ID ' EMP-ID-IN '
       GO TO A300-GET-NEXT
    ELSE IF DB-STATUS-OK
      NEXT SENTENCE
    ELSE
      PERFORM IDMS-STATUS.
    MOVE EMP-ID-IN TO EMP-ID-0415.
    OBTAIN CALC EMPLOYEE.
*** IF ERROR-STATUS = 0326, EMP ID IS INVALID ***
    IF DB-REC-NOT-FOUND
       DISPLAY
      'EMPLOYEE ' EMP-ID-IN ' NOT FOUND'
      'FOR OLD DEPARTMENT ' OLD-DEPT-ID-IN
      '*** NEW DEPARTMENT ' NEW-DEPT-ID-IN
      GO TO A300-GET-NEXT
    ELSE
      PERFORM IDMS-STATUS.
```

Implicit Record Locks

If your run unit executes through central version and readies an area in shared update, the DBMS acquires implicit records locks in order to control concurrent access to data. It may acquire record locks for other ready modes, depending on the mode and the setting of lock-related system generation parameters.

For a complete discussion of when record locks are acquired, see the *CA IDMS Database Administration Guide*.

Implicit Shared Locks

Shared locks are placed on record occurrences that are current of record type, set, area, and run unit and are held until a record occurrence is no longer current.

Implicit Exclusive Locks

Exclusive locks are placed on record occurrences that are updated by your application. As you issue DML verbs that update the database, the DBMS acquires these exclusive locks and holds them until you commit or rollback your changes. They prevent other database sessions that are not sharing your run unit's transaction from accessing the locked records. To increase concurrency and avoid deadlocks, you should commit your changes frequently.

How Long Implicit Locks are Held

The DBMS maintains implicit exclusive locks until the transaction ends, which typically occurs when the next COMMIT, FINISH, or ROLLBACK statement is executed. Implicit share locks are maintained until the record is no longer current of run unit, set, area, or record. COMMIT ALL, FINISH, and ROLLBACK statements nullify all currencies and therefore cause all implicit shared locks to be released. COMMIT (without the ALL option) has no affect on currencies nor implicit share locks.

The following table lists the exclusive record locks set implicitly by each DML verb.

DML Verb	Records Locked
CONNECT/DISCONNECT	■ Connected/disconnected record ■ Next and prior of set being connected/disconnected ■ If set mode is INDEX, typically, one SR8 record

DML Verb	Records Locked
ERASE	■ Erased record ■ The record with the highest possible line number on thepage on which record resided ■ Next and prior of sets in which record is a member ■ If CALC, next and prior of CALC set ■ If owner of nonempty set occurrence, all member records ■ If variable-length record with stored fragments, all record fragments and the record with the highest possible line number on all pages on which fragments resided
FIND (as it applies to indexed records)	■ Typically, one SR8 record (released on returnfrom the FIND)
FIND (as it applies to logically deleted records encountered while walking a set in any update usage mode)	■ If disconnected from set being processed, DISCONNECT locks apply ■ If disconnected from last set occurrence, record is erased; ERASElocks apply
GET (as it applies to relocating variable-length records when fragments can be relocated while processing in any update usage mode)	■ The record with the highest possible line number on the record's homepage (the page on which the root of the record resides) ■ The record with the highest possible line number on all pages on which fragments formerly resided ■ All former fragments

DML Verb	Records Locked
MODIFY	■ Modified record ■ The record with the highest possible line number on thepage on which record resides (if record size changes) ■ If member of sorted set and if changing sort-key value, old and new next and prior in set occurrence ■ If CALC and modifying CALC key, old and new next and prior in CALC set ■ If modifying the symbolic key of an indexed set, two SR8 records ■ If variable-length record with stored fragments, all record fragments and the record with the highest possible line number on all pages on which fragments reside (if the fragment size changes or the fragment is added or deleted)
STORE	■ Stored record ■ The record with the highest possible line number on thepage on which record is being stored ■ Next and prior of all sets in which record is an automatic member ■ If CALC, next and prior of CALC set ■ If variable-length record with stored fragments, all record fragments and the record with the highest possible line number on all pages on which fragments are being stored ■ For indexed records, typically, one SR8 for each index in which the record participates 1 - Locks are set for next records only if the set has prior pointers. 2 - The record with the highest possible line number on a page is used to control updating of the available space on the page.

Collecting Database Statistics

You can collect database run-time statistics with the ACCEPT DATABASE-STATISTICS statement. You can issue this statement any number of times during a run unit, It returns a copy of the IDMS statistics block to a specified location in program variable storage.

Although the ACCEPT DATABASE-STATISTICS statement can be issued any number of times during a run unit, IDMS statistics are cumulative; resetting of IDMS statistics occurs only upon issuing a FINISH or ROLLBACK statement.

Uses of Database Statistics

Possible uses of database statistics include:

- Determining whether a variable-length record was stored on one page or fragments were placed in an overflow area
- Obtaining the date and time at the start and end of a run unit
- Keeping track of the number of update locks being held and issuing regular commits based on that statistic

For more information on collecting database run-time statistics and individual IDMS statistics block fields, see the language-specific *CA IDMS DML Reference Guide*.

'Performance Monitor users'. You can use CA IDMS Performance Monitor to collect statistics about program execution. For more information, see the *CA IDMS Performance Monitor User Guide*.

Chapter 7: Run Units, Locks, and Database Transactions

A run unit is a database session through which a CA IDMS database can be accessed using navigational DML requests. A run unit is associated with a database transaction that represents the recoverable work done by its associated sessions. Run units can share their database transactions with other database sessions.

Record locks and area in-use locks ensure data integrity by preventing concurrent update of database records by other applications. Additionally, your program can specify area usage modes to ensure a particular level of control over database areas to be accessed. You should be familiar with these locks and usage modes, their uses, and their effect on the run-time system, particularly when running under the CA IDMS central version.

Navigational programs that maintain efficient run units help to maximize the resources of a run-time system. Well-managed record locks, area locks, and database transactions are major considerations in maintaining efficient run units.

This section contains the following topics:

Run Units (see page 173)
Sharing Run Units Between Programs (see page 174)
Record Locks (see page 175)
Area Locks (see page 180)
Area Usage Modes (see page 181)
Database Transactions (see page 183)
Sharing Transactions Among Sessions (see page 187)

Run Units

A run unit is a database session that begins with the BIND RUN-UNIT statement and (if successful) ends with the FINISH statement. A program can serially bind and finish any number of run units, but typically binds only one.

Note: If your program binds run units serially, you must reinitialize the ERROR-STATUS field in the IDMS communications block to the value 1400 before reissuing the BIND RUN-UNIT and READY statements.

An application consisting of multiple programs can bind concurrent run units, since each program can bind its own run unit.

When a run unit starts, it is associated with a database transaction representing the recoverable work done by the run unit. A database transaction can be shared by multiple database sessions -- both run units and SQL sessions. Sharing a transaction eliminates deadlocks between the sharing sessions, but impacts the commit operation and introduces the potential for interference between the sharing sessions.

For more information on how to share transactions and the considerations in doing so, see Sharing Transactions Among Sessions.

Sharing Run Units Between Programs

Multiple programs can share a single run unit by passing the IDMS communications block from one program to another program. Typically, the program that binds the run unit will allocate the IDMS communications block and then pass it to the other programs for their use.

COBOL Programmers

Even if the IDMS communications block is shared between programs, it is advisable to allow each program to generate its own list of records, sets, and area names so that if these change due to changes in the subschema, there is no need to recompile all programs sharing the run unit.

Example

The example below shows how to code the PROTOCOL section for two programs that are sharing a run unit. These examples are for a COBOL program.

```
Program-A
   IDENTIFICATION DIVISION
   PROGRAM-ID. MAINLINE.
   ENVIRONMENT DIVISION.
   IDMS-CONTROL SECTION.
   PROTOCOL. MODE IS BATCH-AUTOSTATUS DEBUG
        IDMS-RECORDS WITHIN WORKING-STORAGE SECTION.
   DATA DIVISION.
   SCHEMA SECTION.
   DB EMPSS01 WITHIN EMPSCHM VERSION 100.
   WORKING-STORAGE SECTION.
```

```
    PROCEDURE DIVISION.
    COPY IDMS SUBSCHEMA-BINDS
    READY USAGE-MODE IS UPDATE.
 * read some database records and do updates as needed. Then pass control
to a subprogram keeping the run unit open so that the subprogram can also
retrieve and update records as needed, and then return control to the
main program.
    CALL 'SUBPROG' USING SUBSCHEMA-CTRL.

Program-B
    IDENTIFCATION DIVISION
    PROGRAM-ID. SUBPROG.
    ENVIRONMENT DIVISION.
    IDMS-CONTROL SECTION.
    PROTOCOL. MODE IS BATCH-AUTOSTATUS DEBUG
          IDMS-RECORDS MANUAL.
    DATA DIVISION.
    SCHEMA SECTION.
    DB EMPSS01 WITHIN EMPSCHM VERSION 100.
    WORKING-STORAGE SECTION.
     COPY IDMS SUBSCHEMA-NAMES.
     COPY IDMS SUBSCHEMA-RECORDS.
    LINKAGE SECTION.
     COPY IDMS SUBSCHEMA-CTRL.

    PROCEDURE DIVISION USING SUBSCHEMA-STRL.
 * read some database records and do updates as needed. Then return
control to the main program keeping the run unit open.
       GOBACK.
```

Record Locks

In general, record locks prevent concurrent retrieval and update by separate run units operating under the same central version. This statement does not apply when:

- Run units operate in local mode-concurrent update of record occurrences is prevented by physical area locks.
- RETRIEVAL NOLOCK has been specified in system generation - the system does not maintain locks for retrieval run units.
- Run units share a transaction with other database sessions; all sharing sessions can concurrently retrieve and update the same data.

Exclusive Lock

An exclusive lock indicates that no other run unit can access the designated record occurrence in any way. Only one run unit at a time can place an exclusive lock on a record occurrence. A run unit can place an exclusive lock on a record occurrence only if that occurrence has not been assigned any locks (shared or exclusive) by another run unit. A run unit that tries to place an exclusive lock on an occurrence that already has been locked must wait until all other locks on the occurrence are released.

Shared Lock

A shared lock indicates that other run units can retrieve the designated record occurrence but cannot update it. Any number of run units can place a shared lock on a record occurrence. A run unit that tries to place a shared lock on an occurrence for which an exclusive lock is already held must wait until the exclusive lock is released.

Notify Lock

A notify lock is used in the online environment to monitor database access to a specified record occurrence.

For more information on notify locks, see Maintaining Data Integrity in the Online Environment.

Implicit and Explicit Locks

Record locks can be set implicitly by the central version or you can set them explicitly by coding the DML KEEP function in the program.

Implicit Locks

Implicit locks are maintained automatically by the central version for every run unit accessing the database in shared update usage mode. The DBA can also specify that implicit locks be maintained for run units accessing the database in shared retrieval or protected update usage mode.

For further details about usage modes, see Area Usage Modes.

Types of Implicit Lock

Implicit locks can be shared or exclusive, as follows:

- The central version places **implicit shared locks** on the record occurrences that are current of run unit, record, set, and area. These locks remain in effect until the record occurrences are no longer current, thereby preventing concurrently executing run units from updating the same record.

- The central version places an **implicit exclusive lock** on every record occurrence that is modified by a DML statement (STORE, MODIFY, ERASE, CONNECT, or DISCONNECT). Additionally, the central version sets implicit exclusive locks for:
  - The next and prior record occurrences for all sets in which the record's participation has changed
  - Each database page on which the amount of space has been altered as the result of a STORE, MODIFY, or ERASE statement
  - The central version maintains implicit exclusive locks for the duration of the database transaction to prevent concurrently executing run units that are maintaining locks from accessing modified records that might have to be rolled back because of an error later in the program.

For more information on the implicit locks acquired by the DBMS, see Implicit Record Locks.

Explicit Locks

Explicit locks, which you set in your program, maintain record locks that would otherwise be released after a change in currency. The KEEP statement and the KEEP clause of the FIND/OBTAIN statement are used to set explicit shared and exclusive locks.

For more information about setting explicit locks, see Locking Records.

Managing Record Locks

Accumulating a large number of implicit or explicit record locks during a database transaction hinders system performance. You can maintain efficient database transactions by regularly issuing the DML COMMIT statement (described later in this chapter).

Additionally, certain conditions that result from the use of record locks ca CAuse abnormal termination of run units executing under the central version:

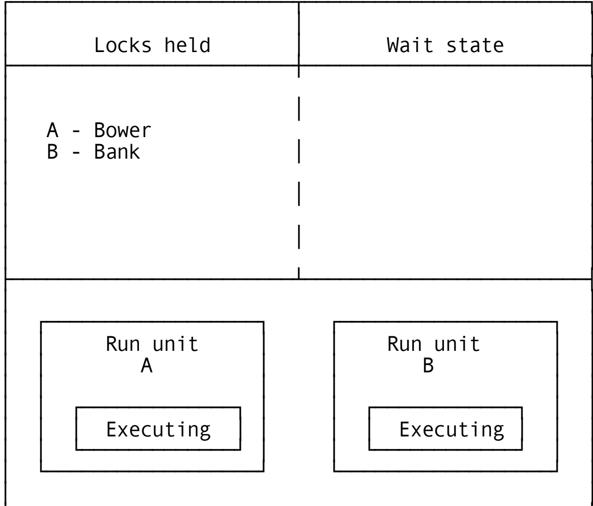
- **Exceeded wait time**-A run unit waiting to set a lock on a record that is currently locked by another transaction abends if it exceeds the internal wait interval specified at central version generation. When this happens, the central version rolls back the database transaction and returns a value of *nn*69 to the ERROR-STATUS field in the IDMS communications block.
- **Deadlock**-If two or more run units would cause a deadlock were they all permitted to wait, one of them is aborted to resolve the deadlock.

  When a run unit is terminated because of a potential deadlock, the central version rolls back the database transaction, returns a value of *nn*29 to the ERROR-STATUS field in the IDMS communications block, and releases all locks held by the aborted run unit.

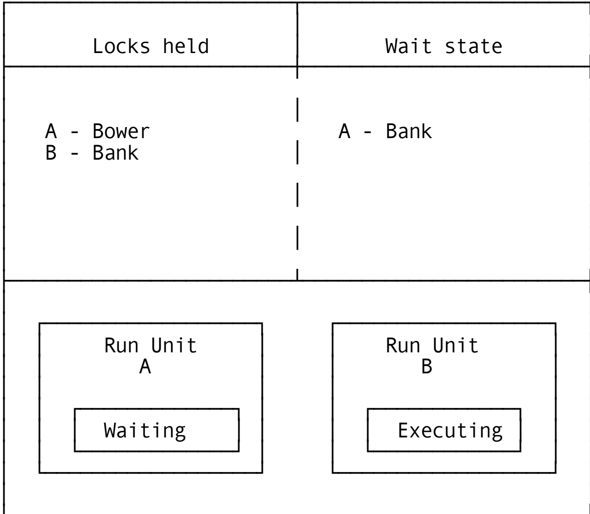
Deadlock Example

The following sequence of figures shows a typical deadlock situation:

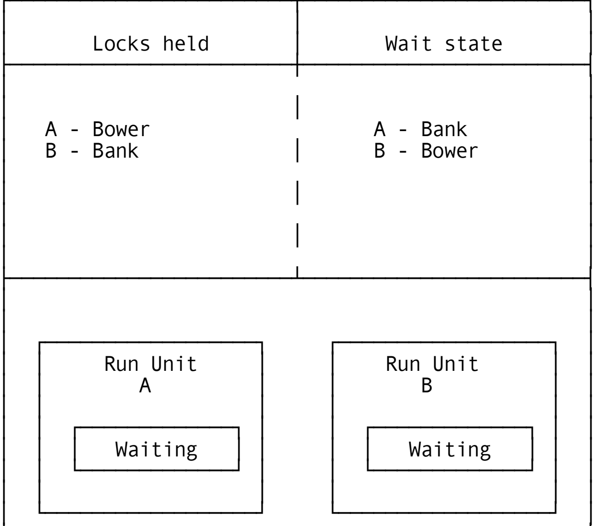
- Run unit A and run unit B have placed shared locks (implicitly or explicitly) on the Bower EMPLOYEE record occurrence and the Bank EMPLOYEE record occurrence respectively; neither of these locks can be released until processing is complete.



- Run unit A tries to place an exclusive lock on the Bank record (locked by run unit B); it is placed in a wait state.



- Run unit B then attempts to place an exclusive lock on the Bower record (locked by run unit A) and deadlock results.



- CA IDMS automatically resolves the deadlock by forcing one of the run units to be rolled back. In general, the younger of the two run units will become the victim unless the issuing task has a higher priority than that of the other issuing task involved in the deadlock.

For more information about deadlock detection and processing, see the *CA IDMS Database Administration Guide*.

Area Locks

Area in-use locks, also referred to as physical area locks, are examined whenever an area is opened in an update usage mode. These locks prevent run units originating in multiple regions or partitions (multiple local mode run units, multiple central versions, or a combination of both) from concurrently updating an area. Area in-use locks also prevent any access to an area that requires recovery of incomplete run units due to a local mode or central version abend.

Local Mode

In local mode the area lock is checked as each area is readied in an update usage mode. If the lock is already set, a value of 0966 is returned to the ERROR-STATUS field in the IDMS communications block and access to the area is disallowed. If the lock is not set, the local mode run unit causes the lock to be set. If the run unit terminates abnormally (that is, without issuing a FINISH statement), the lock remains set. Further update access by subsequent local mode or central version run units is prevented until the area is recovered manually (by restoring files or using a CA-supplied recovery utility).

Central Version

Each area accessible by a central version has an associated access mode. Access modes determine the availability of each area to run units running under the central version, to other central versions, and to programs running in local mode. The access modes are described below:

- **UPDATE (ONLINE)** indicates that areas are available for update to run units running under the central version. Run units running in local mode or other central versions cannot ready the area in any update usage mode.
- **[TRANSIENT] RETRIEVAL** indicates that areas are available for retrieval to run units running under the central version. Run units running in local mode or under other central versions can ready the area in any usage mode.
- **OFFLINE** indicates that areas are not available for update or retrieval to run units running under the central version. Run units running in local mode or under other central versions can ready the area in any usage mode.

Note: The UPDATE, RETRIEVAL, and OFFLINE central version access modes are operator concerns; they are presented here as background information only. You do not specify these modes in your program.

System Startup

When the central version starts up, it checks the in use locks in all areas available for update. If any lock is found to be set, a warning message is displayed at the operator's console and further access to that area is disallowed. The central version proceeds without the use of that area; any run unit attempting to ready that area receives a value of 0966 returned to the ERROR-STATUS field in the IDMS communications block. If the lock is removed after startup, the DBA must change the area status from OFFLINE to ONLINE or RETRIEVAL to make the area available to the central version.

Note: In-use area locks are not set for individual run units running under the central version; run unit conflicts are avoided by internal means.

Area Usage Modes

Run units ready an individual area in a particular usage mode in order to define the scope of operations that can be performed against that area. The area usage modes, which are specified by the DML **READY** statement, are retrieval and update.

Retrieval

Retrieval specifies that the issuing run unit can perform only retrieval functions such as FIND, OBTAIN, and IF against records in that area. It cannot issue the STORE, MODIFY, ERASE, CONNECT, or DISCONNECT statements.

Update

Update specifies that the issuing run unit can modify as well as retrieve records in that area. That is, it can issue all available DML statements.

Ready Options

You can issue a ready option in conjunction with a usage mode to restrict retrieval or update of records in the specified area by other run units executing concurrently under the same central version. The ready options are:

- **Protected** indicates that other run units cannot ready the specified area in update usage mode and must wait until your run unit terminates. If you ready an area with the protected option while a concurrently executing run unit has readied the area in update mode, you will wait until the other run unit terminates.

- **Exclusive** indicates that other run units cannot ready the specified area in *any* usage mode and must wait until your run unit terminates. If you ready an area with the exclusive option while a concurrently executing run unit has readied the area, you will wait until the other run unit terminates.
- **Shared** indicates that more than one run unit running under the same central version can concurrently access the same area.

  Note: You cannot explicitly code SHARED in the READY statement (that is, READY UPDATE is functionally the same as READY SHARED UPDATE).

  Combinations of Usage Mode and Ready Options

The table below summarizes the effect that various combinations of usage modes and ready options have on concurrently executing run units.

The usage mode in which one run unit readies an area restricts the usage mode in which other run units executing under the same central version can ready that area. The following table shows in which usage modes two concurrent run units can ready an area. Y (yes) signifies that the second run unit can ready the area in the specified usage mode; N (no) signifies that it cannot.

Run unit A \ Run unit B	SHARED UPDATE	SHARED RETRIEVAL	PROTECTED UPDATE	PROTECTED RETRIEVAL	EXCLUSIVE UPDATE	EXCLUSIVE RETRIEVAL
SHARED UPDATE	Y	Y	N	N	N	N
SHARED RETRIEVAL	Y	Y	Y	Y	N	N
PROTECTED UPDATE	N	Y	N	N	N	N
PROTECTED RETRIEVAL	N	Y	N	Y	N	N
EXCLUSIVE UPDATE	N	N	N	N	N	N
EXCLUSIVE RETRIEVAL	N	N	N	N	N	N

Wait State

When a run unit cannot ready an area because a protected or exclusive restriction is already placed on that area by another run unit running under the same central version, it is placed in a wait state until the first run unit is finished.

Automatic Implicit Locking

The central version automatically maintains implicit record locks. These record locks are dependent on DBA specifications and on the area usage mode specified:

- **Shared update**- The central version always maintains record locks for run units executing in shared update usage mode.
- **Shared retrieval**-The central version maintains record locks for run units executing in shared retrieval mode only if specified by the DBA at system generation. If they are not maintained, a run unit with shared retrieval usage mode may yield unpredictable results if it accesses records being modified by a concurrently executing run unit with a shared update or protected update usage mode.
- **Protected update**-The central version maintains exclusive record locks for run units executing in protected update mode only if specified by the DBA at system generation. Shared record locks are never maintained for protected access because no other run unit can concurrently update records in the area.
- **Protected retrieval, exclusive update, exclusive retrieval** -The central version does not maintain implicit record locks for run units executing in these modes since the usage modes themselves prohibit concurrent update.

Default Usage Modes

Your DBA can assign default usage modes for subschema areas. The specified default determines the usage mode in which an area will automatically be readied for programs using that subschema. You do not have to code READY statements in programs that use such a subschema; however, if you issue a READY command for one area in the subschema, you must issue READY commands for all database areas to be accessed unless the FORCE option is specified for the default usage mode. Areas using the default usage mode combined with the FORCE option are automatically readied even if the run-unit already issued READY for other areas.

You can use the SUBAREA parameter of the IDMSRPTS utility to determine if the DBA has specified any default usage modes.

Database Transactions

A database transaction is a unit of recovery that represents work done by one or more database sessions. All access to CA IDMS data from within a run unit is done under the control of a database transaction.

Journaling Changes

Every time your program modifies the database, a before and after image of the affected record occurrence is written to the journal file. These images are used in the event of program or system failure to recover (roll back) all changes made to the database. During a rollback, updates are reversed to the last checkpoint written to the journal file.

Checkpoints

Checkpoints are written at the following times:

- When a database transaction is started or when the first update is made. For run units that don't share their transaction, its transaction starts when a BIND RUN-UNIT is issued.
- When a database transaction is committed. For run units that don't share their transaction, this occurs when a FINISH or COMMIT request is issued.
- When a database transaction is backed out. For run units that don't share their transaction, this occurs when a ROLLBACK request is issued.

A database transaction encompasses the work done between two checkpoints. A database transaction is also a recovery unit since all changes made by a transaction are committed or recovered as a single unit.

Note: If a database transaction is shared by more than one session, itimpacts when checkpoint records are written.

For more information, see Sharing Transactions Among Sessions.

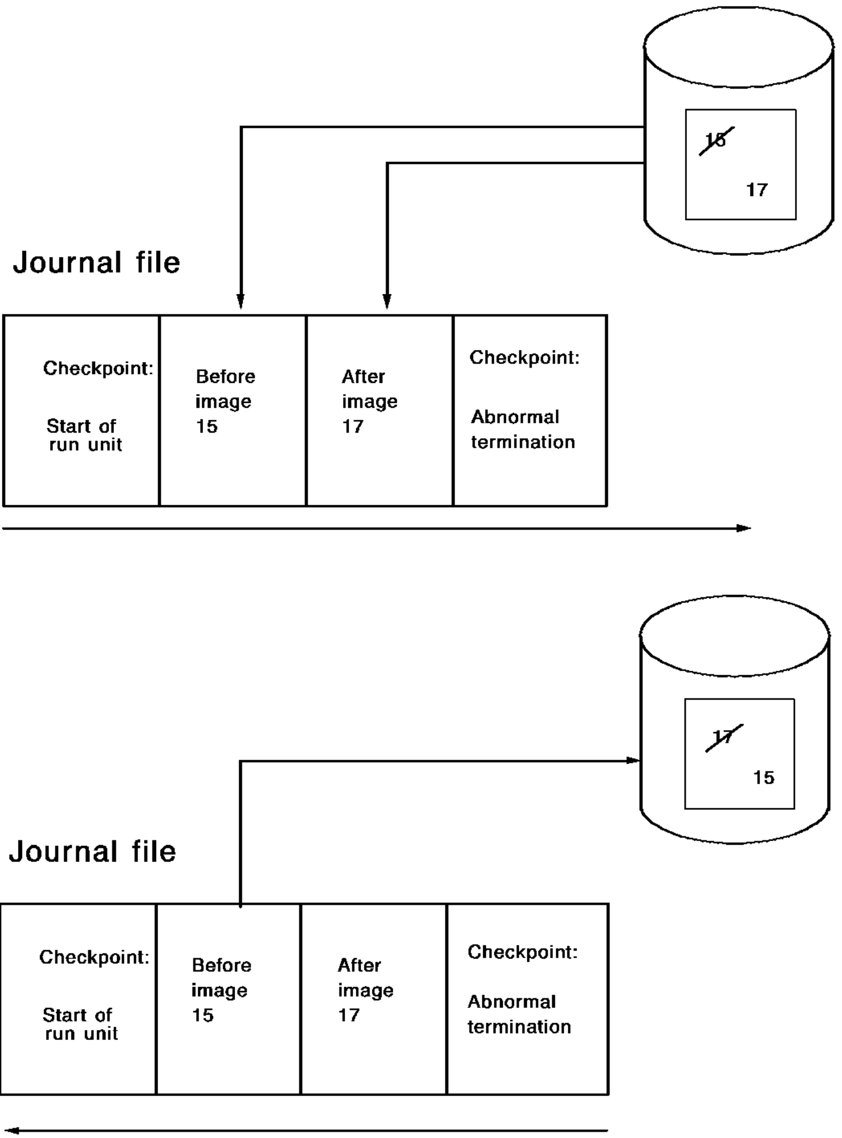
Automatic Recovery Under Central Version

Recovery takes place automatically for programs running under the central version. In order to recover under local mode, you must manually recover the database by using CA IDMS recovery utilities or by restoring the database files from a backup copy.

For more information on recovering the database under local mode, see the *CA IDMS* Database *Administration Guide*.

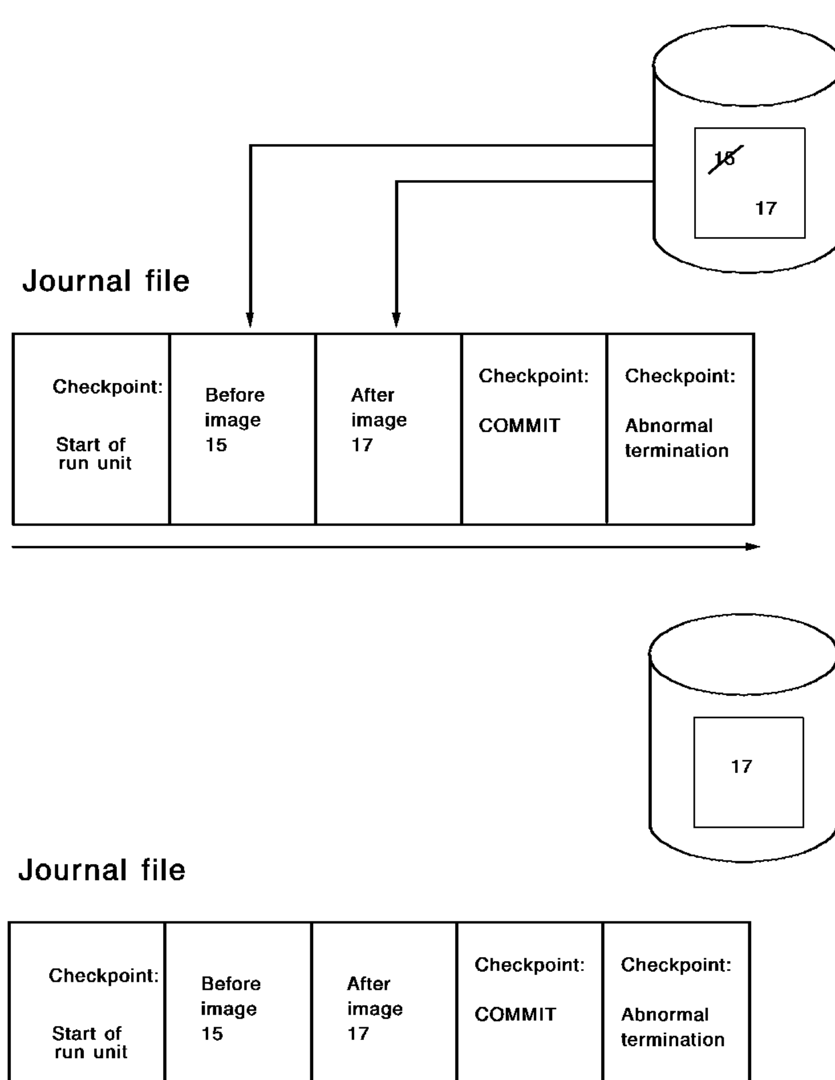
Rolling Back the Database

The figure below shows journaling, checkpoints, and rollback. The BIND RUN-UNIT statement writes the initial checkpoint to the journal file. Before and after images are maintained for every modified record occurrence. In the event of an abend, the central version uses the before images to restore the database back to the last checkpoint. In this figure, there is a one-to-one association between the run unit and its database transaction.



Establishing Checkpoints

The figure below shows the use of the COMMIT statement to establish checkpoints. In the event of an abend, the central version restores the database as far back as the last COMMIT checkpoint.



Committing Changes

If your application performs database updates, you should commit those updates at regular intervals to:

- Release implicit locks held by the database transaction
- Prevent needless rollback of valid database updates

Frequency of COMMIT Statements

Since modified records are implicitly and exclusively locked, timely use of the COMMIT statement is an important programming consideration.

The frequency of issuing COMMITs is a site- and application-specific decision. Some questions to ask when determining the frequency of COMMITs are:

- **Is a logical unit of work complete?**-You should maintain implicit exclusive locks at least until a logical unit of work is complete. For example, if you plan to DISCONNECT and subsequently CONNECT a record, you should not issue the COMMIT until after the CONNECT.
- **What is the application's operating environment?**-If there will be a high volume of concurrent online users, you should try to keep database transactions short by issuing COMMITs more frequently.
- **How many locks will be held for each record modified?**-:ih1.locks Additional implicit exclusive record locks (for example, on the NEXT and PRIOR records) are held when modifying symbolic keys, when erasing occurrences from the database, when connecting or disconnecting records, or when modifying variable-length records.

  For detailed information on the implicit exclusive record locks maintained for each modified record, see Implicit Record Locks.
- **How many locks is too many?**-This is application- and site-specific, although you probably do not want to maintain implicit exclusive locks on more than 70-100 records at any one time.

  Note: You can use the IDMS statistics block (explained in Collecting Database Statistics) to obtain lock-related information at run time.

Sharing Transactions Among Sessions

Sharing a Transaction

A transaction can be shared by multiple database sessions -- both run units and SQL sessions. By sharing a transaction, sessions will not deadlock among themselves even if they access and update the same data.

Enabling Transaction Sharing

A run unit is eligible to share its transaction if transaction sharing is in effect when the BIND RUN UNIT is issued. Whether transaction sharing is in effect depends on whether the run unit is a top-level or subordinate session.

A run unit (or any database session) started by an application program that is not executing as part of a database procedure or an SQL routine, is referred to as a **top-level session**. Transaction sharing is in effect for a top-level session if it is enabled in one of the following ways:

- TRANSACTION_SHARING=ON is specified in the SYSIDMS file for a batch application

  See *CA IDMS Common Facilities Guide* for information about SYSIDMS parameters.

  The IDMSCINT or CICSOPT parameter specifies TXNSHR=ON for CICS applications

  See *CA IDMS System Operations Guide* for information about IDMSCINT and CICSOPT parameters.

- Transaction sharing is enabled for the executing DC/UCF task by means of a SYSGEN or DCMT command.

  See *CA IDMS System Operations Guide* for information about DCMT commands and *CA IDMS System Generation Guide* for information about system generation.

- Transaction sharing is enabled through a call to IDMSIN01 before the BIND RUN UNIT is issued.

  See *CA IDMS Callable Services Guide* for information about calling IDMSIN01.

A run unit (or any database session) started by an application program that is executing as part of a database procedure or an SQL routine is referred to as a **subordinate session**. For subordinate sessions started by database procedures, transaction sharing is in effect if it has been enabled prior to procedure invocation or by a call to IDMSIN01 from within the procedure. For subordinate sessions started by SQL routines, transaction sharing is controlled through the TRANSACTION SHARING parameter of the SQL routine definition unless overridden by a call to IDMSIN01 from within the routine itself.

See *CA IDMS SQL Reference Guide* for information about the TRANSACTION SHARING parameter of the CREATE PROCEDURE, CREATE TABLE PROCEDURE or CREATE FUNCTION statements.

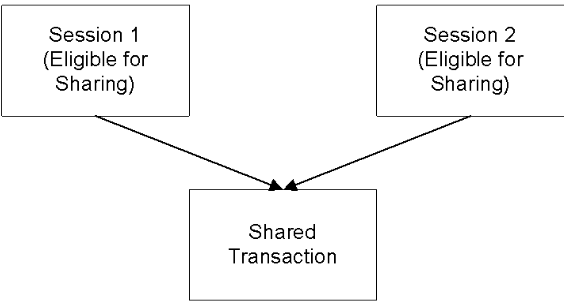
Whether transaction sharing is enabled for a remote run unit is determined by the attribute in effect in the CA IDMS environment in which the BIND RUN UNIT is issued. (A remote run unit is one for which the database being accessed resides on a central version different from where the application is executing.)

System internal run units never share their transactions.

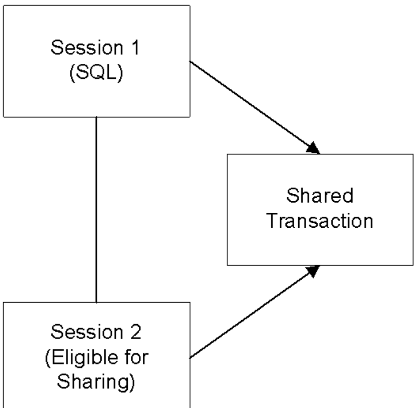
Sharing Transactions

Regardless of how transaction sharing is enabled, if it is in effect at the time a run unit is started, then that run unit is eligible to share its transaction with other database sessions started by the same task or user session. The following rules determine whether a run unit will share a transaction:

- A top-level run unit will share its transaction with another top-level session if they are both eligible for transaction sharing. The other top-level session could be another run unit or an SQL session.



- A subordinate run unit that is eligible for transaction sharing shares its parent session's transaction even if the parent session is not eligible to share its transaction.



Application Programming Considerations

Transaction sharing affects applications in the following ways:

- An update made through a database session may impact other database sessions sharing the same transaction.
- A rollback issued within one database session affects all sessions that share the same transaction.
- A commit issued by a database session whose transaction is shared has no affect on the transaction unless all other sharing sessions have also been committed.

Inter-session Interference

Database sessions that share a transaction can impact each other in ways that would not be possible without transaction sharing since locking would prevent such interactions. For example, a record can be deleted by one database session while it is current of another database session that is sharing the same transaction. This can result in new and possibly unexpected error conditions. If a database session's currency is impacted by an update made through another database session, that currency is invalidated. If a subsequent DML request, such as a MODIFY relies on that invalidated currency, an error is returned:

- For navigational DML, an error status of xx03 is returned to the application.
- For SQL, the application receives an SQLCODE of -4 (statement failure) and an SQLRSN or 1087 (conflicting activity within a shared transaction).

Before enabling transaction sharing for an application, you should ensure that affected programs handle these errors appropriately. For instance, a navigational DML program could re-obtain the record that was the target of a failed MODIFY.

Effect of Rollback Requests

If multiple database sessions share a transaction and one of those sessions issues a rollback request, all changes made within the transaction are immediately rolled out, including those made by other database sessions. Other sessions sharing the transaction must issue their own rollback request before issuing other DML requests. Issuing a non-rollback DML request first will result in an error:

- For navigational DML, the run unit is terminated and an error status of xx19 is returned to the application.
- For SQL, the application receives an SQLCODE of -5 (transaction failure) and an SQLRSN of 1088 (transaction forced to backout).

Effect of Commit Requests

If multiple database sessions share a transaction and one of those sessions issues a commit request, no changes are committed until all top-level sharing sessions that have had activity since the last commit, rollback or start of transaction have issued a commit or until a teleprocessing commit is issued. The term "commit" refers to any DML command that would normally result in committing database changes (COMMIT, FINISH, COMMIT TASK, etc.).

Unless a commit continue request is issued (for which currency locks are retained), all currencies owned by the issuing database session are immediately released; however, implicit exclusive locks and explicit locks acquired by the database session remain until the transaction is committed, even if the request terminates the database session.

Chapter 8: Terminal Management

DC terminal management functions enable your program to transfer data to and from the terminal. You can use one of the following modes to transfer data:

- **Mapping Mode** transfers an entire screen of data on a field-by-field basis. Mapping mode can be used only with 3270-type devices and glass TTYs that have established device-independence tables.
- **Line mode** transfers data one line at a time.
- **Basic mode** transfers a variable amount of data, as specified in the program.

The table below compares the three types of terminal management.

Mode	Data transfer	Device-control characters	Line-control characters	Terminal devices
Mapping	Field-by-field	DC-built	DC-built	3270-type and glass TTYs
Line	One line at a time	DC-built	DC-built	Device independent
Basic	Data length specified in the program	Program	DC-built	Device dependent

For more information about basic mode, see Basic Mode.

This section contains the following topics:

Mapping Mode (see page 193)
Using Pageable Maps (see page 206)
Line Mode (see page 223)

Mapping Mode

In mapping mode, your program communicates with 3270-type terminal devices. DC uses maps to associate screen positions on the terminal with fields in program variable storage.

Example of Map Data Fields

The EMPDISPM map below associates row 4, column 24, with the EMP-ID-0415 field in variable storage; the map associates row 5, column 24, with the EMP-LAST-NAME-0415 field, and so on.

```
        *** EMPLOYEE INFORMATION SCREEN ***

      EMPLOYEE ID : ____
        LAST NAME : ____________
       FIRST NAME : ________________
          ADDRESS : ________________
                  : ____________
                  : __ __________

       DEPARTMENT : ____________________




ENTER AN EMPLOYEE ID AND PRESS ENTER  ***  PRESS CLEAR TO EXIT
```

Creating a Map

To transfer data in mapping mode, you must first create a map by using either the online or batch compiler of the CA IDMS Mapping Facility. You associate map variable fields with either database records or IDD-defined work records.

Maps are available as load modules to the DC run-time system. DC views map load modules as programs.

Mapping Mode Terminal Management

Using mapping mode terminal management, you can perform the following functions:

- Write data to a terminal screen
- Read data input from a terminal screen and query the status of conditions related to the input operation
- Modify previously established map and map field options
- Write unlimited detail occurrences that can be displayed one page at a time by using a pageable map

Mapping Terminology

You should understand the following terms related to maps:

- **Attribute byte**-The nondisplayable byte that begins each map field at run time. The contents of the attribute byte determine the characteristics of the field (such as protection and intensity). Attributes bytes are a 3270 feature.
- **Automatic editing and error handling**-An optional map feature that can be used to perform editing and error-handling functions at run time. These functions can compare input and output data with internal and external pictures, validate data against edit tables, and encode or decode data through code tables.
- **Modified data tag (MDT)**-:ih1.modified data tag The internal switch for a map data field that indicates whether the value in that field has been changed by the user. Modified data tags are a 3270 feature.
- **Write control character (WCC)**-The internal character that holds various specifications for the display of the map such as resetting the keyboard to allow user input. Write control characters are a 3270 feature.

For a complete description of maps and map attributes, see *CA IDMS Mapping Facility* Guide.

Housekeeping

To define the map to the precompiler at compile time, and to establish addressability to DC at run time, you must perform certain mapping mode housekeeping functions:

- **Identify the map you want to use** by including a MAP SECTION (COBOL), a DECLARE MAP statement (PL/I), or the MAP parameter in the @INVOKE statement (Assembler).
- **Copy the map request block (MRB) and the map records** by including compiler-directive statements in program variable storage.
- **Establish addressability between DC and the MRB** by issuing a BIND MAP statement.
- **Establish addressability to map records** by issuing a BIND MAP RECORD statement for each record defined for the map.

For more information on mapping mode housekeeping statements, see the language-specific *CA IDMS DML Reference Guide*.

Displaying Screen Output

To display a map on the terminal screen, perform the following steps:

1. Issue mapping mode housekeeping statements as described above.
2. Initialize variable-storage data fields as needed.
3. Transfer data from variable-storage data fields to map fields on the screen by issuing a MAP OUT statement.

You can also use the MAP OUT statement to transfer data between two variable-storage data fields; this is referred to as a *native mode data transfer*.

For more information about native mode data transfers, see the language-specific *CA IDMS DML Reference Guide*.

Pageable maps have different output considerations. For more information, see Using Pageable Maps.

Mapping Considerations

You need to know about the following considerations when writing a program that displays maps:

- Sending informational messages to the user
- Keeping the data stream short
- Choosing asynchronous or synchronous processing

Sending Informational Messages

You can send a variety of messages to the user's terminal, depending on the situation. For example, if the application is being accessed for the first time, you might transmit the following message:

ENTER AN EMPLOYEE ID AND PRESS ENTER **** PRESS CLEAR TO EXIT

You might send a different message with the same map at another time to indicate the completion status of a task:

**** SPECIFIED EMPLOYEE CANNOT BE FOUND ****

'COBOL and PL/I programmers'. To avoid unpredictable results at run time, specify messages that are 100 bytes or less in length.

Keeping the Data Stream Short

Because you want to promote the fastest possible response time, an important programming consideration is the length of the data stream transmitted to or from the terminal. You should ensure that your program always transmits the smallest amount of data necessary to successfully complete a mapping operation.

Ways to minimize the data stream include:

- **Avoid rewriting literals**. If you are rewriting to the same map, there usually is no need to retransmit literal fields. Specify the NEWPAGE and the LITERALS options only on an initial map output.
- **Transmit only the attribute bytes**. If your program determines that the user has entered invalid data, you need not retransmit the invalid values; these values are still listed on the terminal screen. Instead, you can specify OUTPUT DATA IS ATTRIBUTE to transmit only the attribute bytes for map fields.

  The ATTRIBUTE specification is useful when sending error messages to the terminal because DC still transmits the data in the message field. For example, you could minimize the data stream transmitted by coding the following MAP OUT statement:

  ```
  MAP OUT USING DEPTMAP
    OUTPUT DATA IS ATTRIBUTE
    MESSAGE IS ID-EDIT-ERROR-MESS TO ID-EDIT-ERROR-MESS-END.
  ```

  If automatic editing and error handling are enabled and you use the ERROR option of the MODIFY MAP statement, the ATTRIBUTE specification is automatically invoked.

Synchronous and Asynchronous Processing

Mapping mode supports synchronous and asynchronous map output operations:

- During a **synchronous** map output request, DC places your task in an inactive state until processing is complete.

  To issue a synchronous map output request, specify the WAIT option of the MAP OUT statement. This option allows you to ensure that the output request was completed successfully before continuing program processing.

- During an **asynchronous** map output request, DC returns control to your task before the output processing is complete. Before issuing subsequent map output requests, you must ensure that the first request is finished by issuing a CHECK TERMINAL request. CHECK TERMINAL is a basic mode DML statement that is described in Basic Mode.

  To issue an asynchronous map output request, specify the NOWAIT option of the MAP OUT statement.

  You may want to specify NOWAIT if your program issues a MAP OUT just before task termination. This causes DC to release a task's resources sooner. In this case, however, you cannot issue the CHECK TERMINAL statement; you won't be able to determine the completion status of the MAP OUT operation.

Example of an Initial Application Screen

The program excerpt below displays an application's initial screen. It initializes the EMP-ID-0415 field and displays the screen, soliciting user input.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  TSK02                   PIC X(8)  VALUE 'TSK02'.
 01  MESSAGES.
     05 INITIAL-MESSAGE         PIC X(54) VALUE
      'ENTER AN EMPLOYEE ID AND PRESS ENTER *** CLEAR TO EXIT'.
     05 INITIAL-MESSAGE-END      PIC X.
 PROCEDURE DIVISION.
*** ESTABLISH ADDRESSABILITY TO MAP ***
     BIND MAP SOLICIT.
*** ESTABLISH ADDRESSABILITY TO MAP RECORDS ***
     BIND MAP SOLICIT RECORD EMPLOYEE.
     BIND MAP SOLICIT RECORD DATE-WORK-REC.
     MOVE ZERO TO EMP-ID-0415.
*** DISPLAY THE MAP ***
     MAP OUT USING SOLICIT
       WAIT NEWPAGE
       MESSAGE IS INITIAL-MESSAGE TO INITIAL-MESSAGE-END.
*** RETURN CONTROL TO CA-IDMS/DC NEXT TASK TSK02 ***
     DC RETURN
       NEXT TASK CODE TSK02.
```

Reading Screen Input

When the user finishes inputting data and presses an AID key, DC invokes the specified input task. The task reads data from the screen and tests for certain input conditions.

To transfer data from map fields on the terminal screen to the corresponding variable storage data fields, perform the following steps:

1. Issue mapping mode housekeeping statements, as explained in Housekeeping.
2. Transfer data from map fields on the terminal screen to variable-storage data fields by issuing a MAP IN statement.

You can use the MAP IN statement to transfer data between two variable-storage data fields; this is referred to as a *native mode data transfer*.

For more information about native mode data transfers, see the language-specific *CA IDMS DML Reference Guide*.

Pageable maps have different input considerations. For more information, see Using Pageable Maps.

Example of Reading Input

The program excerpt below reads data from the screen.

It transfers data from the terminal screen to map data fields in program variable storage by issuing a MAP IN statement.

```
 PROCEDURE DIVISION.
*** ESTABLISH ADDRESSABILITY TO THE MAP ***

    BIND MAP SOLICIT.
*** ESTABLISH ADDRESSABILITY TO THE MAP RECORDS ***
    BIND MAP SOLICIT RECORD EMPLOYEE.
    BIND MAP SOLICIT RECORD EMP-DATE-WORK-REC.
*** TRANSFER DATA FROM MAP DATA FIELDS TO VARIABLE STORAGE ***
    MAP IN USING SOLICIT.
*** FURTHER PROCESSING OF ENTERED DATA ***
```

Testing for Input Conditions

After a MAP IN request, your program can inquire about conditions related to the input operation. For example, you may need to perform processing based on the AID key pressed by the user or determine if the user entered data in a particular map data field.

To test for conditions related to a map input operation, issue an INQUIRE MAP statement. By using this statement, you can obtain the following information:

- The control key pressed.
- The current cursor position.
- Information on conditions regarding a map data field or group of map data fields:
  - Is data present?
  - Has data been modified?
  - Has data been truncated?
  - What is the entered length of a specific map input field?
- Whether specified map fields are in error (the error flag has been set on for those fields) or are correct (the error flag has been set off). This option applies only to those maps and map fields for which automatic editing is enabled.
- Whether the screen was formatted before the input operation was performed.

Frequent uses of the INQUIRE MAP statement are listed below:

- **To determine what control key was pressed**. Typically, an application offers various processing options to the user. Each option can be associated with a control key. Your program should check the AID byte after every MAP IN statement to determine the option chosen. The table below lists the AID characters associated with each 3270-type control key.

Key	AID character
Enter	" ' " (single quote)
Clear	'_' (underscore)
PF1	'1'
PF2	'2'
PF3	'3'
PF4	'4'
PF5	'5'
PF6	'6'
PF7	'7'
PF8	'8'
PF9	'9'
PF10	':'
PF11	'#'
PF12	'@'
PF13	'A'
PF14	'B'
PF15	'C'
PF16	'D'
PF17	'E'
PF18	'F'
PF19	'G'
PF20	'H'
PF21	'I'
PF22	'¢.'
PF23	'.'

Key	AID character
PF24	'<'
PA1	'%'
PA2	'>'
PA3	','

- **To ensure that necessary data has been entered**. You should make sure that the user has entered data in all fields necessary for successful processing.
- **To determine if automatic editing and error-handling have detected any errors**. If input errors are detected, DC automatically transmits only the attribute bytes for the next map output operation.

You can use the TASK CODE parameter of the ACCEPT statement to retrieve the calling task code.

For more information about the ACCEPT statement, see Retrieving Task-Related Information.

The program excerpt below performs processing based on conditions related to the last map input operation. It uses the INQUIRE MAP statement to determine what control key was pressed and to ensure that the DEPT-ID-0410 field contains data.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  DC-AID-CONDITION-NAMES.
     03  DC-AID-IND-V         PIC X.
                  88  ENTER-HIT VALUE QUOTE.
                  88  CLEAR-HIT VALUE '_'.
 PROCEDURE DIVISION.
*** ESTABLISH ADDRESSABILITY TO THE MAP AND MAP RECORDS ***
    BIND MAP SOLICIT.
    BIND MAP SOLICIT RECORD EMPLOYEE.
    BIND MAP SOLICIT RECORD DEPARTMENT.
    BIND MAP SOLICIT RECORD EMP-DATE-WORK-REC.
*** TRANSFER DATA FROM THE MAP TO VARIABLE STORAGE ***
    MAP IN USING SOLICIT.
```

```
*** DETERMINE THE AID KEY PRESSED BY THE TERMINAL OPERATOR ***
    INQUIRE MAP SOLICIT
       MOVE AID TO DC-AID-IND-V.
*** IF OPERATOR PRESSED CLEAR THEN DC RETURN ***
    IF CLEAR-HIT DC RETURN.
*** DETERMINE IF THE TERMINAL OPERATOR    ***
*** ENTERED DATA IN THE DEPT-ID-0410 FIELD ***
    INQUIRE MAP SOLICIT
       IF DFLD DEPT-ID-0410
         DATA IS NO
         GO TO A100-NO-DATA.
      .
*** FURTHER PROCESSING OF ENTERED DATA ***
```

Modifying Map Options

Before issuing an input or output request, you may need to modify a map's WCC options or specify attributes for one or more map data fields. You can make modifications either for the length of the session or for the next mapping operation. For example, you may need to:

- Position the cursor on the next MAP OUT operation
- Require that the user enter data in a specified map data field
- Prevent the user from entering data in specified map data fields (this is especially useful on the initial MAP OUT of a session)
- Require that data from a specified map data field be transmitted regardless of whether it was modified by the user
- Modify the WCC and attribute options for an entire session

Steps to Modify a Map

To modify a map's WCC options or to specify attributes for one or more map data fields, perform the following steps:

1. Issue mapping mode housekeeping statements

   For more information about housekeeping statements, see Housekeeping.

2. Issue the MODIFY MAP command
3. Issue either a MAP IN or MAP OUT statement

Example of Modifying a Map

The program excerpt below uses the MODIFY MAP statement to protect map data fields from operator input. The program is used in an application's initial MAP OUT to help ensure that the user will enter data in the correct field (EMP-ID-0415) by positioning the cursor and preventing input to all other map data fields.

```
 PROCEDURE DIVISION.
    BIND MAP SOLICIT.
    BIND MAP SOLICIT RECORD EMPLOYEE.
    BIND MAP SOLICIT RECORD EMP-DATE-WORK-REC.
*** SET CURSOR AND PREVENT INPUT INTO ALL BUT EMP-ID-0415 ***
    MODIFY MAP SOLICIT TEMPORARY
      CURSOR AT DFLD EMP-ID-0415
      FOR ALL EXCEPT DFLD EMP-ID-0415
      ATTRIBUTES PROTECTED.
*
    MOVE ZERO TO EMP-ID-0415.
    MAP OUT USING SOLICIT
      YES NEWPAGE
      MESSAGE IS INITIAL-MESSAGE TO INITIAL-MESSAGE-END.
*
    DC RETURN
      NEXT TASK CODE TSK02.
```

Writing and Reading in One Step

To write data to the terminal and read data input from the terminal in one synchronous operation, issue a MAP OUTIN statement.

Important! MAP OUTIN forces your program to be conversational; it is not recommended.

If your application needs to write and read in one step, perform the following steps:

1. Issue mapping mode housekeeping statements

   For more information about housekeeping statements, see Housekeeping.

2. Modify map or map data fields,

   For more information about modifying map data fields, see Modifying Map Options.

3. Initialize variable-storage data fields as needed

4. Transfer data from variable-storage data fields to map fields on the terminal screen and back again by issuing the MAP OUTIN statement

Suppressing Map Error Messages

You can suppress the display of error messages for map fields. For example, you can code a data validation test so that it suppresses a map field's default error message and displays a different message when the field is in error.

What to Do

Include the ERROR MESSAGE IS ACTIVE/SUPPRESS parameter on your MODIFY MAP statement. ERROR MESSAGE immediately follows the REQUIRED/OPTIONAL parameter.

Example of Suppressing Error Messages

This COBOL example issues a MODIFY MAP statement that suppresses the display of default error messages for the ORDER-AMOUNT field on the current map.

In this application, the data validation routine compares the ORDER-AMOUNT field with the number of widgets on hand. If the current stock restricts the size of ORDER-AMOUNT, an alternative message is displayed.

1. Define an alternative message in working storage. For example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  MESSAGES.
   05 INITIAL-MESSAGE          PIC X(80) VALUE
    'ENTER A NUMERIC ORDER-AMOUNT AND PRESS ENTER'.
   05 EDIT-ERROR-MESSAGE       PIC X(80) VALUE
    'ORDER-AMOUNT EITHER NOT ENTERED OR NOT NUMERIC'.
   05 INVENTORY-MESSAGE        PIC X(80) VALUE
    'NOT ENOUGH WIDGETS IN STOCK TO DELIVER THAT AMOUNT'.
   05 DISPLAY-MESSAGE          PIC X(80) VALUE
    'CLEAR TO EXIT ** ENTER ORDER-AMOUNT AND ENTER TO CONTINUE'
```

2. Modify the map to display alternative messages when a specific error is found:

```
MODIFY MAP MAP01 TEMPORARY
  FOR DFLD ORDER-AMOUNT
    ERROR MESSAGE IS SUPPRESS.
```

3. Perform your data validation routine. For example, you can compare the number of widgets in stock to ORDER-AMOUNT. If ORDER-AMOUNT is greater than the number in stock, issue an alternative message indicating that the order cannot be filled.

   If the data validation routine indicates that there are not enough widgets in stock, display the map with the alternative message.

TEMPORARY and PERMANENT Options

The use of the SUPPRESS option is affected by the TEMPORARY/PERMANENT option:

- If TEMPORARY is specified, error messages are suppressed for the next mapout only.
- If PERMANENT is specified, error messages are suppressed until the program terminates or until the error message specifications are overridden by a subsequent MODIFY MAP statement.

Testing for Identical Data

You can compare the contents of a mapped-in field with the map data that is currently in your program's record buffer.

This means that you can test whether a map field contains the same data that was previously mapped out. By comparing the fields, your program updates the database only when the user enters different data, reducing the number of database I/O operations.

How this Relates to MDT Settings

The input test condition does not test a field's modified data tag (MDT). For example, the statement INQUIRE MAP MAP01 DATA IS IDENTICAL is *true* in either of the following cases:

- The field's MDT is off. On mapin, the MDT is usually off if the user did not type any characters in the field.
- The field's MDT is on, but each character that the user typed in is identical (including capitalization) to the data in variable storage.

What to Do

Include the IDENTICAL/DIFFERENT parameter in your INQUIRE MAP statement.

Example of Testing for Identical Data

This COBOL example uses an INQUIRE MAP statement to test whether the user has entered an employee ID number:

- If the IDENTICAL condition is *true* (the user doesn't specify a different ID number), the program displays the menu screen
- If the IDENTICAL condition is *false* (the user specifies a different ID number), the program obtains the corresponding employee record from the database

The sample INQUIRE MAP statement is shown below:

```
INQUIRE MAP MAP01

 IF DFLD EMP-ID-0415 DATA IS IDENTICAL THEN
   PERFORM EMP-PROMPT-20
 ELSE
   PERFORM EMP-OBTAIN-20.
```

Example of Testing for Changed Data

This COBOL example uses an INQUIRE MAP statement to test whether the user has entered a new department ID or department name. If the user has changed either value (DIFFERENT is *true*), the program branches to DEPTUP-30.

```
INQUIRE MAP MAP02
 IF ANY DFLD DEPT-ID-0410
      DFLD DEPT-NAME-0410 DATA IS DIFFERENT
 THEN PERFORM DEPTUP-30.
```

Using Pageable Maps

A pageable map can contain more occurrences of a set of map fields tha CAn fit on the screen at one time; therefore, it can contain unlimited occurrences of the set of map fields. Each occurrence of the multiply-occurring set is called a **detail occurrence**. The MAP OUT and MAP IN statements can create, retrieve, and modify detail occurrences of a pageable map.

About Pageable Maps

You should know about the following aspects of pageable maps:

- The format of a pageable map
- How to conduct a map paging session
- How to code an application that allows the user to browse through a pageable map but not update it
- How to code an update application that allows the user to perform database updates by using a pageable map

Pageable Map Format

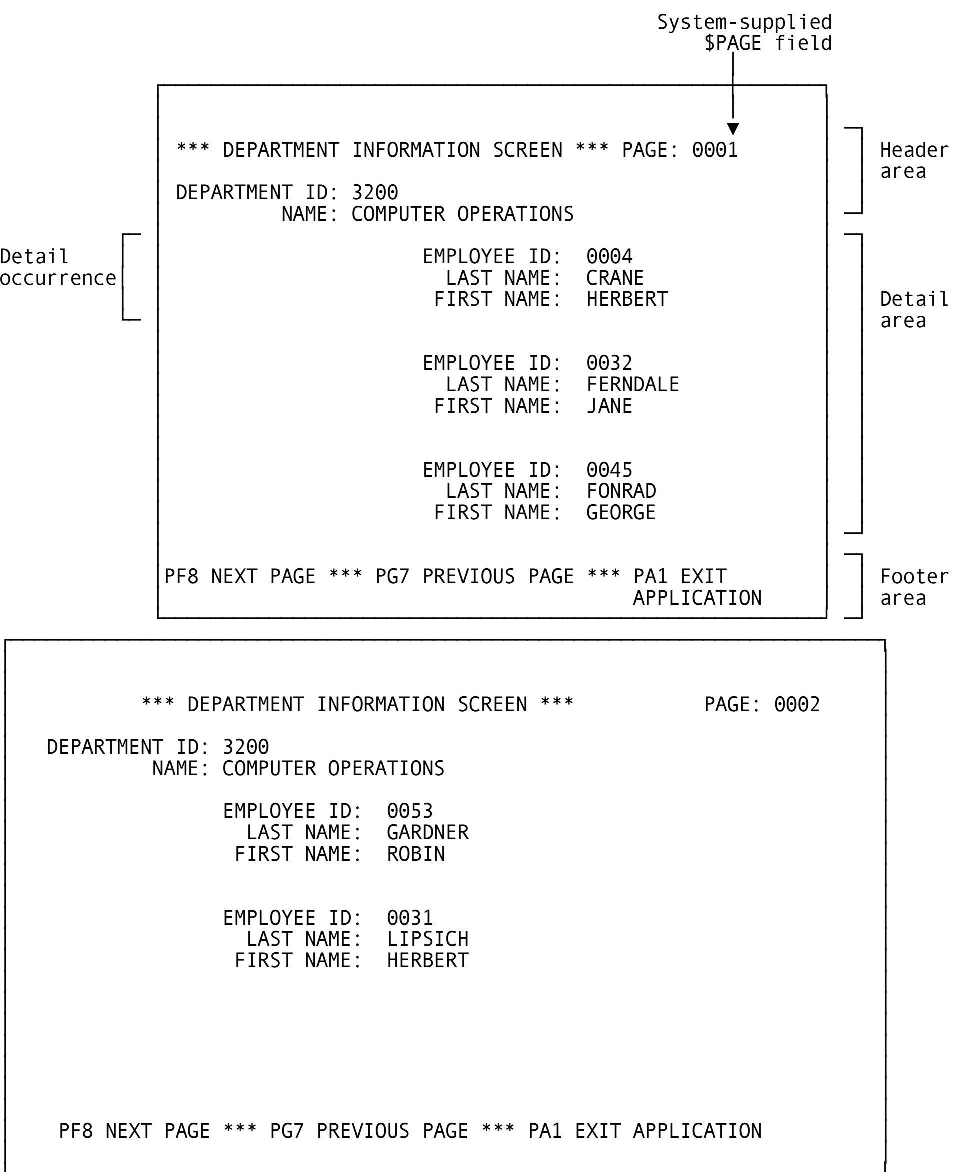
A pageable map is divided into the **header area**, the **detail area**, and the **footer area**. The header and footer areas consist of general information such as the map title, the page number, or the useable PF-keys. The detail area consists of detail occurrences.

For information on defining a pageable map, see *CA IDMS Mapping Facility Guide*.

Important! To prevent excessive database record locking, you should not define database records as map records in a pageable map; use IDD-defined work records instead.

Example of a Pageable Map

The figures below illustrate two pages of a map screen. Note that the display of information in the header and footer areas is unchanged except for the $PAGE field.



Conducting a Map Paging Session

A map paging session involves interaction among the user, the run-time mapping system, and your map paging application program. You should understand this interaction and the sequence of events that occurs during a map paging session before planning the logic of your application program.

Typical Map Paging Sequence

This sequence of events typically occurs during a map paging session:

1. Your program begins the session and defines map paging parameters
2. The program creates detail occurrences
3. A map page is displayed on the terminal
4. The user pages forward and backward through the pageable map
5. The user optionally modifies map data fields
6. The program receives control and updates the database
7. The user ends the map paging session

The following discussion describes each step in detail.

Beginning the Paging Session

A map paging session begins when your program issues a STARTPAGE statement. Options included in this statement specify the following:

- **The runtime flow of control**. The paging type (NOWAIT/WAIT/RETURN) determines whether the run-time mapping system or your program receives control when the user presses a control key, as detailed in the table below.

  The paging type affects the frequency with which your program will receive control and the processing logic you must provide. For example, in NOWAIT, run-time mapping performs all paging operations for you; in WAIT and RETURN, you must provide coding logic that performs the paging operations specified by the user.

  NOWAIT is best for applications in which the user can display but not update; WAIT and RETURN are best for update applications.

- **Whether the user can display a previous map page**. If backpaging is allowed, the run-time system must maintain the resources that describe the detail occurrences of previous pages. If backpaging is not allowed, the run-time system deletes all previous pages of detail occurrences when a new map page is displayed.

  Note: Always allow backpaging for pageable map applications that perform database updates.

- **Whether the user can update map data fields**. A paging mode of UPDATE specifies that the user can modify map data fields, subject to restrictions specified in the map and by previous MODIFY MAP statements. BROWSE specifies that the user can modify only the system-supplied $PAGE field (if present).

The tables below summarize flow of control in a map paging session.

- Paging request*:

Paging Type	No Data Fields Modified	Data Fields Modified**
NOWAIT	Run-time mapping displays the requested map page	Run-time mapping displays the requested map page
WAIT	Run-time mapping displays the requested map page	Control passes to the program
RETURN	Control passes to the program	Control passes to the program

* If the user presses Clear, PA1, PA2, or PA3, and that key is not associated with backward or forward paging, refer instead to "Nonpaging request" below.

** If the user presses Clear, PA1, PA2, or PA3, refer to the "No data fields modified" column.

- Nonpaging request:

Paging Type	No Data Fields Modified	Data Fields Modified**
NOWAIT	Control passes to the program	Run-time mapping redisplays the same map page
WAIT	Control passes to the program	Control passes to the program
RETURN	Control passes to the program	Control passes to the program

** If the user presses Clear, PA1, PA2, or PA3, refer to the "No data fields modified" column.

Creating Detail Occurrences Your program retrieves data, moves it to map data fields, and creates detail occurrences by issuing MAP OUT DETAIL commands.

Displaying the First Page

The first page is displayed on the terminal screen in one of the following ways:

- **Run-time mapping** automatically displays the first map page when the first detail occurrence of the second page of occurrences is created. The program continues to execute and create additional detail occurrences.

  When the first page is displayed by run-time mapping, DC returns a status of 4676 (DC-FIRST-PAGE-SENT). Your program must check for this status after every MAP OUT DETAIL statement.

- **Your program displays the first map page**. When all detail occurrences are created, your program should check to determine if the first page was written to the terminal. You do this by setting a switch when DC returns a status of 4676 (DC-FIRST-PAGE-SENT). If 4676 was never returned, your program explicitly displays the first map page by issuing a MAP OUT RESUME statement.

Paging Forward and Backward

To specify the next map page to be displayed, the user does one of the following:

- Presses the control key associated with paging forward one page
- Presses the control key associated with paging backward one page
- Changes the $PAGE map field, if defined on the map, and presses a control key other than Clear, PA1, PA2, or PA3.

Modifying Map Fields

The user can change map data fields, including header and footer data fields, subject to restrictions specified by the STARTPAGE command (UPDATE/BROWSE) or by a previously specified MODIFY MAP command.

Updating the Database

If the user has modified any map data fields or if the paging type is RETURN, the program reads modified detail occurrences and updates the database.

A modified detail occurrence contains one or more map fields whose modified data tags (MDTs) are set on.

To retrieve a modified detail occurrence, issue a MAP IN DETAIL statement. MAP IN DETAIL can retrieve a modified detail occurrence sequentially, by the order of detail occurrences, or randomly by a key value that can be associated with an occurrence. If sequential or random retrieval cannot retrieve a modified detail occurrence, DC returns a status of 4668 (DC-NO-MORE-UPD-DETAILS).

If you need to modify the current detail occurrence (for example, to send an error message), issue a MAP OUT DETAIL CURRENT statement. This statement modifies the detail occurrence most recently referenced by a MAP IN DETAIL or MAP OUT DETAIL statement.

After processing all modified detail occurrences, write the map to the terminal screen by issuing a MAP OUT RESUME statement. If WAIT or RESUME has been specified, your program is responsible for displaying the next page specified by the user.

If you need to create additional detail occurrences, you can do so at any time by issuing further MAP OUT DETAIL statements. The new occurrences are stored at the end of the set of detail occurrences.

Ending the Paging Session

When a map paging session ends, the system deletes all the detail occurrences created during the session. To end a session, issue an ENDPAGE SESSION command.

How to Code a Browse Application

To write a pageable map application that allows the user to display data but not update it, perform the following steps:

1. Establish a switch in variable storage. This switch should be set on if run-time mapping has transmitted the first page.
2. Issue mapping mode housekeeping statements, as explained in Housekeeping.
3. Initiate the map paging session by issuing a STARTPAGE statement that specifies NOWAIT and BROWSE.
4. Initialize header data fields.
5. Perform the following steps iteratively until all data is retrieved:
   a. Perform database retrieval and move data to map data fields in variable storage.
   b. Issue a MAP OUT DETAIL NEW statement, checking for a status of 4676 (DC-FIRST-PAGE-SENT).
   c. Set the first-page switch if 4676 is returned; perform the IDMS-STATUS routine if 4676 is not returned.

Ending the Browse Session

If, after all detail occurrences have been created, the first-page switch is not set, you should transmit the map page to the terminal screen by issuing a MAP OUT RESUME statement.

The next task specified in the DC RETURN NEXT TASK CODE statement should include logic that tests to see if the user has indicated the end of the map paging session. If so, issue an ENDPAGE SESSION statement.

Example of a Browse Application

The program excerpt below shows a pageable map application in which run-time mapping handles all paging requests (paging type of NOWAIT) and the operator cannot make updates (paging mode of BROWSE).

After acquiring the data passed from a previous task and establishing that database records are present, this program issues MAP OUT DETAIL statements iteratively until all detail occurrences are written. DEPTEND, which is specified as the next task, ends the paging session with the ENDPAGE command and performs processing based on the control key pressed.

```
 DATA DIVISION
 01  FIRST-PAGE-SW          PIC X  VALUE 'N'.
       88 LESS-THAN-A-PAGE      VALUE 'N'.
 01  MAP-WORK-REC.
     05 WORK-FIRST            PIC X(10).
     05 WORK-LAST             PIC X(15).
     05 WORK-EMP-ID           PIC X(4).
 LINKAGE SECTION.
 01  PASS-DEPT-INFO.
     05 PASS-DEPT-ID          PIC 9(4).
     05 PASS-DEPT-INFO-END    PIC X.

 PROCEDURE DIVISION.
     BIND MAP DCTEST01.
     BIND MAP DCTEST01 RECORD MAP-WORK-REC.
*** ACQUIRE DEPT-ID FROM ERROR CHECKING PROGRAM ***
     GET STORAGE FOR PASS-DEPT-INFO TO
                 PASS-DEPT-INFO-END
       WAIT SHORT USER
       STGID 'RKNS'.
*
     MOVE PASS-DEPT-ID TO DEPT-ID-0410.
```

```
    FREE STORAGE STGID 'RKNS'.
*
    COPY IDMS SUBSCHEMA-BINDS.
    READY USAGE-MODE RETRIEVAL.
*
    OBTAIN CALC DEPARTMENT
     ON DB-REC-NOT-FOUND GO TO NO-DEPT-ERR.
    IF DEPT-EMPLOYEE IS EMPTY
        GO TO NO-EMP-ERR.
*** BEGIN MAP PAGING SESSION ***
    STARTPAGE SESSION DCTEST01 NOWAIT BACKPAGE BROWSE.
    PERFORM A100-GET-EMPLOYEES THRU A100-EXIT
                UNTIL DB-END-OF-SET.
    FINISH.
*** IF FIRST PAGE NOT YET SENT, MAP OUT RESUME ***
    IF LESS-THAN-A-PAGE
       MAP OUT USING DCTEST01 RESUME.
*** NEXT TASK ENDS PAGING SESSION ***
    DC RETURN NEXT TASK CODE 'DEPTEND'.

 A100-GET-EMPLOYEES.
    OBTAIN NEXT WITHIN DEPT-EMPLOYEE
       ON DB-END-OF-SET GO TO A100-EXIT.
    MOVE EMP-FIRST-NAME-0415 TO WORK-FIRST.
    MOVE EMP-LAST-NAME-0415 TO WORK-LAST.
    MOVE EMP-ID-0415 TO WORK-EMP-ID.
*** MAP OUT CURRENT DETAIL, CHECK FOR ERROR-STATUS OF 4676 ***
    MAP OUT USING DCTEST01
        DETAIL NEW
        ON DC-FIRST-PAGE-SENT
           MOVE 'Y' TO FIRST-PAGE-SW.
 A100-EXIT.
    EXIT.
 .
 .
 .
*** FURTHER PROCESSING, INCLUDING ERROR ROUTINES ***
```

How to Code an Update Application

To write a pageable map application that allows the user to update map data fields, establish a retrieval program and an update program.

Retrieval Program

The retrieval program initiates the pageable map update session and retrieves and displays the data. This program can be similar to the one displayed in How to Code a Browse Application. You should make the following changes to the retrieval program:

- Specify one of the following options of the STARTPAGE statement:
  - **WAIT** causes your program to acquire control after every *update* paging request and after every nonpaging request.
  - **RETURN** causes your program to acquire control after *every* paging request (update or nonupdate) and every nonpaging request.

  Note: Because of editing and error-handling considerations, updating pageable maps by using a paging type of NOWAIT is not recommended.

- Use the KEY IS parameter of the MAP OUT DETAIL statement to pass the db-key of each retrieved record:

```
MAP OUT USING DCTEST01
  DETAIL NEW
  KEY IS DBKEY.
```

  Including the db-key in this manner allows for DB-KEY retrieval in subsequent tasks.

- Code a DC RETURN statement that indicates the pageable map update program to be invoked when the user presses a control key.

Update Program

The update program retrieves modified detail occurrences and updates the database. In this program, perform the following steps:

1. Establish a switch in variable storage. This switch should be set on if your program encounters any invalid data in modified detail occurrences.
2. Issue mapping mode housekeeping statements, as explained in Housekeeping.
3. Issue a MAP IN HEADER statement that includes the PAGE option. You can use the PAGE value later in your program when determining the next page to map out.
4. Issue an INQUIRE MAP statement to determine what control key was pressed. The control key pressed by the user can specify:
   - **The flow of control**. You can associate certain control keys with specific functions (for example, Clear might always exit the application).
   - **The next page to be displayed**. The user can indicate the next page to be displayed by pressing a site-standard paging control key.
   - **A user error**. If the user presses an invalid control key, you should redisplay the current page.
5. Perform the following steps iteratively until all modified detail occurrences have been mapped in:
   a. Issue a MAP IN DETAIL statement that includes the RETURNKEY parameter.
   b. Check for a status of 4668 (DC-NO-MORE-UPD-DETAILS). If 4668 is returned, all updated details have been returned and you should display the pageable map, as specified by the user. If 4668 is not returned, perform the IDMS-STATUS routine.

c. Perform error and range checking to ensure that the user entered valid data. If invalid data is found, set the error switch and issue a MAP OUT DETAIL CURRENT statement that includes a message that indicates the error.

d. Perform database retrieval to access the database record to be modified. Retrieve the record by using its db-key (acquired from the RETURNKEY parameter). If data cannot be retrieved, set the error switch and issue a MAP OUT DETAIL CURRENT statement that includes a message that indicates the error.

e. Move data from the work record to the database record.

f. Issue database modification statements.

g. After all modified detail occurrences have been successfully processed, issue a MAP OUT RESUME statement that specifies the page requested by the user. If errors were encountered in the MAP IN DETAIL processing, you should redisplay the current page so the operator can correct the invalid data.

Ending the Update Session

The next task specified in the DC RETURN NEXT TASK CODE statement should include logic that tests to see if the user has indicated the end of the map paging session. If so, issue an ENDPAGE SESSION statement.

Example of an Update Application

The program excerpt below shows a pageable map update application. The program contains paging logic that works with a paging type of either WAIT or RETURN.

After determining user specifications, the program issues MAP IN DETAIL statements iteratively, modifying the database as specified, until all modified detail occurrences are processed.

```
DATA DIVISION
WORKING-STORAGE SECTION.
01  RETURN-DBKEY          PIC S9(8) COMP.
01  DEPTMOD             PIC X(8) VALUE 'DEPTMOD'.
01  FIRST-PAGE-SW         PIC X    VALUE 'N'.
      88 LESS-THAN-A-PAGE  VALUE 'N'.
01  MAP-IN-ERR-SW         PIC X    VALUE 'N'.
      88 MAP-IN-ERR       VALUE 'Y'.
01  PAGE-INDICATOR.
    05 SPEC-PAGE          PIC S9(8) COMP.
01  MESSAGES.
```

```
    05 EDIT-ERR-MESS          PIC X(21)
      VALUE 'CORRECT INVALID INPUT'.
    05 EDIT-ERR-MESS-END      PIC X.
*
    05 EMP-NOT-FOUND-MESS     PIC X(18)
      VALUE 'EMPLOYEE NOT FOUND'.
    05 EMP-NOT-FOUND-MESS-END  PIC X.
 01  DC-AID-CONDITION-NAMES.
     03  DC-AID-IND-V         PIC X.
                 88  ENTER-HIT VALUE QUOTE.
                 88  CLEAR-HIT VALUE '_'.
                 88  PF01-HIT VALUE '1'.
                 88  PF02-HIT VALUE '2'.
                 88  PF03-HIT VALUE '3'.
                 88  PF04-HIT VALUE '4'.
                 88  PF05-HIT VALUE '5'.
                 88  PF06-HIT VALUE '6'.
                 88  PF07-HIT VALUE '7'.
                 88  PF08-HIT VALUE '8'.
                 88  PF09-HIT VALUE '9'.
                 88  PF10-HIT VALUE ':'.
                 88  PF11-HIT VALUE '#'.
                 88  PF12-HIT VALUE '@'.
                 88  PF13-HIT VALUE 'A'.
                 88  PF14-HIT VALUE 'B'.
                 88  PF15-HIT VALUE 'C'.
                 88  PF16-HIT VALUE 'D'.
                 88  PF17-HIT VALUE 'E'.
                 88  PF18-HIT VALUE 'F'.
                 88  PF19-HIT VALUE 'G'.
                 88  PF20-HIT VALUE 'H'.
                 88  PF21-HIT VALUE 'I'.
                 88  PF22-HIT VALUE '_'.
                 88  PF23-HIT VALUE '.'.
                 88  PF24-HIT VALUE '>'.
                 88  PA01-HIT VALUE '%'.
                 88  PA02-HIT VALUE '<'.
                 88  PA03-HIT VALUE ','.
                 88  PEN-ATTN-SPACE-NULL VALUE '='.
                 88  PEN-ATTN VALUE QUOTE.
 01  MAP-WORK-REC.
    05 WORK-EMP-ID         PIC X(4).
    05 WORK-FIRST          PIC X(10).
    05 WORK-LAST           PIC X(15).
```

```
 PROCEDURE DIVISION.
    BIND MAP DCTEST01.
    BIND MAP DCTEST01 RECORD MAP-WORK-REC.
    MOVE 'N' TO MAP-IN-ERR-SW.
*** MAP IN HEADER AND PAGE FIELD ***
    MAP IN USING DCTEST01
        HEADER
        PAGE IS SPEC-PAGE
        ON DC-DETAIL-NOT-FOUND
           NEXT SENTENCE.
*** DETERMINE THE PF-KEY PRESSED ***
    INQUIRE MAP DCTEST01 MOVE AID TO DC-AID-IND-V.
    IF PA01-HIT
        ENDPAGE
        DC RETURN.
*** CHECK FOR HEADER ERRORS, MAP OUT IF ANY ARE FOUND ***
    INQUIRE MAP DCTEST01
       IF ANY EDIT IS ERROR
         THEN
           MODIFY MAP DCTEST01 TEMPORARY
                FOR ALL ERROR FIELDS
                ATTRIBUTES BRIGHT
           MAP OUT USING DCTEST01 RESUME
           DC RETURN NEXT TASK CODE DEPTMOD.
*
    COPY IDMS SUBSCHEMA-BINDS.
    READY ORG-DEMO-REGION USAGE-MODE IS UPDATE.
    READY EMP-DEMO-REGION USAGE-MODE IS UPDATE.
*
    PERFORM A100-MAP-IN-DETAILS THRU A100-EXIT
           UNTIL DC-NO-MORE-UPD-DETAILS.
    FINISH.
*** PAGING ROUTINES FOLLOW                ***
*** IF ERROR SWITCH IS SET, REDISPLAY CURRENT PAGE ***
    IF MAP-IN-ERR
      THEN
        MAP OUT USING DCTEST01
            RESUME PAGE IS CURRENT
        DC RETURN NEXT TASK CODE DEPTMOD.
*** IF PF07, DISPLAY PRIOR PAGE ***
    IF PF07-HIT
      THEN
       MAP OUT USING DCTEST01
            RESUME PAGE IS PRIOR
       DC RETURN NEXT TASK CODE DEPTMOD.
.hr left right

*** IF PF08, DISPLAY NEXT PAGE ***
```

```
    IF PF08-HIT
     THEN
      MAP OUT USING DCTEST01
         RESUME PAGE IS NEXT
      DC RETURN NEXT TASK CODE DEPTMOD.
*** ELSE, USE PAGE VALUE FROM MAP IN HEADER ***
    MAP OUT USING DCTEST01
       RESUME PAGE IS SPEC-PAGE.
    DC RETURN NEXT TASK CODE DEPTMOD.
 A100-MAP-IN-DETAILS.
*** MAP IN EACH MODIFIED DETAIL.  EXIT   ***
*** WHEN NO MORE MODIFIED DETAILS REMAIN ***
    MAP IN USING DCTEST01
      DETAIL
      RETURNKEY IS RETURN-DBKEY
      ON DC-NO-MORE-UPD-DETAILS GO TO A100-EXIT.
*** IF ERROR, MAP OUT DETAIL WITH MESSAGE, SET SWITCH ***
    INQUIRE MAP DCTEST01
      IF ANY EDIT IS ERROR
       THEN
        MODIFY MAP DCTEST01 TEMPORARY
           FOR ALL ERROR FIELDS
           ATTRIBUTES BRIGHT
        MAP OUT USING DCTEST01
          MESSAGE IS EDIT-ERR-MESS
              TO EDIT-ERR-MESS-END
          DETAIL CURRENT
          KEY IS RETURN-DBKEY
        MOVE 'Y' TO MAP-IN-ERR-SW
      GO TO A100-EXIT.
*** RETRIEVE EMPLOYEE, USING DBKEY FROM RETURNKEY ***
    OBTAIN EMPLOYEE DB-KEY IS RETURN-DBKEY
           ON ANY-STATUS NEXT SENTENCE.
*** IF ERROR, MAP OUT DETAIL WITH MESSAGE, SET SWITCH ***
    IF DB-REC-NOT-FOUND
      MAP OUT USING DCTEST01
        MESSAGE IS EMP-NOT-FOUND-MESS
            TO EMP-NOT-FOUND-MESS-END
           DETAIL CURRENT
           KEY IS RETURN-DBKEY
      MOVE 'Y' TO MAP-IN-ERR-SW
      GO TO A100-EXIT
```

```
    ELSE
      PERFORM IDMS-STATUS.
*
    MOVE WORK-FIRST  TO EMP-FIRST-NAME-0415.
    MOVE WORK-LAST   TO EMP-LAST-NAME-0415.
    MOVE WORK-EMP-ID TO EMP-ID-0415.
    MODIFY EMPLOYEE.
 A100-EXIT.
    EXIT.
```

Overriding Automatic Mapout for Pageable Maps

You can override the automatic mapout of a pageable map's first page.

By default, the first page of a pageable map is displayed as soon as the first detail occurrence of the second map page is written to scratch.

You can override this automatic mapout by specifying NOAUTODISPLAY in your STARTPAGE statement. By overriding the automatic display of the map's first page, you can add messages or modify the map before the page is displayed.

Return Code for Map Page Built

A map paging return code tells you before mapout whether a map page has been built.

The table below lists the map paging return code for map page built in COBOL, PL/I, and Assembler.

The listed code is returned as soon as a map page is built and before mapout.

Language	Return code	Description
COBOL and PL/I	4680	■ Returned in: IDMS communications block status code field ■ Returned after: MAP OUT DETAIL statement for a pageable map ■ Represented by the COBOL 88-level status code DC-PAGE-READY.
Assembler	X'50'	■ Returned in: DC/UCF run-time register 15 ■ Returned after: #MREQ OUT DETAIL=YES statement for a pageable map

How to Code a Noautosave Application

To code a pageable map application that does not automatically mapout when the first map page is built, perform the following steps:

1. Issue mapping mode housekeeping statements, as explained in Housekeeping.
2. Initiate a map paging session by issuing a STARTPAGE statement that specifies NOAUTODISPLAY.

   ```
   STARTPAGE SESSION MAP01 NOAUTODISPLAY
   ```

3. Initialize header data fields.

Map Out Detail Occurrences

Perform the following steps iteratively until all data is retrieved:

1. Perform database retrieval and move data to map data fields in variable storage.
2. Issue a MAP OUT DETAIL statement. After each pageable map statement that writes a detail occurrence, test for DC-PAGE-READY to determine whether a map page has been built.

   ```
   MAP OUT USING MAP01 OUTPUT DATA IS YES
     DETAIL NEW
       ON DC-PAGE-READY PERFORM FIRST-PAGE THRU FIRST-PAGE-XIT.
   .
   .
   .
   ```

3. If you do find DC-PAGE-READY, you can optionally:
   - Modify the map.
   - Define messages to display on mapout.

   If you do not find DC-PAGE-READY, perform the IDMS-STATUS routine.
4. Manually map out the first page:

   ```
   FIRST-PAGE.
    MAP OUT USING MAP01 OUTPUT DATA IS YES
      RESUME PAGE FIRST
   ```

If You Never Find DC-PAGE-READY

If, after all detail occurrences have been created, you have not received a DC-PAGE-READY status code, you should transmit the map page to the terminal screen by issuing a MAP OUT RESUME statement.

Ending the Paging Session

The next task specified in the DC RETURN NEXT TASK CODE statement should include logic to test whether the user has indicated the end of the map paging session. If so, issue an ENDPAGE SESSION statement.

Example of Suppressing Automatic Mapout

The following application does not automatically display the first page after it has been built.

```
    OBTAIN CALC DEPARTMENT

     ON DB-REC-NOT-FOUND GO TO NO-DEPT.
    IF DEPT-EMPLOYEE IS EMPTY
       GO TO NO-EMP.
    MOVE DEPT-ID-0410 TO WORK-DEPT-ID.
    STARTPAGE SESSION DCTEST01
      NOWAIT
      BACKPAGE
      BROWSE
      NOAUTODISPLAY.
    PERFORM A100-GET-EMPLOYEES THRU A100-EXIT
             UNTIL DB-END-OF-SET.
    FINISH.
    IF LESS-THAN-A-PAGE
      MAP OUT USING DCTEST01 RESUME.
    DC RETURN NEXT TASK CODE 'DEPTEND'.
  A100-GET-EMPLOYEES.
    OBTAIN NEXT WITHIN DEPT-EMPLOYEE
      ON DB-END-OF-SET GO TO A100-EXIT.
    MOVE EMP-ID-0415 TO WORK-EMP-ID.
    MOVE EMP-LAST-NAME-0415 TO WORK-LAST.
    MOVE EMP-FIRST-NAME-0415 TO WORK-FIRST.
    MAP OUT USING DCTEST01 OUTPUT DATA IS YES
         DETAIL NEW
    ON ANY-STATUS
      NEXT SENTENCE.
    IF DC-PAGE-READY
        PERFORM A100-FIRST-PAGE THRU
            A100-FIRST-PAGE-EXIT
      ELSE PERFORM IDMS-STATUS.
  A100-EXIT.
    EXIT.
  A100-FIRST-PAGE.
```

```
    MOVE 'Y' TO FIRST-PAGE-SW.
    IF ALREADY-MAPPED-OUT
      GO TO A100-FIRST-PAGE-EXIT
    ELSE
      MOVE EMP-MESSAGE-01 TO MESSAGE-01
      MAP OUT USING DCTEST01 OUTPUT DATA IS YES
      RESUME PAGE FIRST
A100-FIRST-PAGE-EXIT.
    EXIT.
```

Line Mode

Line mode supports line-by-line transfers of data to and from a terminal buffer. Line mode transfers are recommended for programs requiring a simple transfer of unformatted data, independent of terminal type. Line mode supports synchronous read and write operations and asynchronous write operations.

Note: While a line mode I/O session is in progress, only line-mode requests can be issued; basic mode and mapping mode requests ca CAuse unpredictable results.

By using line mode terminal management statements, you can:

- Initiate a line mode I/O session
- Write a line of data
- Read a line of data from the terminal screen
- End a line mode session

Beginning a Line Mode Session

You initiate a line mode I/O session by issuing either of the following line mode DML statements:

- WRITE LINE TO TERMINAL
- READ LINE FROM TERMINAL

Writing a Line of Data

To transfer data from program variable storage to the screen, issue a WRITE LINE TO TERMINAL statement. DC automatically inserts the appropriate device control characters.

Transmission of Data Stream

WRITE LINE TO TERMINAL transmits a data stream to the terminal, as follows:

- For **line-by-line devices**, DC writes each line to the terminal immediately after the program issues the WRITE LINE TO TERMINAL request. New lines are added to lines already on the screen until the screen becomes full or the program requests DC to begin a new page.
- For **3270-type devices**, DC collects the number of output lines in buffers (or pages) that correspond to the terminal model in use. Data is written to the screen when:
  - The buffer becomes full
  - A READ LINE FROM TERMINAL request is issued
  - A WRITE LINE FROM TERMINAL request that specifies the NEWPAGE option is issued
  - The issuing task terminates

Formatting the Line

With either device type, data passed with each WRITE LINE TO TERMINAL request begins in the first character position of the next available line on the screen. If the length of the data exceeds the width of the screen, DC automatically reformats data into lines of the appropriate width.

Example of WRITE LINE TO TERMINAL

The figure below shows the processing associated with WRITE LINE TO TERMINAL requests for 3270-type terminals.

When the program issues the WRITE LINE TO TERMINAL NEWPAGE request, DC writes all buffered lines to the terminal. Because the data in line 3 exceeds the width of the screen, it is displayed as two lines.

Sequence of WRITE LINE request	Request	Buffer contents	3270-type terminal screen
1	WRITE LINE TO TERMINAL.	1	
2	WRITE LINE TO TERMINAL.	1 2	
3	WRITE LINE TO TERMINAL.	1 2 3	
4	WRITE LINE TO TERMINAL NEWPAGE LENGTH 0.		1 2 3

Displaying Header Lines

If you want to display header lines that will appear on the terminal, include the HEADER option of the WRITE LINE TO TERMINAL statement. This header will be displayed until a subsequent WRITE LINE TO TERMINAL request modifies or deletes it.

You can display a maximum of three header lines; each line can be a maximum of two physical terminal lines in length. Headers are cleared at the end of each line I/O session.

Reading a Line of Data

To transfer data from the terminal buffer to program variable storage, issue a READ LINE FROM TERMINAL statement. READ LINE FROM TERMINAL transfers data to your program as follows:

- For **line-by-line devices**, DC treats the entire screen contents as a single data field; a READ LINE FROM TERMINAL request returns all data to the program at once.
- For **3270-type devices**, a READ LINE FROM TERMINAL request returns the first data field on the screen marked for input. DC queues remaining data fields marked for input and passes them back to the program one at a time.

Uses of READ LINE FROM TERMINAL

Typical uses of the READ LINE FROM TERMINAL function are:

- **To retrieve any information entered in addition to a task code**. That is, for tasks assigned the INPUT attribute, the user can enter data following the task code. For example, if the user enters:

  GETEMP HENDON

  DC replaces the task code (GETEMP) with leading blanks and returns the data (HENDON) to the program.

- **To read the one-line response to a WRITE LINE TO TERMINAL request that has prompted the terminal operator for information**. On non-3270 devices, when a READ LINE FROM TERMINAL request is issued after one or more WRITE LINE TO TERMINAL requests, DC writes a question mark (?) to the terminal to indicate that a response is required.
- **To enable a program to read formatted 3270-type data fields sequentially**. The first READ LINE FROM TERMINAL request returns the first field on the screen that is marked for input. Subsequent READ LINE FROM TERMINAL requests return the remaining fields to the program, one at a time, as illustrated in the example below.

Example of READ LINE FROM TERMINAL

In the figure below, the first READ LINE FROM TERMINAL request returns the value in the first data field; subsequent READ LINE FROM TERMINAL requests return the values in the remaining data fields in the order in which they appear on the screen.

Sequence of READ LINE request	Returned data	Screen contents
1	KING	LAST NAME: KING
2	DORIS	FIRST NAME: DORIS
3	0106	ID: 0106 STATUS: 04
4	04	DATE: 040886
5	040866	

Ending a Line Mode Session

A line mode I/O session ends when one of the following events occurs:

- The task terminates without issuing a DC RETURN request. Programs that specify the NEXT TASK CODE parameter in a DC RETURN request can extend the line I/O session to include data transfers initiated by the next task.
- The user presses one of the following keys:
  - Clear-3270 terminals
  - Attn- 2741 terminals
  - Break- Teletype terminals
- The program issues an END LINE TERMINAL SESSION request.

  Following an END LINE TERMINAL request, DC does not automatically display lines that remain in a partially filled buffer; typically, this data is of no use to the user. However, to display the contents of a partially filled buffer before ending the line I/O session, your program can issue a WRITE LINE TO TERMINAL request that specifies the NEWPAGE option and a dummy data line (that is, one with a length of zero).

3270-type Considerations

The following special considerations apply to 3270-type devices:

- DC assigns each page of data in the line I/O session a sequential number starting with 1; page numbers are displayed at the bottom of the screen.
- DC keeps all pages associated with a line I/O session in a scratch area unless otherwise requested. When the I/O session terminates, these pages remain in the scratch area where they can be viewed by the user and subsequently deleted. At any point, the user can display any page either by its position relative to the currently displayed page or by page number:
  - **Next page** - Press PA1
  - **Previous page** - Press PA2 or the CANC key.
  - **Specific page** - Enter the desired page number following the words NEXT PAGE at the bottom of the screen and press Enter.

Unless the NOBACKPAGE option has been specified in a READ LINE FROM TERMINAL or WRITE LINE TO TERMINAL request, all pages processed during the I/O session remain available until the user signals completion of their use by pressing Enter with no request to see another page. If the page displayed is the last page of the session, DC deletes all pages associated with the current session, clears page header lines, and resets the current page number to one (1).

For further details regarding line mode DML statements, see the language-specific *CA IDMS* DML *Reference Guide*.

Chapter 9: Storage, Scratch, and Queue Management

Pseudoconversational programming demands techniques that efficiently pass data from one task to another. You should choose the method or combination of methods best suited to the needs of your application. You can choose different methods based on the following considerations:

- Length of time that the data is needed
- Availability of the data to other users
- Data recoverability
- System resources used
- Network resources used
- Number of variables

CA IDMS provides these services for managing online variable storage:

- **Storage pools** manage short-term variable-storage resources and pass data from one task to another
- **Scratch records** pass temporary data between tasks running on the same logical terminal
- **Queue records** pass more permanent data from one task to another
- The **terminal buffer** passes very small amounts of data between tasks running on the same logical terminal.

This section contains the following topics:

Using Storage Pools (see page 229)
Using Scratch Records (see page 241)
Using Queue Records (see page 249)
Using the Terminal Screen To Transmit Data (see page 253)

Using Storage Pools

To facilitate online programming and intertask communication, CA IDMS provides storage management functions that allow you to acquire space explicitly in storage pools.

These functions control allocation of variable storage in a CA IDMS storage pool or work area. Shared by system and user programs, the storage pool also contains space for buffers and initial storage areas (ISAs) used by Assembler and PL/I programs.

Note: All variable-storage entries (except COBOL LINKAGE SECTION and PL/I BASED storage entries) defined by your program are acquired automatically from the CA IDMS storage pool when the program starts and released automatically when the program ends.

Using CA IDMS storage management functions, you can:

- Acquire variable storage from a storage pool
- Establish addressability to previously acquired variable storage
- Release all or part of previously acquired variable storage

Types of Acquired Storage

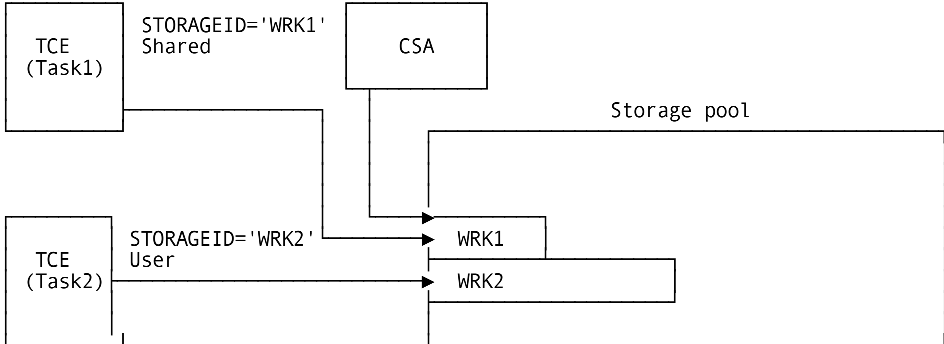
You must specify whether the acquired storage is available to other users:

- **User** storage is available only to the issuing task; no other tasks can access it. CA IDMS maintains user storage through the issuing task's task control element (TCE).
- **Shared** storage is available to all tasks running under the CA IDMS system. CA IDMS links shared storage to the common system area (CSA) as well as to the TCE, as illustrated in the figure below. CA IDMS uses the CSA to locate the address of a shared area to satisfy requests from other tasks for shared storage.

  Note: Shared storage is available to all tasks within the CA IDMS system; however, each task must explicitly establish addressability to access such storage.

TCE and CSA Ownership

Shared storage is linked to both the TCE and CSA; user storage is linked only to the TCE, as the figure below shows.



Kept Storage

If you require that storage remain allocated after a task ends, it should be assigned the **KEEP** attribute when it is initially allocated. Kept storage is associated with the logical terminal on which the task is executing and with the task itself; such storage can be released only through a program request.

Releasing Storage

When storage is explicitly released or a task terminates, CA IDMS releases linkage to the TCE.

For a quick reference of storage release procedures and conditions, see Storage Pool Summary.

'User storage only'. You can explicitly release all or a part of user storage. For a partial release, TCE linkage and the KEEP attribute remain unaffected.

User Storage

User storage is associated exclusively with the issuing task through the TCE; when the task terminates, user storage is released. By dynamically acquiring only the amount of storage needed, you can make more effective use of storage resources.

Steps to Acquire User Storage

To dynamically acquire and use variable storage from the storage pool within a single task, perform the following steps:

1. Acquire variable storage from the storage pool by issuing a GET STORAGE statement that specifies the USER parameter.
2. Check for an ERROR-STATUS of 3210 (DC-NEW-STORAGE).
3. Perform the IDMS-STATUS routine if 3210 is not returned.
4. Perform processing, using the acquired storage as needed.
5. Release the acquired storage by issuing a FREE STORAGE statement that specifies the appropriate storage ID.

Example of Acquiring User Storage

The program excerpt below shows the acquisition and release of user storage.

The program acquires the minimum amount of storage needed to complete the processing specified by the user.

```
DATA DIVISION.
LINKAGE SECTION.
01  COPY IDMS RECORD EMPLOYEE.
   05 EMPLOYEE-END      PIC X.
01  COPY IDMS RECORD DEPARTMENT.
   05 DEPARTMENT-END    PIC X.
01  ERROR-DATA.
   05 ERROR-DEPT-ID      PIC 9(4).
   05 ERROR-MESSAGE-CODE  PIC X(4).
   05 ERROR-DATA-END     PIC X.
PROCEDURE DIVISION.
MAIN-LINE.
*** THIS PROGRAM ACQUIRES STORAGE FOR EITHER THE ***
*** DEPARTMENT RECORD OR THE EMPLOYEE RECORD     ***
*** DEPENDING ON THE CONTROL KEY PRESSED BY THE  ***
*** TERMINAL OPERATOR.                          ***
   BIND MAP SOLICIT.
   BIND MAP SOLICIT RECORD SOLICIT-REC.
   MAP IN USING SOLICIT.
   INQUIRE MAP SOLICIT MOVE AID TO DC-AID-IND-V.
   IF CLEAR-HIT DC RETURN
   ELSE
     IF PA01-HIT GO TO A100-GET-EMPLOYEE
   ELSE
     IF PA02-HIT GO TO A100-GET-DEPARTMENT
   ELSE
     GO TO U100-ERROR-PROC.
*
A100-GET EMPLOYEE.
   IF SOLICIT-EMP-ID NOT NUMERIC
     GO TO U200-ERROR-EMP-ID.
*** ACQUIRE USER STORAGE FOR THE EMPLOYEE RECORD ***
   GET STORAGE FOR EMPLOYEE TO
           EMPLOYEE-END
     NOWAIT SHORT USER
     STGID 'EMPL' VALUE IS LOW-VALUE
   ON DC-NEW-STORAGE
     NEXT SENTENCE.
   MOVE SOLICIT-EMP-ID TO EMP-ID-0415.
   OBTAIN CALC EMPLOYEE
    ON DB-REC-NOT-FOUND
      GO TO U200-ERROR-NO-EMP.
        .
```

User Kept Storage

User kept storage is available to all tasks running on a logical terminal until a task associated with that terminal releases the storage. User kept storage is ideal for passing small amounts of information between tasks. CA IDMS maintains TCE linkage for user kept storage across tasks by using the logical terminal element (LTE). When a new task is initiated from the same terminal, CA IDMS transfers this linkage from the LTE to the TCE of the new task.

Steps to Acquire User Kept Storage

To dynamically acquire and use variable storage from the storage pool and make the storage available to multiple tasks running on the same logical terminal:

1. Acquire variable storage from the storage pool by issuing a GET STORAGE statement that specifies both the USER and the KEEP parameters.

   Note: You can indicate that storage is eligible for allocation above the 16Mb line by specifying LOCATION IS ANY on the GET STORAGE statement.

2. Check for an ERROR-STATUS of 3210 (DC-NEW-STORAGE).
3. Perform the IDMS-STATUS routine if 3210 is not returned.
4. Perform processing, using the acquired storage as needed.
5. Issue a DC RETURN statement, optionally specifying the next task to be invoked.

Accessing User Kept Storage

In subsequent tasks invoked on the same logical terminal:

1. Establish addressability to the previously acquired storage by issuing a GET STORAGE request that names the storage ID specified for the storage area when it was first allocated.
2. Perform processing, using the acquired data.

You should release the acquired storage as soon possible by issuing a FREE STORAGE statement that specifies the appropriate storage ID.

Example of Acquiring User Kept Storage

The program excerpt below shows the initial assignment of user kept storage.

The program performs preliminary error checking before transferring control to a database retrieval program.

```
DATA DIVISION.
01  TRANSPROG           PIC X(8)   VALUE 'DEPTGET'.
01  SOLICIT-REC.

    05 SOLICIT-DEPT-ID    PIC X(4).
LINKAGE SECTION.
01  PASS-DEPT-INFO.
    05 PASS-DEPT-ID       PIC 9(4).
    05 PASS-DEPT-INFO-END  PIC X.

PROCEDURE DIVISION.
    BIND MAP SOLICIT.
    BIND MAP SOLICIT RECORD SOLICIT-REC.
*
    MAP IN USING SOLICIT.
    INQUIRE MAP SOLICIT MOVE AID TO DC-AID-IND-V.
    IF CLEAR-HIT DC RETURN.
*
    IF SOLICIT-DEPT-ID NOT NUMERIC
      GO TO ERROR-DEPT-ID.
*** ACQUIRE USER KEPT STORAGE ***
    GET STORAGE FOR PASS-DEPT-INFO
            TO PASS-DEPT-INFO-END
       NOWAIT KEEP LONG USER
       STGID 'DEPT' VALUE IS LOW-VALUE
    ON DC-NEW-STORAGE
       NEXT SENTENCE.
*** MOVE MAP DATA TO FIELDS IN ACQUIRED STORAGE ***
    MOVE SOLICIT-DEPT-ID TO PASS-DEPT-ID.
*** TRANSFER CONTROL TO DATABASE ACCESS PROGRAM ***
    TRANSFER CONTROL TO TRANSPROG
      NORETURN.
```

Reestablishing Addressability to User Kept Storage

The program excerpt below establishes addressability to the previously acquired storage and releases it. The program uses data from the previously acquired storage to perform database access.

```
DATA DIVISION.
01  NTCODES.
    05 NEXT-TASK          PIC X(8)   VALUE 'DEPTMOD'.
01  MESSAGES.
    05 DEPT-DISPLAY-MESS     PIC X(20)
      VALUE 'DEPARTMENT DISPLAYED'
```

```
   05 DEPT-DISPLAY-MESS-END. PIC X.
 01  SOLICIT-REC.
   05 SOLICIT-DEPT-ID       PIC X(4).
 LINKAGE SECTION.
 01  PASS-DEPT-INFO.
   05 PASS-DEPT-ID       PIC 9(4).
   05 PASS-DEPT-INFO-END  PIC X.

 PROCEDURE DIVISION.
*** ESTABLISH ADDRESSABILITY TO PREVIOUSLY ACQUIRED STORAGE ***
   GET STORAGE FOR PASS-DEPT-INFO
          TO PASS-DEPT-INFO-END
     NOWAIT KEEP LONG USER
     STGID 'DEPT'.
   BIND MAP SOLICIT.
   BIND MAP SOLICIT RECORD SOLICIT-REC.
*** MOVE DATA TO DATABASE CALC-KEY AND MAP DATA FIELD ***
   MOVE PASS-DEPT-ID TO DEPT-ID-0410.
   MOVE PASS-DEPT-ID TO SOLICIT-DEPT-ID.
*** RELEASE STORAGE ***
   FREE STORAGE STGID 'DEPT'.
        .
*** DATABASE ACCESS ***
        .
   MAP OUT USING SOLICIT
     MESSAGE IS DEPT-DISPLAY-MESS TO DEPT-DISPLAY-MESS-END.
   DC RETURN NEXT TASK CODE NEXT-TASK.
```

Shared Storage

Shared storage is available to all tasks running concurrently under the CA IDMS system.

Shared storage is usually accessed by a concurrent nonterminal task. For example, such a nonterminal task might support the main task by performing print functions.

For more information on nonterminal tasks, see Initiating Nonterminal Tasks.

When Shared Storage Is Released

CA IDMS maintains an in-use counter for each area of shared storage. Each time a task establishes addressability to an area of shared storage, CA IDMS adds 1 to the in-use counter. When a task terminates or releases the storage, CA IDMS subtracts 1 from the in-use counter. CA IDMS releases shared storage when the in-use counter is set to zero.

Steps to Acquire Shared Storage

To dynamically acquire and use variable storage from the storage pool and make the storage available to other tasks running under the same CA IDMS system, perform the following steps:

1. Acquire variable storage from the storage pool by issuing a GET STORAGE statement that specifies the SHARED parameter.
2. Check for an ERROR-STATUS of 3210 (DC-NEW-STORAGE).
3. Perform the IDMS-STATUS routine if 3210 is not returned.
4. Perform processing, using the acquired storage as needed.
5. Optionally, release the shared storage by issuing a FREE STORAGE statement that specifies the appropriate storage ID.

Steps to Access Shared Storage

To access the data from another task executing concurrently under the same CA IDMS system, perform the following steps:

1. Establish addressability to the previously acquired storage by issuing a GET STORAGE request that names the storage ID specified for the storage area when it was first allocated.
2. Perform processing using the acquired data.
3. Optionally, release the shared storage by issuing a FREE STORAGE statement that specifies the appropriate storage ID.

Shared Kept Storage

Shared kept storage is available to all tasks running under the CA IDMS system. Once a storage area with the SHARED KEEP attribute is established, any task running under the CA IDMS system can access that area.

When Shared Kept Storage Is Released

CA IDMS maintains an in-use counter and a keep flag for each area of shared kept storage. Shared kept storage is released only when *both* of the following conditions are true:

1. The in-use counter is set to zero, indicating that there are no current users of the area.
2. The keep flag is turned off (the FREE STORAGE statement turns the keep flag off)

If either condition is false, the storage area remains allocated. With this feature, shared kept storage areas remain allocated even when they are not being used, provided the keep flag remains on.

Each time a task establishes addressability to an area of shared kept storage, CA IDMS adds 1 to the in-use counter. When the task terminates, CA IDMS subtracts 1 from the in-use counter. When a program issues a FREE STORAGE request, CA IDMS subtracts 1 from the in-use counter *and* turns off the keep flag. Once a FREE STORAGE request is issued, if the in-use counter is zero, CA IDMS releases the storage. Once turned off, the keep flag cannot be reset.

Difference from User Kept Storage

Unlike user kept storage, shared kept storage is not linked to the LTE across tasks executing on the same terminal.

Startup Shared Kept Storage

One shared storage area having the keep attribute is allocated at system startup for use by all tasks; this storage area is never freed. This area is called the common work area (CWA) and can contain application-defined information, if so requested during system generation.

For more information about the CWA, see *CA IDMS System Generation Guide*.

Example of Shared Kept Storage

The program excerpt below shows programmatic access to data previously placed in the CWA.

This program accesses the CWA in order to obtain the current date in Gregorian format:

```
DATA DIVISION.
01  NTCODES.

    05 NEXT-TASK          PIC X(8)   VALUE 'DEPTGET'.
01  CWA              PIC X(4)   VALUE 'CWA'.
01  SOLICIT-REC.
    05 SOLICIT-DEPT-ID     PIC X(4).
    05 SOLICIT-GREG-DATE   PIC X(8).
LINKAGE SECTION.
01  CWA-DATA.
    05 CWA-DATE           PIC X(8).
    05 CWA-DATA-END        PIC X.
```

```
PROCEDURE DIVISION.
    BIND MAP SOLICIT.
    BIND MAP SOLICIT RECORD SOLICIT-REC.
*** GET THE DATE IN GREGORIAN FORMAT FROM THE CWA ***
    GET STORAGE FOR CWA-DATA TO CWA-DATA-END
        NOWAIT KEEP SHORT SHARED
        STGID CWA.
    MOVE ZEROS TO SOLICIT-DEPT-ID.
    MOVE CWA-DATE TO SOLICIT-GREG-DATE.
    MAP OUT USING SOLICIT.
    DC RETURN NEXT TASK CODE NEXT-TASK.
```

Storage Pool Summary

Acquired storage is associated with the TCE, the CSA, or both. Additionally, user storage with the keep attribute is linked to the LTE.

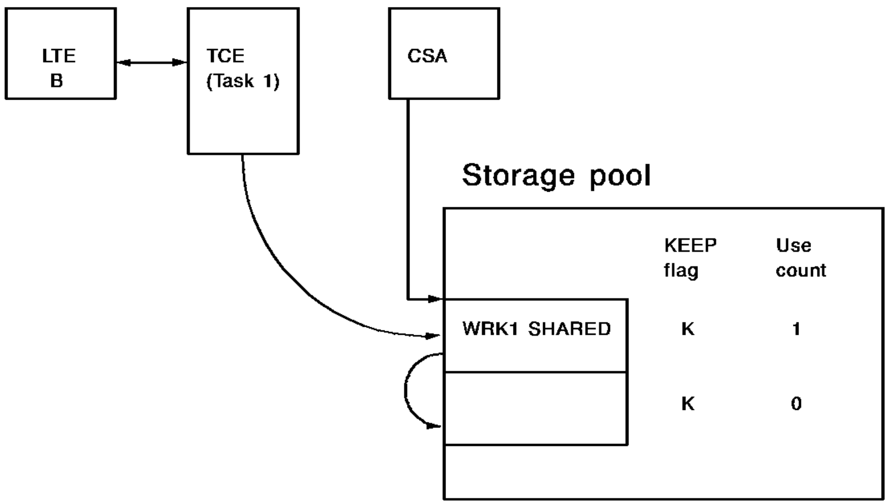
The table below shows the procedures and conditions under which CA IDMS maintains linkage when storage is released. This table assumes that the FREE STORAGE request releases the entire storage area.

Storage Attribute	After FREE STORAGE Request	After Task Termination
USER	Storage is released.	Storage is released.
USER KEEP	Storage is released.	Storage remains allocated; TCE linkage is transferred to the LTE.
SHARED	Storage is released only if the in-use counter is set to zero.	Storage is released only if the in-use counter is set to zero.
SHARED KEEP	Storage is released only if the in-use counter is set to zero.	Storage remains allocated.

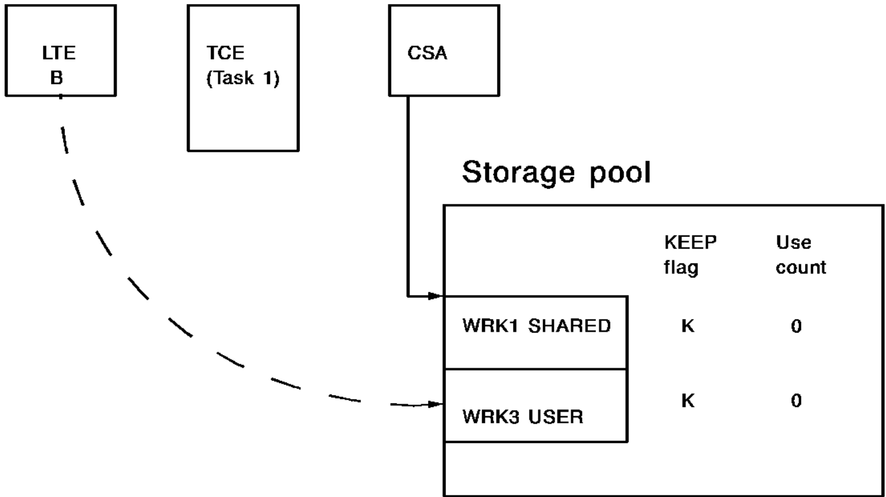
How Storage Is Allocated and Released

The following diagrams illustrate how CA IDMS allocates and releases storage.

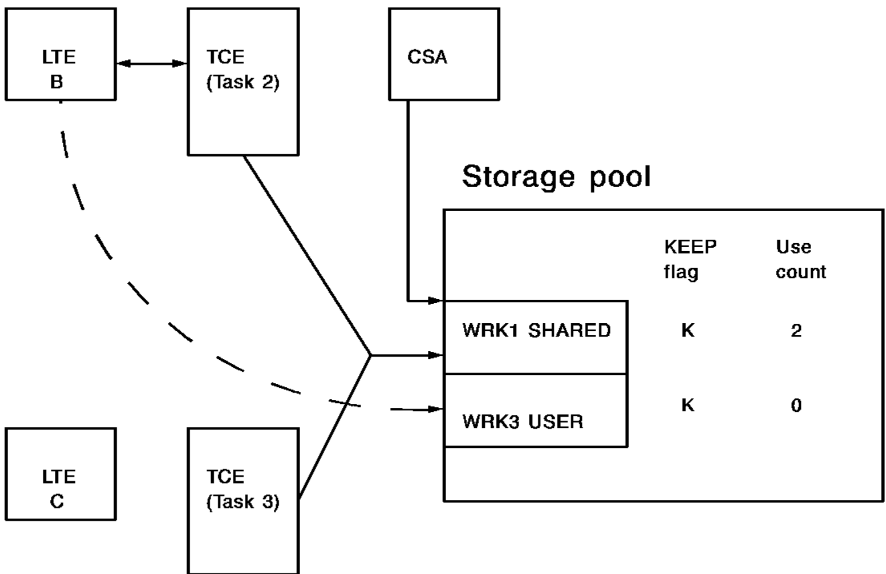
1. Task 1, running on terminal B, establishes addressability to two variable areas of kept storage. WRK1 is designated shared keep; WRK3 is designated user keep. Because task 1 is the only task using WRK1, the in-use counter associated with WRK1 is set to 1.



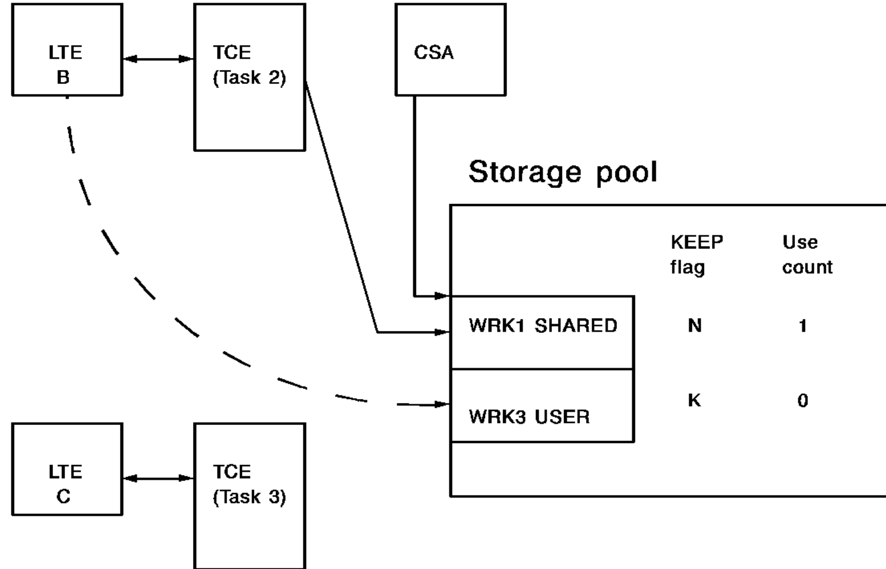
2. Task 1 terminates without issuing a FREE STORAGE request for either WRK1 or WRK3. CA IDMS automatically decrements the in-use counter and transfers linkage for WRK3 to the LTE for terminal B. Because WRK1 is shared, CA IDMS does not maintain linkage to the LTE. Although WRK1 has no users, it remains allocated because an explicit FREE STORAGE was not issued.



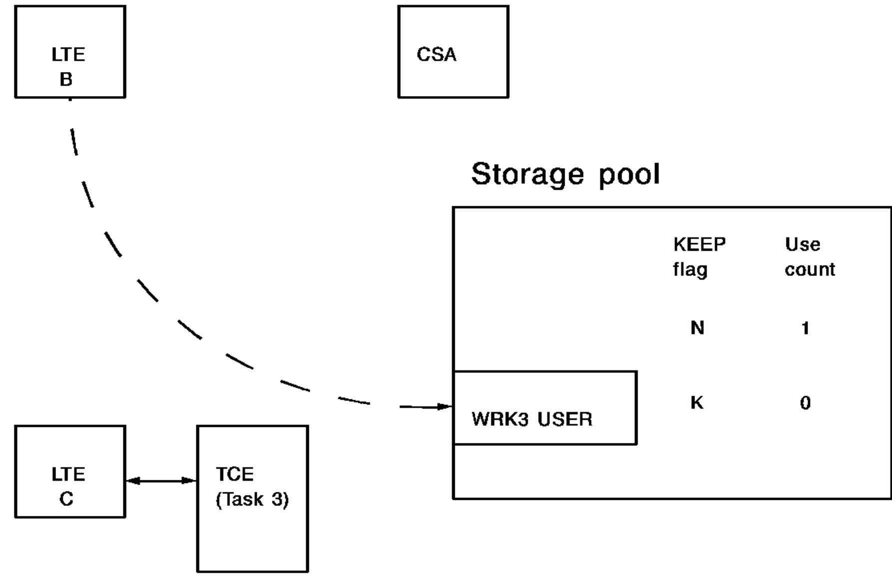
3. Task 2 is initiated on terminal B and issues a GET STORAGE request for WRK1. Task 3 is initiated on terminal C and issues a GET STORAGE request for WRK1. The in-use counter for WRK1 indicates two users.



4. Task 3 issues a FREE STORAGE request for WRK1; CA IDMS turns the keep flag off and decrements the in-use counter by 1.



5. Task 2 terminates without issuing a FREE STORAGE request for WRK1; CA IDMS decrements the in-use counter by 1. Because the keep flag is off and the in-use counter is set to zero, CA IDMS releases the storage associated with WRK1.



Using Scratch Records

CA IDMS scratch management functions allow you to allocate, retrieve, and delete scratch records. Scratch records, which are stored in the DDLDCSCR area of the dictionary, are used to pass data from one task to subsequent tasks running on the same terminal. These records are not accessible to tasks executing on other terminals.

Fast Access

Scratch records provide fast access because:

- **Scratch records are indexed**. They are stored in an indexed set in the DDLDCSR area of the dictionary.
- **The DDLDCSCR area provides efficient access**. It is initialized at system startup; any previously existing records are deleted.
- **Scratch records are unavailable to other users**. You do not have to wait for record locks to be freed.

Best Use of Scratch Records

Scratch records are not recoverable across a shutdown/startup or a system crash. All scratch records are deleted at system startup. Because they are not saved across a system shutdown, scratch records are best used for temporary storage of data.

Availability to a Subsequent Task

When a task terminates, CA IDMS temporarily associates that task's scratch areas with the logical terminal from which the task was invoked. This is done using the logical terminal element (LTE). When a new task is initiated on the same terminal, CA IDMS transfers the scratch areas to the task control element (TCE) for the new task. All scratch records and currencies associated with the old task are available to the new task.

What You Can Do with Scratch Records

You can use CA IDMS scratch management functions to do the following:

- Store or replace a scratch record in the dictionary
- Retrieve a scratch record from the dictionary and place it in a variable-storage area associated with the issuing task
- Delete a scratch record from the dictionary

Steps to Allocate or Replace a Scratch Record

To allocate or replace a scratch record, perform the following steps:

1. Initialize the appropriate fields in program variable storage.
2. Issue a PUT SCRATCH command that specifies the variable-storage location of the data to be stored; to replace a record, include the REPLACE parameter.
3. If you specify the REPLACE parameter, check for a status of 4317 (DC-REC-REPLACED).
4. Perform the IDMS-STATUS routine. (If you specify REPLACE, perform this step only if 4317 is not returned.)

Scratch Area

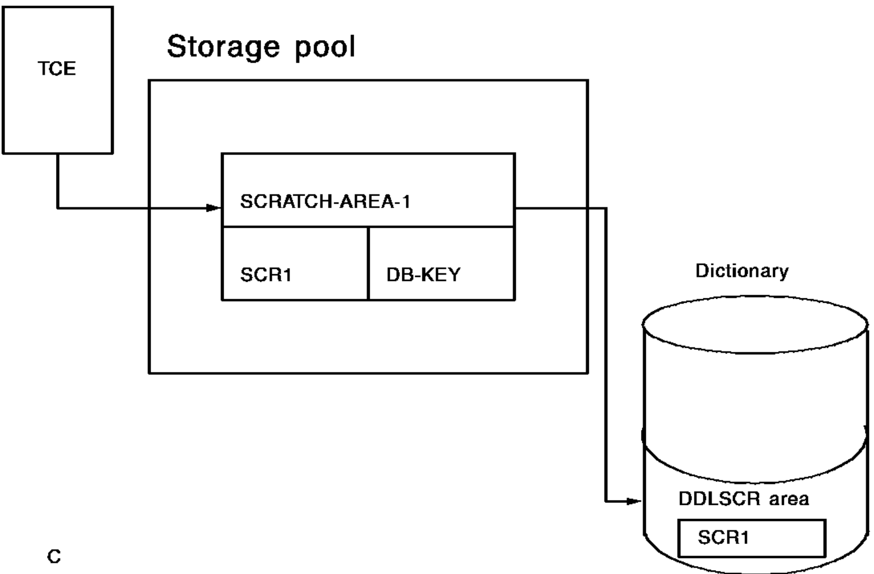
In response to your PUT SCRATCH request, CA IDMS places the scratch record in the DDLDCSCR area of the dictionary. An index pointer to the record is placed in a storage pool scratch area. Each scratch area is identified by its area ID; scratch records in each area are indexed in ascending order by scratch record ID (SRID).

Typically, your program assigns the SRID. If not, CA IDMS assigns the SRID, places the record last within the scratch area, and returns the SRID to your program.

Any number of scratch areas can be associated with a task and any number of scratch records can be associated with a scratch area.

Example of Scratch Record Allocation

The figure below shows scratch record allocation. When a PUT SCRATCH request is issued, CA IDMS creates a scratch record in the dictionary and places a pointer to that record in a scratch area associated with the issuing task.



Steps to Retrieve a Scratch Record

To retrieve a scratch record, perform the following steps:

1. Issue a GET SCRATCH command that specifies the appropriate scratch area ID and indicates the variable-storage location in which the scratch record is to be placed. You can retrieve scratch records by position within the area, by relationship to the current record of the scratch area, or by SRID.

2. If you are issuing the GET SCRATCH command iteratively and specifying the DELETE parameter, check for a status of 4303 (DC-AREA-ID-UNK); this indicates the end of the scratch area. If you specify KEEP, check for a status of 4305 (DC-REC-NOT-FOUND); this indicates the end of the scratch area.

   If there is any chance that the length of the retrieved record exceeds the length of its allocated variable storage, you should do the following:

   Include the KEEP parameter of the GET SCRATCH statement to ensure that data is not deleted when it is retrieved.

   - Check for a status of 4319 (DC-TRUNCATED-DATA).

3. Perform the IDMS-STATUS routine if neither 4303, 4305, nor 4319 is returned.

'Scratch record currency'. CA IDMS maintains currency for the records in each scratch area. Because CA IDMS maintains currency across tasks, you should be aware that the NEXT option does not default to FIRST, and PRIOR does not default to LAST.

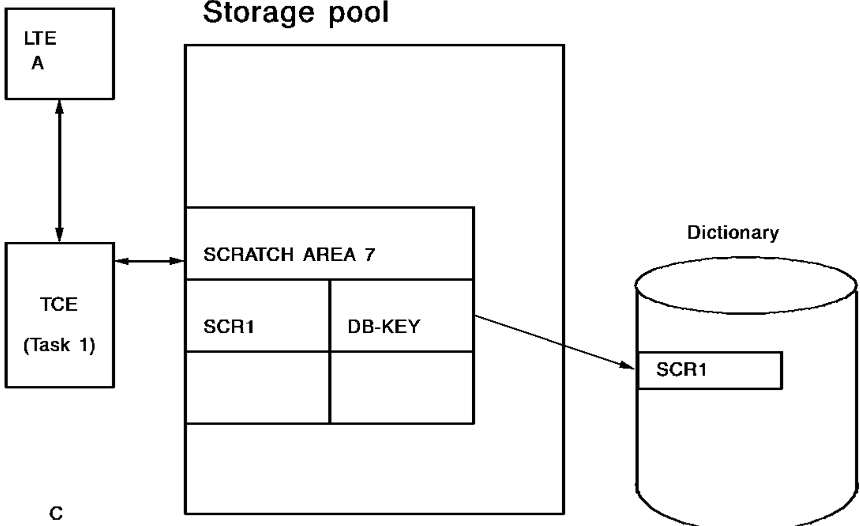
Steps to Delete a Scratch Record

To delete a scratch record, issue either of the following commands:

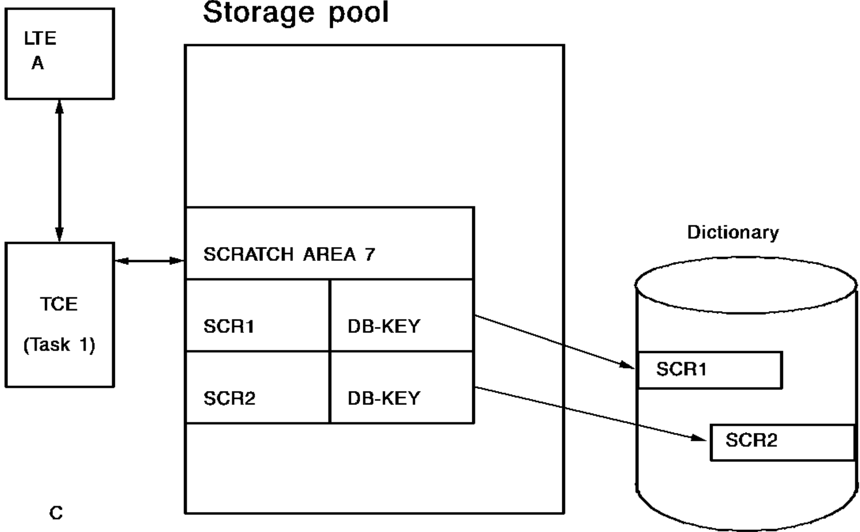
- A GET SCRATCH command that specifies the DELETE parameter. CA IDMS copies the scratch record to the appropriate variable-storage area and deletes the record. When all scratch records associated with a given scratch area have been deleted, CA IDMS deletes the scratch area. CA IDMS returns a status of 4303 (DC-AREA-ID-UNK) to later GET SCRATCH requests that specify that area ID.
- A DELETE SCRATCH command that specifies one of the following:
  - That a particular occurrence of the scratch record is to be erased
  - That the entire scratch area should be erased
  - **Allocating Scratch Records Across Tasks**

The following diagrams illustrate how CA IDMS dynamically allocates scratch records across tasks:

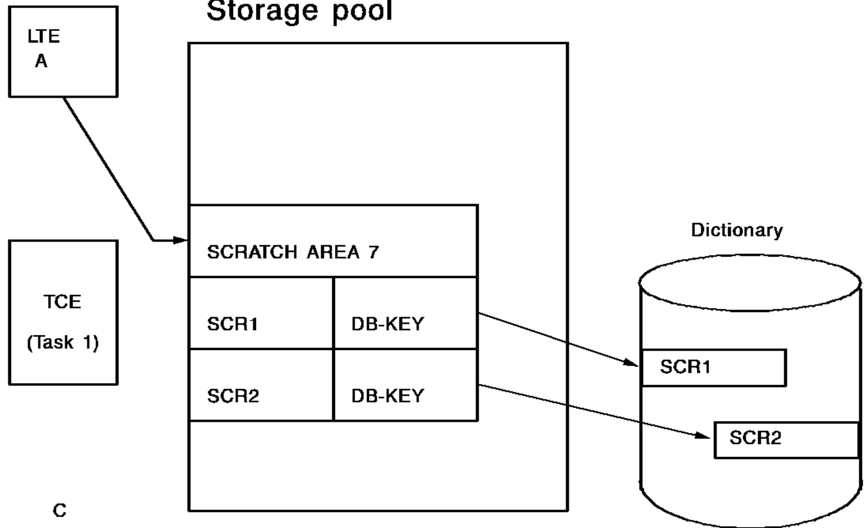
1. Task 1 stores scratch record SCR1 in scratch area 7. Because no scratch area with that identifier exists for task 1, CA IDMS dynamically allocates the area within the variable-storage pool. A scratch record is placed in the dictionary and is associated with task 1's TCE.



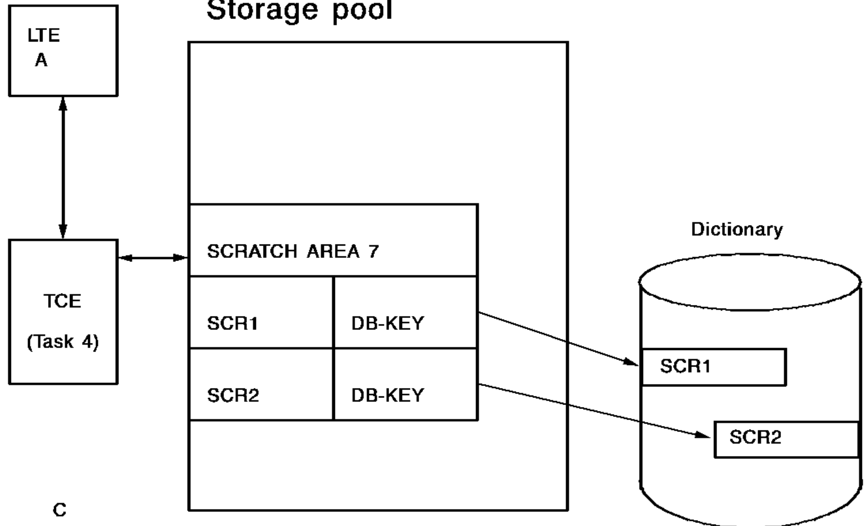
2. Task 1 stores SCR2 in scratch area 7. CA IDMS creates a second entry in scratch area 7 and places the new record in the dictionary.



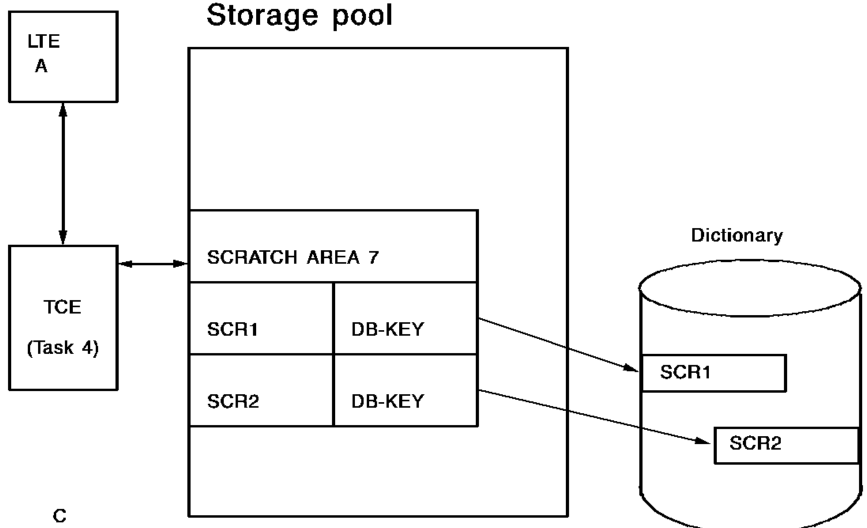
3. Task 1 terminates. CA IDMS associates scratch area 7 with the LTE for terminal A. Scratch area 7 is no longer associated with task 1.



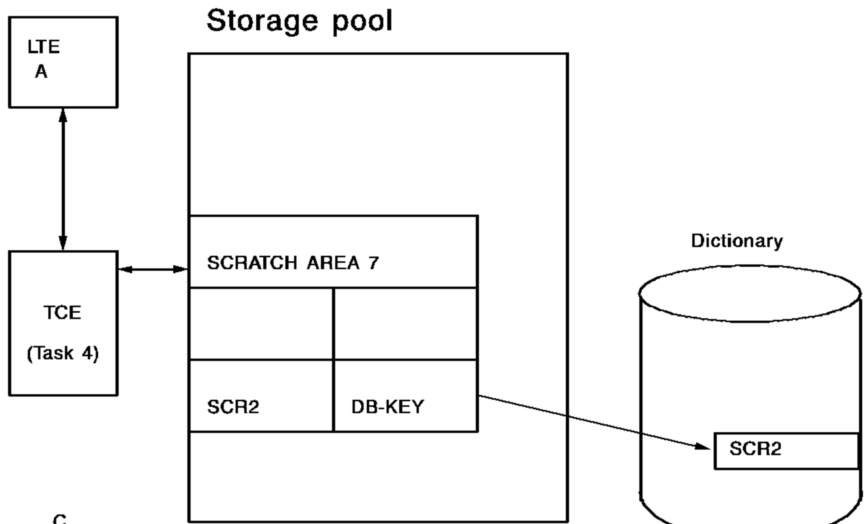
4. Task 4 is initiated on terminal A. CA IDMS associates scratch area 7 with task 4's TCE.



5. Task 4 issues a GET SCRATCH to obtain SCR2. Data associated with scratch record SCR2 now resides in variable storage for task 4, as well as in the dictionary.



6. Task 4 deletes SCR1. CA IDMS deletes the scratch area entry for that record and removes the record from the dictionary.



Example of Retrieving Scratch Records

The program excerpt below retrieves scratch records from the TEST-SCRATCH scratch area. The program uses a pageable map in order to display an unlimited number of scratch records.

The program retrieves all occurrences in the TEST-SCRATCH scratch area. Each occurrence contains the employee's ID, last name, and first name.

```
WORKING-STORAGE SECTION.
01  TC                    PIC X(8).
   88 GETOUT              VALUE 'GETSCR2'.
01  SWITCHES.
   05 FIRST-PAGE-SW       PIC X  VALUE 'N'.
     88 LESS-THAN-A-PAGE  VALUE 'N'.
01  GETSCR2               PIC X(8)   VALUE 'GETSCR2'.
01  TESTSCR               PIC X(8)   VALUE 'TESTSCR'.
01  TEST-SCRATCH.
```

```
    05 SCR-ID            PIC 9(4).
    05 SCR-LNAME         PIC X(15).
    05 SCR-FNAME         PIC X(10).
    05 TEST-SCRATCH-END   PIC X.
 01  SCRMAP-REC.
    02  ID               PIC 9(4).
    02  LNAME            PIC X(15).
    02  FNAME            PIC X(10).
 PROCEDURE DIVISION.
 MAIN-LINE.
    ACCEPT TASK CODE INTO TC.
    IF GETOUT ENDPAGE
          DC RETURN.
    BIND MAP SCRMAP01.
    BIND MAP SCRMAP01 RECORD SCRMAP-REC.
    STARTPAGE SESSION SCRMAP01 NOWAIT BACKPAGE BROWSE
      ON DC-SECOND-STARTPAGE NEXT SENTENCE.
*
    GET SCRATCH AREA ID TESTSCR FIRST KEEP
       INTO TEST-SCRATCH TO
           TEST-SCRATCH-END
     ON DC-AREA-ID-UNK
        GO TO ERR-NO-SCR.
    MOVE SCR-ID TO ID.
    MOVE SCR-LNAME TO LNAME.
    MOVE SCR-FNAME TO FNAME.
    MAP OUT USING SCRMAP01
        DETAIL NEW.
    PERFORM A100-GET-SCRATCH THRU A100-EXIT
                 UNTIL DC-REC-NOT-FOUND.
    IF LESS-THAN-A-PAGE
      MAP OUT USING SCRMAP01
         NEWPAGE RESUME.
    DC RETURN NEXT TASK CODE GETSCR2.

 A100-GET-SCRATCH.
    GET SCRATCH AREA ID TESTSCR NEXT KEEP
       INTO TEST-SCRATCH TO
           TEST-SCRATCH-END
      ON DC-REC-NOT-FOUND GO TO A100-EXIT.
    MOVE SCR-ID TO ID.
    MOVE SCR-LNAME TO LNAME.
    MOVE SCR-FNAME TO FNAME.
    MAP OUT USING SCRMAP01
        DETAIL NEW
    ON DC-FIRST-PAGE-SENT
      MOVE 'Y' TO FIRST-PAGE-SW.
 A100-EXIT.
    EXIT.
```

Using Queue Records

CA IDMS queue management functions allow you to store, retrieve, and delete queue records. Queue records, which are stored in the dictionary, are available to all tasks running under CA IDMS and to batch programs with an operating mode of DC-BATCH.

Queue records are saved across a system shutdown/startup and recovered across a system crash; however, currencies are lost when the system crashes or is shut down.

In a data sharing environment, queues can be shared between members of a data sharing group.

Queue Record Storage

CA IDMS stores queue records in the DDLDCRUN area of the dictionary. Each queue record is a member record in a set owned by a queue header record. All records associated with one queue header are referred to collectively as a **queue**. You can direct records to queues defined at system generation, to queues defined through the DDDL compiler, to program-defined queues, or to null queues.

Sharing Queues Between CA IDMS Systems

In a data sharing environment, queues can be shared between CA IDMS systems that are members of a sharing group. The benefit of a shared queue is that it can be read and updated by programs executing on any member of the group. Whether or not a specific queue is shared, is determined by specifications made by the CA IDMS system administrator. Programs accessing queues are not sensitive to whether or not a queue is shared, since the DML syntax is the same in either case.

How You Can Use Queue Management

You can use CA IDMS queue management functions to do the following:

- Store a queue record and assign an ID to uniquely identify the record
- Retrieve a queue record and place it in a variable-storage area associated with the issuing task
- Delete a record from a specified queue
- Delete an entire queue

Steps to Store a Queue Record

To store a queue record, perform the following steps:

1. Initialize the appropriate fields in program variable storage.
2. Issue a PUT QUEUE command that specifies the variable-storage location of the data to be stored.

Steps to Retrieve a Queue Record

To retrieve a queue record, perform the following steps:

1. Issue a GET QUEUE command that specifies the appropriate queue ID and indicates the variable-storage location in which the queue record is to be placed.

   If there is any chance that the length of the retrieved record exceeds the length of its allocated variable storage, you should do the following:

   Include the KEEP parameter of the GET QUEUE statement to ensure that the record is not deleted when it is retrieved.

   - Check for a status of 4419 (DC-TRUNCATED-DATA).

2. Check for a status of 4405 (DC-REC-NOT-FOUND), which indicates that you have retrieved all queue records for the specified queue ID.

3. Perform the IDMS-STATUS routine if neither 4405 nor 4419 is returned.

'Queue record currency'. CA IDMS maintains currency for each queue by task. If several tasks are accessing a queue concurrently, CA IDMS maintains currency separately for each task. Access to a queue record can be by queue ID, by position within the queue, or by relationship of the specified record to the current record of the queue.

Steps to Delete a Queue Record

To delete a queue record, issue either of the following commands:

- A GET QUEUE command that specifies the DELETE parameter. CA IDMS copies the record's data to the appropriate variable-storage area and deletes the record.
- A DELETE QUEUE command that specifies one of the following:
  - That the current occurrence of the queue record is to be erased
  - That the entire queue should be deleted

Implicit Deletion of Queue Records

CA IDMS saves the next and prior currencies following a DELETE QUEUE function so that you can still access the next and prior records in the queue. When all records associated with a given queue have been deleted, CA IDMS deletes the header record as well. Queue records are also deleted implicitly if the associated queue header record is deleted.

Deleting Queues

Queues can also be deleted at system startup or at run time:

- **At system startup** -- Each queue is assigned a retention period; the retention period specifies the number of days that CA IDMS will retain the queue. At system startup, CA IDMS deletes queues that have exceeded their retention periods.
- **At run time** -- The DCMT VARY QUEUE command can be used to delete unwanted queues at run time.

  For more information on DCMT commands, see *CA IDMS System Operations* Guide.

Queue Record Locks

Because queues are shared among tasks, CA IDMS must ensure that two tasks do not update a queue record concurrently, causing unexpected alteration of data. Additionally, if a task terminates abnormally, CA IDMS must ensure that the queue can be restored to its state before the failure. To accomplish this, CA IDMS handles queues in the following manner:

- When a task stores or retrieves a queue record, CA IDMS places an implicit exclusive lock on that record, thereby preventing it from being retrieved or updated by other tasks.
- All records locked by CA IDMS remain locked until the task terminates or until your program issues a COMMIT TASK statement. COMMIT TASK causes some or all of the locks to be released, as specified.
- Queue currencies and locks are not passed from one task to the next on a terminal. Each task is responsible for reestablishing any required currencies.

Avoiding Task Waits for Queue Access

Only one task can access a queue record at a time; other tasks attempting access must wait until the current task is complete. Therefore, you should ensure that queue access is short lived. There should be no long waits, such as pseudoconverses, embedded within queue access code.

Retrieving Queue Records

The program excerpt below retrieves and displays queue records. This program uses a pageable map in order to display an unlimited number of queue records.

The program retrieves all occurrences in the DISPQ queue. This queue lists the employee's ID and last name, and the date and time that each queue record was established.

```
WORKING-STORAGE SECTION.
01  TC                  PIC X(8).
    88 GETOUT               VALUE 'QOUT'.
01  SWITCHES.
    05 FIRST-PAGE-SW        PIC X  VALUE 'N'.

       88 LESS-THAN-A-PAGE     VALUE 'N'.
01  GETQUE2                 PIC X(8) VALUE 'QOUT'.
01  CURR-TIME               PIC X(11).
01  CURR-DATE               PIC S9(7) COMP-3.
01  MESSAGES.
    05 DIS-QUE-MESS         PIC X(20) VALUE
       'QUEUE TESTQ DISPLAYED'.
    05 DIS-QUE-MESS-END     PIC X.
01  TESTQ               PIC X(6) VALUE 'TESTQ'.
01  TEST-QUEUE.
    05 Q-ID             PIC 9(4).
    05 Q-LNAME          PIC X(15).
    05 Q-TIME           PIC X(11).
    05 Q-DATE           PIC 9(5).
    05 TEST-QUEUE-END   PIC X.
01  QUEMAP-REC.
    05  ID                  PIC 9(4).
    05  LNAME                   PIC X(15).
    05  QTIME                   PIC X(11).
    05  QDATE                   PIC 9(5).
    05  MAP-DATE                PIC 9(5).
    05  MAP-TIME                PIC X(11).
PROCEDURE DIVISION.
MAIN-LINE.
    BIND MAP QUEMAP01.
    BIND MAP QUEMAP01 RECORD QUEMAP-REC.
    ACCEPT TASK CODE INTO TC.
    IF GETOUT ENDPAGE
          DC RETURN.
    GET TIME INTO CURR-TIME EDIT
      DATE INTO CURR-DATE.
    MOVE CURR-TIME TO MAP-TIME.
    MOVE CURR-DATE TO MAP-DATE.
    STARTPAGE SESSION QUEMAP01 NOWAIT BACKPAGE BROWSE
       ON DC-SECOND-STARTPAGE NEXT SENTENCE.
*
```

```
    PERFORM A100-GET-QUEUE-REC THRU A100-EXIT
           UNTIL DC-REC-NOT-FOUND.
    IF LESS-THAN-A-PAGE
      MAP OUT USING QUEMAP01
       NEWPAGE RESUME
       MESSAGE IS DIS-QUE-MESS TO DIS-QUE-MESS-END.
*
    DC RETURN NEXT TASK CODE GETQUE2.
*

 A100-GET-QUEUE-REC.
    GET QUEUE ID TESTQ NEXT KEEP
       INTO TEST-QUEUE TO
           TEST-QUEUE-END
       ON DC-REC-NOT-FOUND GO TO A100-EXIT.
    MOVE Q-ID       TO ID.
    MOVE Q-LNAME    TO LNAME.
    MOVE Q-TIME     TO QTIME.
    MOVE Q-DATE     TO QDATE.
    MAP OUT USING QUEMAP01
       DETAIL NEW
       ON DC-FIRST-PAGE-SENT
        MOVE 'Y' TO FIRST-PAGE-SW.
 A100-EXIT.
    EXIT.
```

Using the Terminal Screen To Transmit Data

You can transfer small amounts of alphanumeric data between tasks by using map data fields defined with the following attributes:

- Dark
- Protected
- MDT set on (nonpageable maps only)

For example, you can convert a record's db-key to display format and transmit the reformatted db-key in the map data stream to allow for DB-KEY retrieval on subsequent database access. You can also transmit the next task code to be invoked by a program.

The terminal screen is ideal for transmitting small amounts of data; more than a small amount of data can affect transmission time.

Example of Transmitting Screen Data

The program excerpt below uses the terminal screen to transmit the db-key of a database record to be modified. This allows for more efficient database access.

The program uses the record's db-key, which was transmitted in the map data stream, to retrieve the EMPLOYEE record.

```
01  MAP-WORK-REC.
    05 WORK-DEPT-ID        PIC 9(4).
    05 WORK-EMP-ID         PIC 9(4).
    05 WORK-FIRST          PIC X(10).
    05 WORK-LAST           PIC X(15).
    05 WORK-ADDRESS        PIC X(42).
    05 WORK-DEPT-NAME      PIC X(45).
    05 DARK-DBKEY          PIC X(12).
    05 RETRIEVE-DBKEY      PIC S9(8) COMP.
PROCEDURE DIVISION.
    BIND MAP EMPMAP.
    BIND MAP EMPMAP RECORD MAP-WORK-REC.
    MAP IN USING EMPMAP.
    IF WORK-EMP-ID NOT NUMERIC
       GO TO U100-INVALID-EMP-ID.
*
    COPY IDMS SUBSCHEMA-BINDS.
    READY.
*** CHANGE DARK-DBKEY FROM DISPLAY TO COMP ***
    MOVE DARK-DBKEY TO RETRIEVE-DBKEY.
    OBTAIN EMPLOYEE DB-KEY IS RETRIEVE-DBKEY
       ON DB-REC-NOT-FOUND
          GO TO U100-INVALID-DBKEY.
*
    MOVE WORK-FIRST  TO EMP-FIRST-NAME-0415.
    MOVE WORK-LAST   TO EMP-LAST-NAME-0415.
    MOVE WORK-ADDRESS TO EMP-ADDRESS-0415.
    MODIFY EMPLOYEE.
    FINISH.
*** MAP OUT PROCESSING AND ERROR ROUTINES ***
 .
 .
 .
```

Chapter 10: DC Programming Techniques

This chapter discusses programming techniques used to request DC services. Functionally similar DC DML statements are presented together; sample code that demonstrates typical usage of each statement is included. The DC DML functions are divided into these categories:

- Controlling the flow of processing in the different levels of your task
- Retrieving task-related information-Accessing system, terminal, and user information related to the current task
- Maintaining online data integrity-Monitoring concurrent database access locking database records across tasks
- Managing tables-Adding and deleting tables from the program pool
- Retrieving the current time and date-Accessing the time and date from the DC system
- Writing to the journal file-Writing task-defined records to the journal file
- Collecting DC statistics-Accessing run-time transaction statistics
- Sending messages-Transmitting messages to other terminals, the user, and the log file
- Writing to a printer-Directing data to printer devices
- Writing JCL to a JES2 internal reader-Sending a JCL stream from the application program to a JES2 internal reader
- Modifying a task's priority-Changing the dispatching priority of a task
- Initiating nonterminal tasks-Using nonterminal tasks
- Controlling abend processing-Specifying the flow of control in the event of an abend
- Establishing and posting events-Establishing and posting event control blocks

This section contains the following topics:

Passing Program Control (see page 256)
Retrieving Task-Related Information (see page 262)
Maintaining Data Integrity in the Online Environment (see page 264)
Managing Tables (see page 272)
Retrieving the Current Time and Date (see page 275)
Writing to the Journal File (see page 277)
Collecting DC Statistics (see page 279)
Sending Messages (see page 281)
Writing to a Printer (see page 285)
Writing JCL to a JES2 Internal Reader (see page 287)
Modifying a Task's Priority (see page 288)
Initiating Nonterminal Tasks (see page 288)
Controlling Abend Processing (see page 290)
Establishing and Posting Events (see page 294)

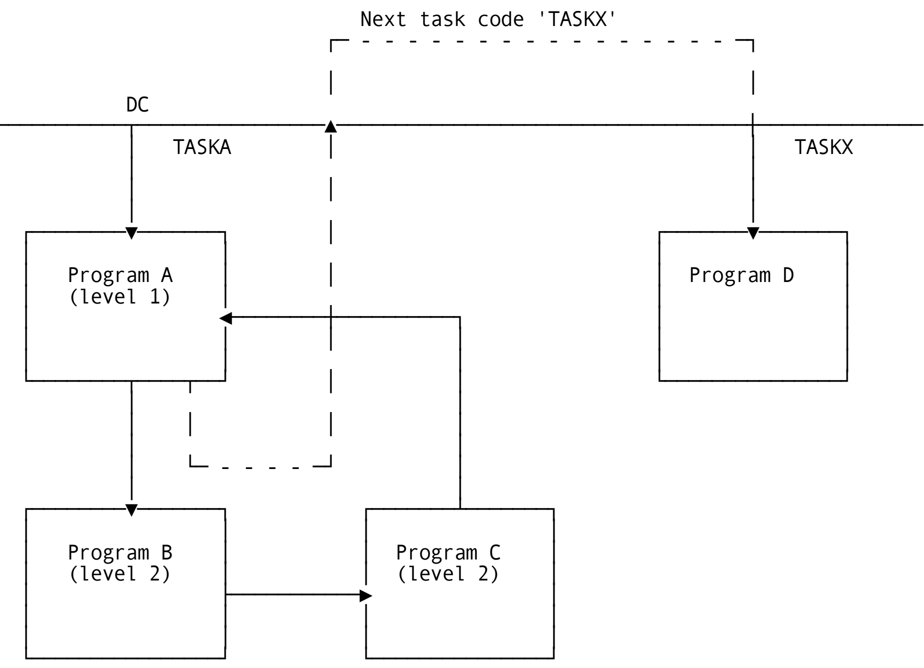
Passing Program Control

DC provides program management facilities that allow you to pass control either between programs in a single task thread or from task to task. Using these program management functions, you can:

- Return control to the next-higher level within a task, optionally specifying the next task to be invoked on the same terminal
- Initiate execution of a program on the same level within a task; control cannot return to the calling program
- Initiate execution of a subordinate-level program within the same task, with the expectation that control will return to the instruction immediately following the request

Levels of Program Control

The figure below shows levels of programs in a task. TASKA invokes Program A, which calls Program B expecting return of control. Program B passes control laterally to Program C, which then returns control to Program A. When Program A is finished, it returns control to DC specifying that TASKX should be the next task invoked on that logical terminal.



Returning to a Higher-level Program

You can return control to a higher level within a task or to DC. If you return control to DC and specify the next task code to be invoked, the task ends and a pseudoconverse begins.

DC RETURN Statement

To return control to the next-higher level in a task, issue a DC RETURN statement, optionally specifying the next task code to be invoked on the terminal.

If the next-higher-level program specifies a next task code, it overrides any task code specified by the subordinate program. If the issuing program is the highest-level program, DC regains control.

Note: You can bypass intervening link levels and return control to DC by issuing a DC RETURN IMMEDIATE statement.

When the Next Task is Invoked

DC invokes the next task differently depending on how it is defined to the DC system:

- If the next task is defined with the **INPUT** attribute, it is executed when the user next presses an AID key.
- If the next task is defined with the **NOINPUT** attribute, it is executed immediately.

Example of Return Specifying Next Task

The program excerpt below returns control to DC and specifies the next task code to be invoked on that terminal.

The first DC RETURN statement returns control to DC. The second DC RETURN statement also specifies that DEPTDISM is the next task invoked on that terminal.

```
 DATA DIVISION.

 WORKING-STORAGE SECTION.
 01  DEPTDISM              PIC X(8)  VALUE 'DEPTDISM'.
 01  SOLICIT-REC.
     05  SOLICIT-DEPT-ID       PIC X(4).
 PROCEDURE DIVISION.
*** ESTABLISH ADDRESSABILITY TO THE MAP AND MAP RECORD ***
     BIND MAP SOLICIT.
     BIND MAP SOLICIT RECORD SOLICIT-REC.
*** CHECK THE AID BYTE ***
     INQUIRE MAP SOLICIT MOVE AID TO DC-AID-IND-V.
*** RETURN CONTROL TO CA IDMS/DC IF OPERATOR HAS PRESSED CLEAR ***
     IF CLEAR-HIT
          DC RETURN.
     MOVE ZERO TO SOLICIT-DEPT-ID.
*** TRANSMIT THE MAP TO THE TERMINAL SCREEN ***
     MAP OUT USING SOLICIT
       NEWPAGE
       MESSAGE IS INITIAL-MESSAGE LENGTH 80.
*** RETURN CONTROL TO CA IDMS/DC AND SPECIFY THE NEXT TASK ***
     DC RETURN
       NEXT TASK CODE DEPTDISM.
```

Passing Control Laterally

After DC gives control to the program specified by an initial task code, that program can transfer control to other DC programs on the same level. That is, the issuing program does not expect return of control.

Steps to Transfer Control

To transfer control laterally, perform the following steps:

1. Invoke the main program specified by the task code.
2. Perform processing, as required.
3. Acquire storage for any parameters to be passed.
4. Transfer control to the second program by issuing a TRANSFER CONTROL XCTL statement, optionally specifying a parameter list.

   Because control is transferred, there is no need to perform the IDMS-STATUS routine.

'COBOL programmers'. If you specify a parameter list, the specified data items must be defined in the LINKAGE SECTION of both the calling and the receiving programs.

'PL/I programmers'. If you specify a parameter list, the specified data items must be defined as based storage in both the calling and the receiving programs. For further considerations related to this subject, see Appendix A, “PL/I Considerations”.

Example of Transferring Control Laterally

The program excerpt below shows a TRANSFER CONTROL request that includes a parameter list containing database retrieval information.

The ERRCHEK program performs error checking and passes control to the GETPROG program, which performs the database access.

```
 PROGRAM-ID.              ERRCHEK.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  GETPROG              PIC X(8) VALUE 'GETPROG'.
 LINKAGE SECTION.
 01  PASS-DEPT-INFO.
     05 PASS-DEPT-ID          PIC 9(4).
     05 PASS-DEPT-INFO-END    PIC X.
 PROCEDURE DIVISION.
     BIND MAP SOLICIT.
     BIND MAP SOLICIT RECORD SOLICIT-REC.
     MAP IN USING SOLICIT.
*** PERFORM ERROR CHECKING ***
     IF SOLICIT-DEPT-ID NOT NUMERIC
        THEN GO TO SOLICIT-ERROR.
*** ACQUIRE STORAGE FOR DEPT-ID TO BE PASSED ***
```

```
    GET STORAGE FOR PASS-DEPT-INFO TO PASS-DEPT-INFO-END
      WAIT LONG USER KEEP STGID 'PDIN'
      ON DC-NEW-STORAGE NEXT SENTENCE.
    MOVE SOLICIT-DEPT-ID TO PASS-DEPT-ID.
*** TRANSFER CONTROL TO DATABASE ACCESS PROGRAM ***
    TRANSFER CONTROL TO GETPROG XCTL
                USING PASS-DEPT-INFO.
```

```
 PROGRAM-ID.              GETPROG.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  GETPROG              PIC X(8) VALUE 'GETPROG'.
 LINKAGE SECTION.
 01  P-DEPT-INFO.
    05 P-DEPT-ID          PIC 9(4).
    05 P-DEPT-INFO-END    PIC X.
 PROCEDURE DIVISION USING P-DEPT-INFO.
 COPY IDMS SUBSCHEMA-BINDS.
    READY.
    MOVE P-DEPT-ID TO DEPT-ID-0410.
*** OBTAIN DEPARTMENT USING PASSED DEPT-ID ***
    OBTAIN DEPARTMENT CALC
      ON DB-REC-NOT-FOUND
      PERFORM ERR-NO-DEPT.
        .
*** FURTHER DATABASE PROCESSING ***
```

Passing Control, Expecting to Return

To transfer program control to a subordinate level, expecting return of control to the instruction immediately following the request, perform the following steps:

1. Invoke the main program specified by the task code.
2. Perform processing, as required.
3. Transfer control to the second program by issuing a TRANSFER CONTROL LINK statement, optionally specifying a parameter list.
4. Perform processing in the subordinate-level program, as required. DC returns control to the next-higher-level program when the subordinate program issues a DC RETURN statement.

Example of Passing Control to a Lower Level

The program excerpt below transfers control to DEPTCHEK, a subroutine that performs error-checking.

The GETPROG program performs processing based on the status returned by the DEPTCHEK program.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  DEPTCHEK               PIC X(8) VALUE 'DEPTCHEK'.

 01  ERRCHEK-INFO.
     05 CHEK-DEPT-ID          PIC 9(4).
     05 CHEK-ERRSTAT           PIC X(4).
 PROCEDURE DIVISION.
     BIND MAP SOLICIT.
     BIND MAP SOLICIT RECORD SOLICIT-REC.
     MAP IN USING SOLICIT.
     MOVE SOLICIT-DEPT-ID TO CHEK-DEPT-ID.
     MOVE 'OK' TO CHEK-ERRSTAT.
*** TRANSFER CONTROL TO ERROR CHECKING PROGRAM ***
     TRANSFER CONTROL TO DEPTCHEK LINK USING
                  CHEK-DEPT-ID
                  CHEK-ERRSTAT.
     IF CHEK-ERRSTAT NOT = 'OK'
       GO TO ERR-DEPT-ID.
     COPY IDMS SUBSCHEMA-BINDS.
     READY.
     MOVE SOLICIT-DEPT-ID TO DEPT-ID-0410.
     OBTAIN DEPARTMENT CALC
       ON DB-REC-NOT-FOUND
       PERFORM ERR-NO-DEPT.
*** FURTHER DATABASE PROCESSING ***
____________________________________________________________________________

 PROGRAM-ID.                DEPTCHEK.
 DATA DIVISION.
 LINKAGE SECTION.
 01  CH-DEPT-INFO.
     05 CH-ID               PIC 9(4).
     05 CH-ERRSTAT            PIC X(4).
```

```
 PROCEDURE DIVISION USING CH-DEPT-INFO.
*** PERFORM ERROR AND RANGE CHECKING ***
    IF CH-ID NOT NUMERIC
       THEN MOVE 'NNUM' TO CH-ERRSTAT
    ELSE
      IF CH-ID > 8000 OR < 1000
         MOVE 'RANG' TO CH-ERRSTAT.
*** RETURN CONTROL TO CALLING PROGRAM ***
    DC RETURN.
```

Retrieving Task-Related Information

DC provides task- and system-related information that you can use in your program. Although you can use this information for any number of purposes, it is most often used for the following:

- **Program flexibility**-You can perform various chapters of code based on the calling task code.
- **Operator information**-You can display the logical terminal ID, the physical terminal ID, and the current DC system number on the terminal screen. The program excerpt below shows this technique.
- **Journaling information**-You can write information such as the user ID, logical terminal ID, the physical terminal ID, and the current DC system number to the journal file. For more information, see Writing to the Journal File.
- **System security**-You can restrict program access based on site-specific factors. For example, you can permit only certain tasks or certain terminals to access a specified program.

Using the ACCEPT Statement

To retrieve task- and system-related information, issue an ACCEPT statement that indicates the information needed and the variable-storage location to which it is to be returned.

Example of Retrieving Task Information

The program excerpt below uses ACCEPT statements to retrieve the task code, the logical terminal ID, the physical terminal ID, and the user ID.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DEPTDISM              PIC X(8)  VALUE 'DEPTDISM'.
01  SOLICIT-REC.

    05 SOLICIT-DEPT-ID        PIC X(4).
    05 TASK-INFO.
      07 TC                 PIC X(8).
        88 GETOUT           VALUE 'DEPTBYE'.
      07 LTERMINAL          PIC X(8).
      07 PTERMINAL          PIC X(8).
      07 CURR-USER          PIC X(32).
PROCEDURE DIVISION.
*** RETRIEVE THE TASK CODE ***
    ACCEPT TASK CODE INTO TC.
*** IF TASK CODE = DEPTBYE, RETURN TO CA IDMS/DC ***
    IF GETOUT DC RETURN.
    BIND MAP SOLICIT.
    BIND MAP SOLICIT RECORD SOLICIT-REC.
*** RETRIEVE LTERM, PTERM, AMD USER ID ***
    ACCEPT LTERM ID INTO LTERMINAL.
    ACCEPT PTERM ID INTO PTERMINAL.
    ACCEPT USER ID  INTO CURR-USER.
    MOVE ZERO TO SOLICIT-DEPT-ID.
    MAP OUT USING SOLICIT
      NEWPAGE
      MESSAGE IS INITIAL-MESSAGE LENGTH 80.
*
    DC RETURN
      NEXT TASK CODE DEPTDISM.
```

The mapout performed by the program excerpt results in this screen display:

```
LTERM:  LT12014                    PTERM:  PV12014
                              USER:  RKN

       *** DEPARTMENT SOLICITOR SCREEN ***


         DEPARTMENT ID:  0000










ENTER AN DEPT ID AND PRESS ENTER ** CLEAR TO EXIT
```

Maintaining Data Integrity in the Online Environment

To maintain database integrity in the online environment, DC allows you to perform the following functions:

- **Place an explicit lock on a database record**-You can restrict other run units' access to a specified database record occurrence.
- **Monitor concurrent database access across a pseudoconverse**-You can determine if other run units have accessed a certain database record during a pseudoconverse.

Setting Longterm Explicit Locks

In pseudoconversational programming, you may be required to lock records across run units for the duration of a transaction. For example, a high-priority update application may lock record occurrences as they are retrieved in order to prevent other run units from accessing data that is about to be modified.

Steps to Set Longterm Locks

To lock a database record explicitly across a pseudoconverse, perform the following steps:

1. Retrieve the database record.
2. Issue a KEEP LONGTERM statement that specifies either the SHARE CURRENT or the EXCLUSIVE CURRENT parameter:
   - **SHARE CURRENT** places a shared lock on the specified record occurrence; other run units can access the record but not update it.
   - **EXCLUSIVE CURRENT** places an exclusive lock on the specified record occurrence; other run units cannot access the record in any way.
3. Perform pseudoconversational processing, as required.
4. As soon as possible, release the explicit lock by issuing a KEEP LONGTERM statement with the RELEASE parameter.

Important! Release longterm locks as soon as possible to provide availability to other run units.

Interaction of Longterm Locks

Locks in effect	Locks allowed for other run units	Locks disallowed for other run units
Shared	Shared and longterm shared	Exclusive and longterm exclusive
Exclusive	None	Shared, exclusive, longterm shared, and longterm exclusive
Longterm shared	**For all run units:** shared and longterm shared .sp **For run units on the same terminal:** exclusive and longterm exclusive	**For run units on other terminals:** exclusive and longterm exclusive
Longterm exclusive	**For run units on the same terminal:** shared, exclusive, longterm shared, and longterm exclusive	**For run units on other terminals:** shared, exclusive, longterm shared, and longterm exclusive

Example of Setting Longterm Exclusive Locks

The first program excerpt below sets longterm exclusive locks in order to ensure that other programs cannot access any data. (The second program excerpt performs database modifications and releases the locks as soon as possible.)

The first program excerpt locks the EMPLOYEE and DEPARTMENT records in order to prevent other run units from modifying them during the pseudoconverse.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CHNGDEPT               PIC X(8)   VALUE 'CHNGDEPT'.
01  KEEP-INFO.
    05 DEPT-LNGTRM-ID        PIC X(4)   VALUE 'DEPT'.
    05 EMPL-LNGTRM-ID        PIC X(4)   VALUE 'EMPL'.
01  MAP-WORK-REC.
    05  WORK-OLD-DEPT-ID      PIC 9(4).
    05  WORK-NEW-DEPT-ID      PIC 9(4).
    05  WORK-EMP-ID           PIC 9(4).
    05  WORK-FIRST            PIC X(10).
    05  WORK-LAST             PIC X(15).
    05  WORK-ADDRESS          PIC X(42).
PROCEDURE DIVISION.
    BIND MAP DCTEST03.
    BIND MAP DCTEST03 RECORD MAP-WORK-REC.
    MAP IN USING DCTEST03.
    MOVE WORK-EMP-ID TO EMP-ID-0415.
    OBTAIN CALC EMPLOYEE
      ON DB-REC-NOT-FOUND GO TO ERR-NO-EMP.
*** SET AN EXCLUSIVE LOCK ON THE CURRENT EMPLOYEE RECORD ***
    KEEP LONGTERM EMPL-LNGTRM-ID
       EXCLUSIVE CURRENT EMPLOYEE.
    MOVE EMP-ID-0415 TO WORK-EMP-ID.
    MOVE EMP-LAST-NAME-0415 TO WORK-LAST.
    MOVE EMP-FIRST-NAME-0415 TO WORK-FIRST.
    MOVE EMP-ADDRESS-0415 TO WORK-ADDRESS.
    IF DEPT-EMPLOYEE IS NOT EMPTY
      OBTAIN OWNER IN DEPT-EMPLOYEE
    ELSE GO TO NO-DEPT.
*** SET AN EXCLUSIVE LOCK ON THE CURRENT DEPARTMENT RECORD ***
    KEEP LONGTERM DEPT-LNGTRM-ID
       EXCLUSIVE CURRENT DEPARTMENT.
```

```
    MOVE DEPT-ID-0410 TO WORK-OLD-DEPT-ID.
*** ALLOW INPUT IN THE NEW DEPARTMENT FIELD ONLY ***
    MODIFY MAP DCTEST03 FOR ALL EXCEPT
        DFLD WORK-NEW-DEPT-ID
        ATTRIBUTES PROTECTED.
    MAP OUT USING DCTEST03.
    DC RETURN NEXT TASK CODE CHNGDEPT.
```

Example of Releasing Longterm Exclusive Locks

This program excerpt maps in the new department ID, disconnects the employee from the old department, and connects the record to the new department.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CHNGSHOW              PIC X(8)   VALUE 'CHNGSHOW'.
 01  TEMP-DEPT-DBKEY          PIC S9(8) COMP.
 01  KEEP-INFO.
    05 DEPT-LNGTRM-ID        PIC X(4)   VALUE 'DEPT'.
    05 EMPL-LNGTRM-ID        PIC X(4)   VALUE 'EMPL'.
 01  MAP-WORK-REC.
    05  WORK-OLD-DEPT-ID      PIC 9(4).
    05  WORK-NEW-DEPT-ID      PIC 9(4).
    05  WORK-EMP-ID          PIC 9(4).
    05  WORK-FIRST           PIC X(10).
    05  WORK-LAST            PIC X(15).
    05  WORK-ADDRESS          PIC X(42).
 PROCEDURE DIVISION.
    BIND MAP DCTEST03.
    BIND MAP DCTEST03 RECORD MAP-WORK-REC.
    MAP IN USING DCTEST03.
    IF WORK-NEW-DEPT-ID IS NOT NUMERIC
       GO TO ERR-NONNUMERIC-DEPT-ID.
*** OBTAIN NEW DEPARTMENT RECORD TO ENSURE IT EXISTS ***
    MOVE WORK-NEW-DEPT-ID TO DEPT-ID-0410.
    FIND CALC DEPARTMENT
       ON DB-REC-NOT-FOUND GO TO ERR-NO-NEW-DEPT.
    MOVE DBKEY TO TEMP-DEPT-DBKEY.
*** REOBTAIN OLD DEPARTMENT ***
    MOVE WORK-OLD-DEPT-ID TO DEPT-ID-0410.
    FIND CALC DEPARTMENT.
*** REOBTAIN EMPLOYEE RECORD ***
    MOVE WORK-EMP-ID TO EMP-ID-0415.
```

```
    FIND CALC EMPLOYEE.
    DISCONNECT EMPLOYEE FROM DEPT-EMPLOYEE.
*** REOBTAIN NEW DEPARTMENT USING SAVED DB-KEY ***
    FIND DEPARTMENT USING TEMP-DEPT-DBKEY.
    CONNECT EMPLOYEE TO DEPT-EMPLOYEE.
*** RELEASE ALL LONGTERM LOCKS ***
    KEEP LONGTERM ALL RELEASE.
    MAP OUT USING DCTEST03 OUTPUT DATA IS ATTRIBUTE
       MESSAGE IS EMP-CONNECTED-MESS LENGTH 80.
    DC RETURN NEXT TASK CODE CHNGSHOW.
```

Monitoring Concurrent Database Access

You can monitor concurrent database access associated with a specific record during a pseudoconverse, instead of locking the record. In most cases, monitoring is preferable to locking because it allows other run units unrestricted access to the specified database record.

'Pageable map applications'. Because you cannot predict the number of occurrences that will be accessed and displayed on a pageable map, it is especially useful to monitor, rather than lock, such records.

Steps Before the Pseudoconverse

To monitor concurrent database access across a pseudoconverse, perform the following steps:

1. Request DC to begin monitoring database concurrent access for the specified record occurrence by issuing a KEEP LONGTERM statement that includes the NOTIFY parameter.
2. Begin the pseudoconverse by issuing a DC RETURN statement.

Steps After the Pseudoconverse

In subsequent tasks, perform the following steps:

1. Determine if the record has been accessed by another run unit by issuing a KEEP LONGTERM statement with the TEST parameter. The components of the value returned as a result of the KEEP LONGTERM TEST statement are as follows:

   - **0**- The record was not accessed.
   - **1**- The record was obtained.
   - **2**- The record was modified.
   - **4**- The record's prefix was modified by a CONNECT or DISCONNECT operation.
   - **8**- The record was logically deleted.
   - **16**- The record was physically deleted.
   - **32**- The status of the record is uncertain.

   For example, a value of 9 means that the record was obtained and logically deleted; the highest possible value is 31, which indicates that all the above actions were performed. You should proceed according to the effect that other run units' processing has on your application and the extent of the other run units' processing.

   Typically, you should require the user to resubmit any transaction in which another run unit has modified a record's data.

   'Pageable map applications'. You should be aware of the effect modified detail occurrences have on each other when using longterm notify locks. For example, if you are modifying a series of records that participate in the same occurrence of a sorted set, a value of 5 (obtained and modified by DISCONNECT/CONNECT) is returned beginning with the second modified detail occurrence.

2. If necessary, issue a KEEP LONGTERM statement with the UPGRADE parameter to place a longterm explicit lock on the specified record.
3. Access the database, as required.
4. Finish longterm monitoring and release longterm locks by issuing a KEEP LONGTERM statement with the RELEASE parameter.

Data Sharing Considerations

A data sharing environment allows programs executing on more than one CA IDMS system to concurrently access and update data in the same areas of the data base. In order to do this, such systems must be members of a data sharing group.

KEEP LONGTERM DML statements will control or monitor data access across members of a data sharing group just as they do within a single CA IDMS system. Programs do not need to be concerned with whether or not the data is being shared between members, with one exception: the retrieval of data is not monitored between members. This means that if a program executing on one member issues a KEEP LONGTERM NOTIFY statement and a program on another member subsequently obtains (but does not update) the affected record, then no indication of the retrieval will be returned to the monitoring program when it checks to see what access has taken place using the KEEP LONGTERM TEST statement. If the accessing program updates the record, the notification value returned to the monitoring program will be an even number greater than 1.

Example of Establishing Longterm Monitoring

The first program excerpt below uses the NOTIFY option of the KEEP LONGTERM statement to monitor concurrent database access across a pseudoconverse. (The second program excerpt performs processing based on the result of database monitoring.)

The first program excerpt uses the NOTIFY option of the KEEP LONGTERM statement to establish monitoring of other run units' access to the specified EMPLOYEE record. It uses the employee's CALC key as the longterm ID.

```
DATA DIVISION.

WORKING-STORAGE SECTION.
01  EMPMOD                 PIC X(8)   VALUE 'EMPMOD'.
01  KEEP-INFO.
    05 KEEP-LNGTRM-ID         PIC X(4).
01  MAP-WORK-REC.
    05 WORK-EMP-ID           PIC 9(4).
    05 WORK-FIRST            PIC X(10).
    05 WORK-LAST             PIC X(15).
    05 WORK-ADDRESS          PIC X(42).
PROCEDURE DIVISION.
    BIND MAP DCTEST03.
    BIND MAP DCTEST03 RECORD MAP-WORK-REC.
.
```

```
 .
 .
    OBTAIN CALC EMPLOYEE
       ON DB-REC-NOT-FOUND GO TO ERR-NO-EMP.
*** USE EMPLOYEE'S CALC KEY FOR THE LONGTERM ID ***
    MOVE EMP-ID-0415 TO KEEP-LNGTRM-ID.
*** BEGIN MONITORING ***
    KEEP LONGTERM KEEP-LNGTRM-ID
        NOTIFY CURRENT EMPLOYEE.
    MOVE EMP-ID-0415 TO WORK-EMP-ID.
    MOVE EMP-LAST-NAME-0415 TO WORK-LAST.
    MOVE EMP-FIRST-NAME-0415 TO WORK-FIRST.
    MOVE EMP-ADDRESS-0415 TO WORK-ADDRESS.
    MAP OUT USING DCTEST03.
    DC RETURN NEXT TASK CODE EMPMOD.
```

Monitoring Concurrent Database Access

The program excerpt below checks to determine if any other run units have accessed the specified record. If any modifications have been made, the program issues a ROLLBACK and informs the user. If no modifications have been made, the program locks the record by issuing a KEEP LONGTERM UPGRADE statement before performing database access and modification.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  REDISP                PIC X(8)   VALUE 'REDISPLY'.
 01  KEEP-INFO.
    05 KEEP-LNGTRM-ID        PIC X(4)   VALUE 'KPID'.
    05 KL-STAT               PIC S9(8)  COMP.
 01  MAP-WORK-REC.
    05 WORK-EMP-ID           PIC 9(4).
    05 WORK-FIRST            PIC X(10).
    05 WORK-LAST             PIC X(15).
    05 WORK-ADDRESS          PIC X(42).
 PROCEDURE DIVISION.
    BIND MAP DCTEST03.
    BIND MAP DCTEST03 RECORD MAP-WORK-REC.
    MAP IN USING DCTEST03.
    MOVE WORK-EMP-ID TO EMP-ID-0415.
    OBTAIN CALC EMPLOYEE
       ON DB-REC-NOT-FOUND GO TO ERR-NO-EMP.
    MOVE EMP-ID-0415 TO KEEP-LNGTRM-ID.
*** TEST TO SEE IF OTHER RUN UNITS HAVE ACCESSED THE RECORD ***
    KEEP LONGTERM KEEP-LNGTRM-ID
        TEST RETURN NOTIFICATION INTO KL-STAT.
*** A RETURNED VALUE THAT IS GREATER THAN 1 MEANS ***
*** THAT THE RECORD WAS MODIFIED IN SOME WAY.    ***
*** ROLLBACK AND REQUIRE THE OPERATOR TO RESUBMIT ***
```

```
***   NOTE: THE SIGNIFICANCE OF THE RETURNED    ***
***       VALUE IS APPLICATION-SPECIFIC.      ***
***       FOR EXAMPLE, FOR SOME APPLICATIONS  ***
***       A RETURNED VALUE > 1 MAY BE         ***
***       ACCEPTABLE, FOR OTHERS, IT MAY NOT.  ***
    IF KL-STAT > 1
      ROLLBACK TASK CONTINUE
      MAP OUT USING DCTEST03 DATA IS ATTRIBUTE
        MESSAGE IS EMPMOD-MESS LENGTH 40
      DC RETURN DEXT TASK CODE REDISP
*** OTHERWISE UPGRADE THE LOCK TO SHARED        ***
    ELSE
      KEEP LONGTERM KEEP-LNGTRM-ID
        UPGRADE SHARE.
*** DATABASE UPDATE PROCESSING ***
```

Managing Tables

At run time, your program can request DC to load a table (for example, an edit or code table) from either the DDLDCLOD area or a load (core-image) library into the program pool. This load does not imply automatic execution; your program continues to run. Typically, you use this function to place nonexecutable data in the program pool.

Making Tables Nonoverlayable

By default, tables and other programs loaded into the program pool can be overlaid when not in use or when in use and waiting for an event. However, unlike an executable module, a table is not reloaded during program execution if it has been overlaid. Therefore, you should define the table with the nonoverlayable attribute during system generation (or at run time with a DCMT VARY DYNAMIC PROGRAM command) so that it cannot be overlaid before the program deletes it.

Deleting Tables

When your program requests DC to delete a table, it does not physically delete that table; rather, it decrements the in-use counter maintained by DC. An in-use count of 0 signals DC that the space occupied by the table can be reused. When your task terminates, DC automatically deletes any tables that have not been explicitly deleted.

If your task requests a nonreentrant table more than once, DC loads a new copy of the table for each request and adds 1 to the in-use counter; each copy corresponds to a separate location in program variable storage. If your task loads the same reentrant or quasi-reentrant table more than once, it must delete that table the same number of times in order to set the in-use counter to 0.

Steps to Load and Delete a Table

To load a table into the program pool and later delete it, perform the following steps:

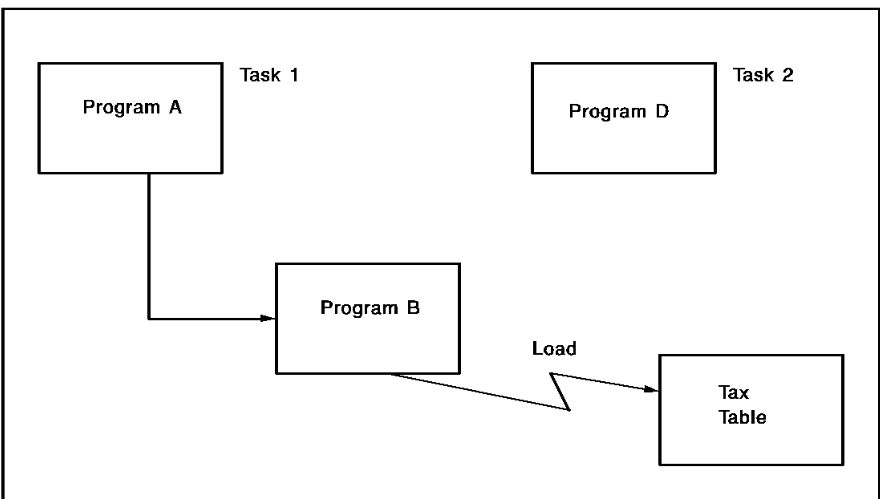
1. Request DC to load the table into the program pool by issuing a LOAD TABLE statement.
2. Perform processing, using the table as needed.
3. When processing is complete, decrement the table's in-use counter by issuing a DELETE TABLE statement.

Note: You can qualify the name of the table by providing the DICTNAME, DICTNODE, or LOADLIB parameter on the LOAD or DELETE statement.

Illustration of Table Management

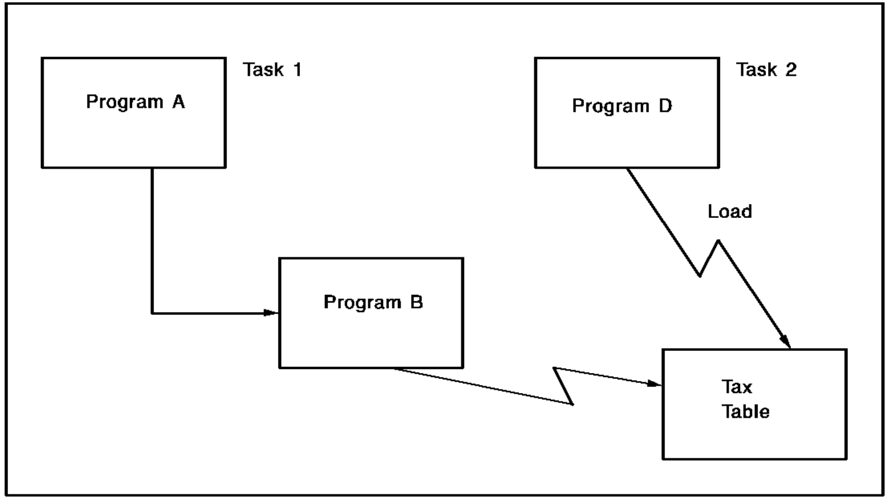
Assume that two tasks are executing under a DC system. Task 1 consists of programs A and B; task 2 consists of program D. The following diagrams illustrate how the tasks load and delete a table:

1. Program B, which is in control of task 1, loads a tax table. Program B continues to execute; DC loads the table into the program pool.



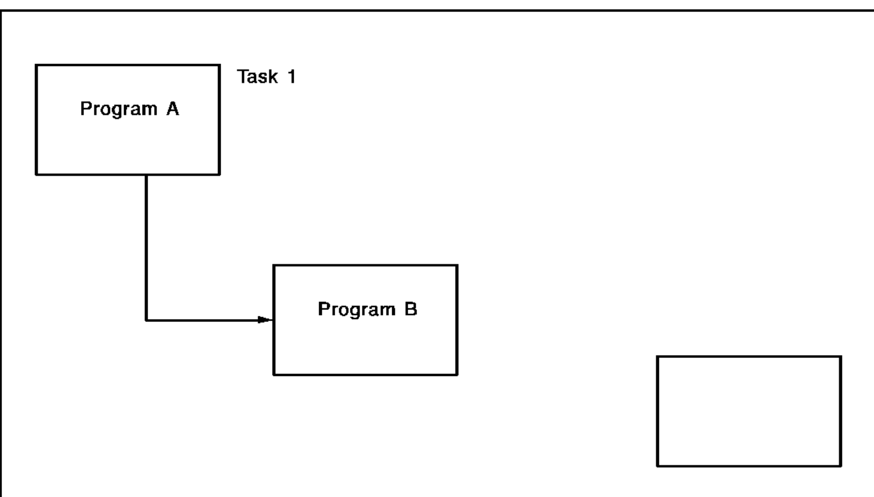
2. Program D, which is in control of task 2, loads the same tax table. Because a copy of the table exists in the program pool and is available (concurrent and not overlaid during a temporary wait), the load is completed with no physical I/O. When task 2 terminates, the table remains in the program pool, as task 1 requires its use.

Program pool



3. Task 1 deletes (signals completion of use) the table. The table remains in the program pool but its in-use counter is set to 0; its storage is now freed for use by other programs.

Program pool



Example of Loading and Deleting a Table

The program excerpt below loads a sales tax table into the LINKAGE SECTION and computes the tax for all items in a specified order. When processing is complete, it decrements the table's in-use count by issuing a DELETE TABLE command.

```
PROGRAM-ID.              SALESTAX.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SALES-TRANS-COUNT        PIC S9(5) COMP-3.
LINKAGE SECTION.

01  SALES-TAX-TABLE.
   02 STATE-AND-TAX          OCCURS 50 TIMES.
     05 STATE-ABB            PIC XX.
     05 STATE-SALES-TAX      PIC SV999.
   02 SALES-TAX-TABLE-END    PIC X.
PROCEDURE DIVISION.
.
.
.
*** LOAD THE SALES TAX TABLE INTO THE LINKAGE SECTION ***
   LOAD TABLE 'SALESTAX' INTO
     SALES-TAX-TABLE TO SALES-TAX-TABLE-END.
   PERFORM A100-COMPUTE-TAX UNTIL SALES-TRANS-COUNT = 0.
*** DECREMENT THE TABLE'S IN-USE COUNT ***
   DELETE TABLE FROM SALES-TAX-TABLE.
```

Retrieving the Current Time and Date

DC allows you to obtain the current time and date from the operating system. You can use these values either for screen display or for journaling purposes.

For more information on journaling, see Writing to the Journal File.

To obtain the current time and date, issue a GET TIME statement that specifies the variable-storage location into which DC is to return the current time and, optionally, the current date.

Example of Obtaining the Current Time and Date

The program excerpt below obtains the time and date for display on the terminal screen.

It obtains the current time in edit format (*hh*:*mm*:*ss*:*hhh*) and the current date in fixed binary format. You must change the date to display format in order to display it on the terminal screen.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DEPTDISM              PIC X(8)  VALUE 'DEPTDISM'.
01  SOLICIT-REC.
    05 SOLICIT-DEPT-ID        PIC X(4).
    05 TASK-INFO.
     07 TC                PIC X(8).
       88 GETOUT            VALUE 'DEPTBYE'.
     07 LTERMINAL           PIC X(8).
     07 PTERMINAL           PIC X(8).
     07 CURR-USER           PIC X(32).
     07 CURR-TIME           PIC X(11).
     07 SYS-DATE            PIC 9(7) COMP-3.
     07 CURR-DATE           PIC 9(5).
PROCEDURE DIVISION.
    ACCEPT TASK CODE INTO TC.
    IF GETOUT DC RETURN.
    BIND MAP SOLICIT.
    BIND MAP SOLICIT RECORD SOLICIT-REC.
*
    ACCEPT LTERM ID INTO LTERMINAL.
    ACCEPT PTERM ID INTO PTERMINAL.
    ACCEPT USER ID INTO CURR-USER.
*** GET THE CURRENT TIME AND DATE ***
    GET TIME INTO CURR-TIME EDIT
      DATE INTO SYS-DATE.
*** CHANGE THE DATE TO DISPLAY FORMAT ***
    MOVE SYS-DATE TO CURR-DATE.
    MOVE ZERO TO SOLICIT-DEPT-ID.
    MAP OUT USING SOLICIT
      NEWPAGE
      MESSAGE IS INITIAL-MESSAGE LENGTH 80.
*
    DC RETURN
      NEXT TASK CODE DEPTDISM.
```

Example of Displaying Current Time and Date

The mapout in the program excerpt results in this screen display:

```
 LTERM:  LT12002                    PTERM:  PV12002
                              USER:  RKN

        *** DEPARTMENT SOLICITOR SCREEN ***

TIME:  11:49:45.60                  DATE:  86.037

           DEPARTMENT ID:  0000











ENTER AN DEPT ID AND PRESS ENTER ** CLEAR TO EXIT
```

Writing to the Journal File

You can write information to the DC journal file to document run-unit related information. For example, you could write to the journal for the following reasons:

- Your **site standards** may require that you record journal information at certain points in a program (for example, when signing on or off).
- You can facilitate **debugging** by writing records to the journal file. For example, as a debugging aid, you can write duplicate scratch and queue entries to the journal file because such records are deleted during ROLLBACK processing.

Steps to Write to the Journal File

To write to the journal file, perform the following steps:

1. Initialize the variable-storage area from which you will write to the journal file.
2. Issue a WRITE JOURNAL statement that specifies the appropriate variable-storage location.

Example of Writing to the Journal File

The program excerpt below writes the current task code, logical-terminal ID, physical-terminal ID, user ID, time, and date to the journal file.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  SOLICIT-REC.
     05 SOLICIT-DEPT-ID        PIC X(4).
     05 TASK-INFO.
       07 TC                 PIC X(8).
         88 GETOUT           VALUE 'DEPTBYE'.
       07 LTERMINAL          PIC X(8).
       07 PTERMINAL          PIC X(8).
       07 CURR-USER          PIC X(32).
       07 CURR-TIME          PIC X(11).
       07 SYS-DATE           PIC 9(7) COMP-3.
       07 CURR-DATE          PIC 9(5).
       07 TASK-INFO-END      PIC X.
 PROCEDURE DIVISION.
*** RETRIEVE TASK CODE ***
     ACCEPT TASK CODE INTO TC.
     IF GETOUT DC RETURN.
     BIND MAP SOLICIT.
     BIND MAP SOLICIT RECORD SOLICIT-REC.
*** RETRIEVE LTERM ID, PTERM ID, AND USER ID ***
     ACCEPT LTERM ID INTO LTERMINAL.
     ACCEPT PTERM ID INTO PTERMINAL.
     ACCEPT USER ID INTO CURR-USER.
*** RETRIEVE CURRENT TIME AND DATE ***
     GET TIME INTO CURR-TIME EDIT
        DATE INTO SYS-DATE.
     MOVE SYS-DATE TO CURR-DATE.
     MOVE ZERO TO SOLICIT-DEPT-ID.
*** WRITE DATA TO THE JOURNAL FILE ***
     WRITE JOURNAL FROM TASK-INFO TO TASK-INFO-END
          NOWAIT SPAN.
     MAP OUT USING SOLICIT
       NEWPAGE
       MESSAGE IS INITIAL-MESSAGE LENGTH 80.
*
     DC RETURN
       NEXT TASK CODE 'DEPTDISM'.
```

Collecting DC Statistics

You can collect run-time statistics related to DC transactions on a logical terminal. This information can be useful both for debugging purposes and as an aid in determining overall program efficiency.

Steps to Collect Statistics

To collect run-time DC statistics related to the transactions performed on a logical terminal, perform the following steps:

1. Establish a 248-byte field in program variable storage in which to copy the transaction statistics.
2. Define the beginning of the transaction by issuing a BIND TRANSACTION STATISTICS statement.
3. Perform pseudoconversational processing, as required.
4. Copy the contents of the transaction statistics block (TSB) into the specified location in variable storage and, optionally, to the DC log file by issuing an ACCEPT TRANSACTION STATISTICS statement.
5. When processing is complete, terminate statistics collection by issuing an END TRANSACTION STATISTICS statement, optionally writing the statistics to variable storage and the DC log file.

Example of Collecting Transaction Statistics

Depending on the invoking task, the program excerpt below initiates statistics collection, copies the TSB to the DC log file, or terminates statistics collection and displays selected statistics on the terminal screen.

```
DATA DIVISION
WORKING-STORAGE SECTION.
 01  TASKCODE              PIC X(8).

    88 FIRSTTIME           VALUE 'INIT'.
    88 SECONDTIME           VALUE 'TRANS'.
    88 FINALTIME           VALUE 'TERMSESS'.
 01  STATISTICS-BLOCK.
    05 USER-ID             PIC X(32).
    05 LTERM-ID             PIC X(8).
    05 PROG-CALL            PIC S9(8) COMP.
    05 PROG-LOAD            PIC S9(8) COMP.
    05 TERM-READ            PIC S9(8) COMP.
    05 TERM-WRITE           PIC S9(8) COMP.
    05 TERM-ERROR            PIC S9(8) COMP.
    05 STORAGE-GET           PIC S9(8) COMP.
    05 SCRATCH-GET           PIC S9(8) COMP.
    05 SCRATCH-PUT           PIC S9(8) COMP.
```

```
    05 SCRATCH-DEL           PIC S9(8) COMP.
    05 QUEUE-GET             PIC S9(8) COMP.
    05 QUEUE-PUT             PIC S9(8) COMP.
    05 QUEUE-DEL             PIC S9(8) COMP.
    05 GET-TIME              PIC S9(8) COMP.
    05 SET-TIME              PIC S9(8) COMP.
    05 DB-CALLS              PIC S9(8) COMP.
    05 MAX-STACK             PIC S9(8) COMP.
    05 USER-TIME             PIC S9(8) COMP.
    05 SYS-TIME              PIC S9(8) COMP.
    05 WAIT-TIME             PIC S9(8) COMP.
    05 PAGES-READ            PIC S9(8) COMP.
    05 PAGES-WRIT            PIC S9(8) COMP.
    05 PAGES-REQ             PIC S9(8) COMP.
    05 CALC-NO               PIC S9(8) COMP.
    05 CALC-OF               PIC S9(8) COMP.
    05 VIA-NO                PIC S9(8) COMP.
    05 VIA-OF                PIC S9(8) COMP.
    05 RECS-REQ              PIC S9(8) COMP.
    05 RECS-CURR             PIC S9(8) COMP.
    05 FILLER                PIC X(4).
    05 FRAG-STORED           PIC S9(8) COMP.
    05 RECS-RELO             PIC S9(8) COMP.
    05 TOT-LOCKS             PIC S9(8) COMP.
    05 SEL-LOCKS             PIC S9(8) COMP.
    05 UPD-LOCKS             PIC S9(8) COMP.
    05 STG-HI-MARK           PIC S9(8) COMP.
    05 FREESTG-REQ           PIC S9(8) COMP.
    05 SYS-SERV              PIC S9(8) COMP.
    05 RESERVED              PIC X(40).
    05 USER-SUPP-ID          PIC X(8).
    05 BIND-DATE             PIC S9(7) COMP-3.

01  STAT-DIS.
    05 WORK-CURR-DATE        PIC 9(5).
    05 WORK-USER-ID          PIC X(32).
    05 WORK-DB-CALLS         PIC 9(4).
    05 WORK-WAIT-TIME        PIC 9(12).
    05 WORK-PAGES-READ       PIC 9(5).
    05 WORK-PAGES-WRIT       PIC 9(5).
PROCEDURE DIVISION.
    BIND MAP STATMAP.
```

```
    BIND MAP STATDIS RECORD STATISTICS-BLOCK.
*
    ACCEPT TASK CODE INTO TASKCODE.
*** FIRST TIME, INITIATE STATISTICS COLLECTION ***
    IF FIRSTTIME
      BIND TRANSACTIONS STATISTICS
      DC RETURN.
*** SUBSEQUENT TIMES, COPY STATISTICS TO VARIABLE STORAGE ***
    IF SECONDTIME
      ACCEPT TRANSACTION STATISTICS
       WRITE INTO STATISTICS-BLOCK
      DC RETURN.
*** LAST TIME, END STATISTICS COLLECTION AND ***
*** COPY STATISTICS TO VARIABLE STORAGE     ***
    IF FINALTIME
      END TRANSACTION STATISTICS
       WRITE INTO STATISTICS-BLOCK
      PERFORM U100-MOVE-FIELDS-TO MAP
      MAP OUT USING STATMAP
         MESSAGE IS STAT-DISPLAY-MESS LENGTH 40
      DC RETURN.
    DC RETURN.
```

Sending Messages

DC provides message management functions that allow you to send messages to the following destinations:

- The log file and the current user, optionally terminating the program
- Other users, logical terminals, or destinations

Sending messages to the current user and to other users is discussed below.

Sending a Message to the Current User

You can send a message predefined in the DDLDCMSG area of the dictionary to the current user, the log file, or both. The message definition can also specify other destinations (for example, the user's console). Additionally, the specified message indicates the action to be taken after the message is written; such action can include the following:

- **Waiting for user reply**-DC does not return control to your task until it receives a reply from the user's console.

- **Abending the program**-DC abends your program, or, optionally, the DC system.
- **Continuing program execution**-DC returns control to your program, optionally writing a snap dump of specified resources.

Retrieving Predefined Messages

One typical use for dictionary-defined messages is to retrieve predefined messages from the DDLDCMSG area rather than include all possible messages in program variable storage.

Messages stored in the dictionary can contain symbolic parameters. Symbolic parameters, identified by an ampersand (&). followed by a two-digit number, can appear in any order within the message. Symbolic parameters provide flexibility in message management.

Steps to Send a Predefined Message

To send a predefined message, perform the following steps:

1. If you are using symbolic parameters, initialize the appropriate variable-storage locations.
2. Issue a WRITE LOG statement that specifies the appropriate variable-storage locations for symbolic parameters, user reply, and message text.

Note: You can specify your own message prefix (to distinguish your messages from DC/UCF system messages) by using the MESSAGE PREFIX IS parameter on the WRITE LOG statement.

Example of Sending a Message from the Dictionary

The program excerpt below uses the following message from the DDLDCMSG area of the dictionary:

```
INPUT DATA IS IN ERROR; DATA FIELD: &01.  &02.
```

The symbolic parameters allow you to transmit more meaningful messages.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  SYMBOLIC-PARAMETERS.
   03 ERR-1.
     05 ERR-1-TEXT           PIC X(15).
     05 ERR-1-END            PIC X.
   03 ERR-2.
     05 ERR-2-TEXT           PIC X(15).
     05 ERR-2-END            PIC X.
   03 ERR-DEPT-ID            PIC X(6)    VALUE 'DEPT-ID'
   03 ERR-NONNUMERIC         PIC X(10)   VALUE 'NONNUMERIC'.
 01  MESSAGES.
     05 MESSAGE-AREA.        PIC X(80).
     05 MESSAGE-AREA-END       PIC X.
 PROCEDURE DIVISION.
     BIND MAP SOLICIT.
     BIND MAP SOLICIT RECORD SOLICIT-REC.
*
     MAP IN USING SOLICIT.
*** IF ERROR, INITIALIZE FIELDS FOR SYMBOLIC PARMS ***
     IF SOLICIT-DEPT-ID NOT NUMERIC
        THEN MOVE ERR-DEPT-ID TO ERR-TEXT-1
             MOVE ERR-NONNUMERIC TO ERR-TEXT-2
             GO TO SOLICIT-ERROR.
        .
        .
 SOLICIT-ERROR.
```

```
*** USE WRITE LOG STATEMENT TO COPY    ***
*** DICTIONARY MESSAGE WITH PARMS INTO ***
*** PROGRAM VARIABLE STORAGE           ***
    WRITE LOG MESSAGE ID 9001080
      PARMS FROM ERR-1 TO ERR-1-END
          FROM ERR-2 TO ERR-2-END
      TEXT INTO MESSAGE-AREA TO MESSAGE-AREA-END
       TEXT IS ONLY.
*** MAP OUT USING MESSAGE FROM DATA DICTIONARY ***
    MAP OUT USING SOLICIT
      MESSAGE IS MESSAGE-AREA TO MESSAGE-AREA-END.
    DC RETURN
      NEXT TASK CODE 'DEPTDIS'.
```

Sending a Message to Other Users

DC provides the facilities for you to send messages to another terminal or user or to a group of terminals or users defined as a destination during system generation.

To send a message to another user, perform the following steps:

1. Initialize the variable-storage location from which the message is to be sent.
2. Issue a SEND MESSAGE statement that specifies the message's destination.

Note: To conserve resources, it is best not to specify the ALWAYS parameter in conjunction with a group of users.

Example of Sending a Message to Another User

The program below is called by other programs in order to send a message to a specified user.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01  MESS-INFO.
    05 MESS-USER-ID          PIC X(32).
    05 MESS-TEXT             PIC X(79).
    05 MESS-TEXT-END         PIC X.
    05 MESS-INFO-END         PIC X.
PROCEDURE DIVISION.
```

```
*** ESTABLISH ADDRESSABILITY TO USER ID AND MESSAGE TEXT ***
    GET STORAGE  FOR MESS-INFO TO MESS-INFO-END
       KEEP SHORT USER STGID 'MSG1'
       ON DC-NEW-STORAGE NEXT SENTENCE.
*** SEND MESSAGE TO SPECIFIED USER ID ***
    SEND MESSAGE ONLY TO USER ID MESS-USER-ID
      FROM MESS-TEXT TO MESS-TEXT-END.
*
    FREE STORAGE STGID 'MSG1'.
    DC RETURN.
```

Writing to a Printer

You can request DC to transmit data from a task to a printer; this allows you to print reports during online processing.

Steps to Transmit Data to a Printer

To transmit data to a printer, perform the following steps:

1. Initialize the variable-storage location from which DC is to write the specified information.
2. Initiate the printing procedure by issuing a WRITE PRINTER statement that indicates the appropriate variable-storage location and report ID, and specifies the print class or destination.
3. Issue subsequent WRITE PRINTER statements that indicate the variable-storage location of the data and the report ID.
4. Optionally, you can indicate the end of a report by issuing a WRITE PRINTER statement that includes the ENDRPT parameter.

CA IDMS Queue

DC does not transmit data directly from program variable storage to the printer. Rather, data is passed to a queue maintained by DC, and from the queue to the printer. The data stream passed to the queue by the WRITE PRINTER request contains only data; DC adds the necessary line and device control characters when it writes the data to the printer.

Note: The WRITE PRINTER command is used extensively under the DC-BATCH operating mode. For more information, see Appendix C, “Batch Access to DC Queues and Printers”.

Example of Writing to a Printer

The program excerpt below writes a report to the printer associated with print class 33. The report consists of the employee ID and name, old department ID, and new department ID for each employee assigned to a new department.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CHNGSHOW              PIC X(8)   VALUE 'CHNGSHOW'.
01  PRINT-CLASS           PIC 999    VALUE 33.
01  PRINT-AREA.
   05 PRI-EMP-ID          PIC X(4).
   05 PRI-EMP-LNAME       PIC X(15).
   05 PRI-EMP-FNAME       PIC X(10).
   05 PRI-OLD-DEPT-ID     PIC X(4).
   05 PRI-NEW-DEPT-ID     PIC X(4).
   05 PRINT-AREA-END      PIC X.
01  TEMP-DEPT-DBKEY       PIC S9(8) COMP.
01  MAP-WORK-REC.
   05  WORK-PRI-CTR       PIC 99.
   05  WORK-OLD-DEPT-ID   PIC 9(4).
   05  WORK-NEW-DEPT-ID   PIC 9(4).
   05  WORK-EMP-ID        PIC 9(4).
   05  WORK-FIRST         PIC X(10).
   05  WORK-LAST          PIC X(15).
   05  WORK-ADDRESS       PIC X(42).
PROCEDURE DIVISION.
 .
 .
 .
*** DISCONNECT EMPLOYEE FROM OLD DEPARTMENT ***
*** CONNECT EMPLOYEE TO NEW DEPARTMENT      ***
 .
 .
 .
*** PRINT PROCESSING FOR EMP TRANSFER REPORT ***
*** IF COUNTER = ZERO, SPECIFY CLASS 33      ***
   IF MAP-PRI-CTR = 0
 WRITE PRINTER FROM PRINT-AREA
          TO PRINT-AREA-END
          REPORT ID 100
          CLASS 33
   ELSE
```

```
*** IF COUNTER > 50, GO TO NEW PAGE ***
 IF-MAP-PRI-CTR > 50
   WRITE PRINTER NEWPAGE FROM PRINT-AREA
            TO PRINT-AREA-END
            REPORT ID 100
*** OTHERWISE WRITE LINE ***
   ELSE
 WRITE PRINTER FROM PRINT-AREA
          TO PRINT-AREA-END
          REPORT ID 100.
   ADD 1 TO MAP-PRI-CTR.
   MAP OUT USING DCTEST03 OUTPUT DATA IS ATTRIBUTE
  MESSAGE IS EMP-CONNECTED-MESS LENGTH 80.
   DC RETURN NEXT TASK CODE CHNGSHOW.
```

Writing JCL to a JES2 Internal Reader

You can write JCL to a JES2 internal reader from a DC application program by issuing a WRITE PRINTER statement that specifies the CLASS parameter.

System Prerequisites

For your program to write JCL to a JES2 internal reader, the system administrator must first take these steps:

1. Define in system generation a SYSOUT line, physical terminal, and logical terminal, using these guidelines:

   ADD LINE *physical-line* TYPE IS SYSOUTL DDNAME IS *ddname*.
   ADD PTERM *physical-terminal* TYPE IS SYSOUTT PRINTER CLASS IS 0
      PAGE WIDTH IS 80.
   ADD LTERM *logical-terminal* PRINTER CLASS = ADD (*nn*)

   In the LTERM statement, *nn* is any valid DC printer class. This class should be reserved for JES2 internal readers only.

2. Include a DD card in the DC run JCL that links *dd-name* to a JES2 internal reader, using this format:

   //*ddname* DD SYSOUT=(A,INTRDR),DCB=(RECFM=F,LRECL=80,BLKSIZE=80)

What the Program Does

The DC application program can write JCL to the JES2 internal reader by using this command:

WRITE PRINTER FROM JCL-STATEMENT-AREA LENGTH 80 CLASS *nn*.

After the last JCL statement is written, you use the same command to write one additional line consisting of: /*EOF with 75 trailing blanks.

Modifying a Task's Priority

DC selects a task for processing based on its priority assignment. A task's priority is determined by the sum of the priority values assigned for the task code, the user, and the terminal. Tasks with the same priority are handled on a first-in/first-out (FIFO) basis.

To change the dispatching priority of a task:

1. Invoke the specified task.
2. Issue a CHANGE PRIORITY statement that specifies a new dispatching priority for the issuing task. The new priority applies only to the current execution of the task.

   Note: You cannot use this statement to alter the priorities of other tasks executing under the same DC system.

Initiating Nonterminal Tasks

Not all tasks in a DC system are associated with a logical terminal; a task not associated with a logical terminal is called a **nonterminal** task. For example, you can initiate processing of another task while your task is still running; this is called **attaching** a task. The new task competes for processor time and runs concurrently with all other tasks, but is not associated with any terminal. You can indicate either that the nonterminal task should begin processing immediately or after a specified length of time.

Because nonterminal tasks are not associated with an LTE, they cannot perform processing related to a logical terminal. For example, nonterminal tasks cannot perform terminal I/O, receive messages, or monitor resource usage.

Attaching a Task

Attached tasks typically perform support functions for the initiating task. For example, an attached task might perform print functions that can be requested by the user.

Steps to Attach a Task

To initiate a nonterminal task to be performed immediately:

1. Issue an ATTACH statement that specifies the task code of the task to be initiated.
2. If the NOWAIT parameter is specified, check for a status of 3711 (DC-MAX-TASKS), which indicates that the task was not initiated because a maximum task condition exists.
3. Perform alternative processing if 3711 is returned.
4. Perform the IDMS-STATUS routine if 3711 is not returned.

Example of Attaching a Task

The program excerpt below initiates the nonterminal task DEPTPRNT if the user presses PA02.

```
 PROGRAM-ID.              GETMENU.
 DATA DIVISION.

 WORKING-STORAGE SECTION.
 01  DEPTPRNT            PIC X(8)     VALUE 'DEPTPRNT'.
 PROCEDURE DIVISION.
    BIND MAP SOLICIT.
    BIND MAP SOLICIT RECORD SOLICIT-REC.
    MAP IN USING SOLICIT.
    INQUIRE MAP SOLICIT MOVE AID TO DC-AID-IND-V.
*** ATTACH TASK IF OPERATOR PRESSES PA02 ***
    IF PA02-HIT
      ATTACH TASK CODE DEPTPRNT
        PRIORITY 100 NOWAIT
      ON DC-MAX-TASKS GO TO MAP-OUT-ERR-MT.
        .
*** INPUT PROCESSING ***
```

Time-Delayed Tasks

You may want to initiate a nonterminal task, but your processing needs require that it not be concurrent with the issuing task. For example, the time-delayed task may compete for resources with the issuing task. DC allows you to initiate a task at the end of a specified period of time.

Steps to Initiate a Time-Delayed Task

To initiate a time-delayed task, perform the following steps:

1. Initiate all appropriate fields.
2. Issue a SET TIMER statement that specifies the START parameter, the time interval (in seconds), the time-delayed task's task code, the timer ID, and the variable-storage location of any data to be passed to the time-delayed task.

External Requests

DC starts an external request task in response to a request issued by a batch program running outside of the DC region of the operating system. The batch program's operating mode (PROTOCOL) must specify DC-BATCH.

For more information on DC-BATCH, see Appendix C, “Batch Access to DC Queues and Printers”.

Queue Threshold Tasks

A sysgen-defined queue ca CAuse a nonterminal task to be started automatically if a predefined threshold is reached. When the queue threshold is reached, DC initiates the nonterminal task. Such a nonterminal task reports on the queue records and then deletes them.

For example, a queue may have a threshold of 100. When the queue exceeds 100 records, DC initiates a task that prints a report and deletes the queue records.

Queue threshold tasks must completely drain the queue and delete all the queue records.

Controlling Abend Processing

A program can abnormally terminate in the following ways:

- **DC** terminates a program upon encountering a processing error (for example, a program check).
- The **program** terminates itself upon discovering a situation that would result in invalid results.

DC allows you to specify abend exits, which are invoked upon a system or a user abend request. These exits specify a program to be invoked in the event of an abend; you can include an abend exit program for each level of a task. Abend exits allow you to determine the cause and severity of the abend. Based on that information, you can return control to the task, return control to the next-higher abend exit, or terminate the program.

Terminating a Task

When your program encounters data that indicates errors have occurred, you should terminate processing. Typically, the IDMS-STATUS routine discovers processing errors and abends your program. You should also terminate processing if a situation exists that makes it impossible to ensure valid results (for example, if you are unable to reaccess a previously obtained database record).

To abnormally terminate a task, issue an ABEND statement that specifies a user-defined abend code. Optionally, you can write a formatted dump to the log file and specify whether previously established abend exits should be invoked or ignored.

For more information on abend exits, see Performing Abend Routines.

Handling db-key Deadlocks

You can include logic in your program that is invoked if your run unit is terminated because of a db-key deadlock. This enables your program to maintain the terminal session and save any data that was previously entered on the screen.

At that point, your program can do one of the following:

- Ask the user to resubmit the transaction.
- Automatically restart the run unit, establish currency, and try again.

What Happens When a Deadlock Occurs

When a run unit is terminated because its request would cause a deadlock condition, the DBMS:

1. **Rolls back the database transaction and terminates the run unit**. The rollback operation releases all locks held by the aborted run unit.
2. **Writes the following message to the log**:

   TASK: *task-code* PROG: *program-name*
   SUBS: *subschema-name* SSCSTAT: *subschema-status*
   RUN-UNIT *run-unit-id* ROLLED OUT.'

3. **Returns control to the issuing task** with a status code of *nn*29, which indicates that a deadlock has occurred.

What To Do

You can continue a terminal session in the event of a deadlock by having your program resubmit a transaction in response to a minor status code of *nn*29. How you do this is largely a site-specific decision. Typically, you resubmit a transaction in one of two ways:

- Inform the user of the deadlock and request the user to resubmit the transaction
- Programmatically resubmit the transaction

Automatically Restarting the Run Unit

If your program automatically restarts the run unit and retries the transaction, it must:

1. Rebind the run unit by:
   a. Reinitializing the ERROR-STATUS field in the IDMS communications block to the value 1400
   b. Issuing the appropriate BIND/READY sequence
2. Reestablish the appropriate currencies before retrying the transaction that originally caused the deadlock.

If You Don't Check for the Minor Code

If your program fails to check for a minor code of *nn*29, you can expect the following results:

- **If AUTOSTATUS is in effect**, your program takes the action specified in the site-specific IDMS-STATUS routine.
- **If AUTOSTATUS is not in effect**, your program responds as specified in the program code that checks status codes.

  If your program does not contain any generic error-checking logic (such as the IDMS-STATUS routine) and, after receiving a minor code of *nn*29, continues to issue database requests without reestablishing a run unit, the DBMS returns a database status of *nn*77 (run unit not bound).

'COBOL'. COBOL programs must redefine the ERROR-STATUS field of the IDMS communications block to access the minor code value.

Example of Resubmitting the Transaction

The program excerpt below informs the user of a database minor code of *nn*29 and requests that the transaction be resubmitted:

```
 WORKING-STORAGE SECTION.

 01  SUBSCHEMA-CTRL.
    03  PROGRAM-NAME         PIC X(8) VALUE SPACES.
    03  ERROR-STATUS         PIC X(4) VALUE '1400'.
         .
         .
    03 SUBSCHEMA-CTRL-END      PIC X(4).
 01  SSC-REDEF REDEFINES SUBSCHEMA-CTRL.
    03  FILLER              PIC X(8) VALUE SPACES.
    03  ERRSTAT-REDEF.
      05  ERRSTAT-MAJ        PIC XX.
      05  ERRSTAT-MIN        PIC XX.
        88 DEADLOCK        VALUE '29'.
    03  FILLER              PIC X(292).
*
 01  MESSAGES.
     05 DBKEY-DEADLOCK-MESSAGE  PIC X(80) VALUE
       'REQUESTED RECORD IN USE.  PLEASE RESUBMIT TRANSACTION'.
 .
 .
 .
 PROCEDURE DIVISION.
 .
 .
 .
```

```
IDMS-ABORT.
   IF DEADLOCK
     THEN
        MODIFY MAP TSKMAP01 TEMPORARY
           FOR ALL FIELDS NOMDT
        MAP OUT USING TSKMAP01
           MESSAGE IS DBKEY-DEADLOCK-MESSAGE LENGTH 80
        DC RETURN NEXT TASK CODE 'UPDATASK'.
IDMS-ABORT-EXIT.
   EXIT.
   COPY IDMS IDMS-STATUS.
******************************************************************
IDMS-STATUS                            SECTION.
******************** IDMS-STATUS FOR IDMS-DC ********************
       IF DB-STATUS-OK GO TO ISABEX.
       PERFORM IDMS-ABORT.
       MOVE ERROR-STATUS TO SSC-ERRSTAT-SAVE
       MOVE DML-SEQUENCE TO SSC-DMLSEQ-SAVE
       SNAP FROM SUBSCHEMA-CTRL TO SUBSCHEMA-CTRL-END
            ON ANY-STATUS NEXT SENTENCE.
       ABEND CODE SSC-ERRSTAT-SAVE
            ON ANY-STATUS NEXT SENTENCE.
ISABEX. EXIT.
DMCL-DC-GEN-GOBACK SECTION.
   GOBACK.
```

Performing Abend Routines

You can establish linkage to an abend routine to which DC passes control if the issuing task terminates. Optionally, you ca CAncel linkage to a previously established abend routine. Each level in a task can have one abend exit in effect at any given time; if more than one abend exit has been established for a level, DC recognizes the last abend exit requested.

Executing Abend Exits

When a task terminates abnormally (following a processing error or an ABEND request), abend exits for the program that was executing at the time of the abend and for all higher-level programs will be executed before the task is terminated. You can prevent DC from executing abend exits automatically by coding the EXITS IGNORED clause in an ABEND request (explained above) or by specifying the abort or continue options in the abend routine's DC RETURN statement. DC RETURN requests are typically handled as follows:

- **Normal** termination passes control to an abend exit at a higher level or to DC:

  ```
  DC RETURN.
  ```

  Abort termination passes control directly to DC, bypassing any other exit programs:

  ```
  DC RETURN ABORT.
  ```

SET ABEND EXIT Statements

To establish linkage to an abend exit, which will be invoked if the issuing task terminates, issue a SET ABEND EXIT statement that specifies the program to be called in the event of an abend.

To cancel any previously requested abend exits for the issuing task level, issue a SET ABEND EXIT OFF command.

Establishing and Posting Events

At certain times, you may need to suspend execution of your task (that is, enter a wait state) until some specific *event* is completed. The most frequent event is I/O. Typically, the wait is automatically handled by DC, which puts the task in a wait state and, upon completion of the I/O, places the task in a ready state.

You can define an event simply by naming the event in a wait request. DC, upon receiving the wait request, places your task in a wait state. The task is returned to a ready state when another task (the task performing the event), upon completion, posts the event by name. One typical use of user-defined events is to synchronize the concurrent execution of different tasks; for example, a terminal task and a concurrent nonterminal task.

Steps to Establish and Post an Event

To place a task in a wait state, waiting for the completion of an event, perform the following steps:

1. Establish a binary fullword field (PIC S9(8) COMP) that identifies the event control block (ECB) to be posted.
2. Begin execution of the task that will post the event by issuing an ATTACH or a SET TIMER statement.
3. Place the issuing task in a wait state by issuing a WAIT statement that names the event to be posted.
4. Post the event, redispatching the waiting task, by issuing a POST or a SET TIMER POST statement in the secondary program.

Chapter 11: Advanced CA IDMS Programming Topics

This section contains the following topics:

Calling a DC Program from a CA ADS Dialog (see page 297)
Basic Mode (see page 299)
Communicating with Database Procedures (see page 301)
Managing Queued Resources (see page 304)

Calling a DC Program from a CA ADS Dialog

CA ADS dialogs can call COBOL, PL/I, or Assembler programs by using the LINK function. For example, a commonly used date conversion routine could be coded in COBOL for use by all CA ADS dialogs running under a DC system.

Because CA ADS calls your program using the LINK command, linkage conventions are the same as if the call were from another DC program.

The calling dialog can pass the following records to the linked program:

- Subschema control block
- Map request block
- Any records to be used in the linked program

Within the linked program, you can issue DC RETURN statements with the NEXT TASK CODE parameter to perform pseudoconversational processing as required.

Extended Run Unit

The linked program may not need to issue any BIND statements or reestablish currencies if the CA ADS dialog establishes an extended run unit.

For more information on CA ADS and extended run units, see *CA ADS Reference Guide*.

Steps to Call a Program from CA ADS

To code a program to be called by CA ADS dialogs, perform the following steps:

1. Define any passed records in the LINKAGE SECTION and code a PROCEDURE DIVISION USING statement.
2. If an extended run unit has been established, do not issue a BIND RUN-UNIT statement. You can issue BIND RECORD statements for any records which have not already been bound for the run unit, and you can issue other appropriate BIND statements.
3. Perform processing, as required.

   If an extended run unit has been established, do not issue FINISH or ROLLBACK statements within the called program. To issue either of these statements, return to the calling dialog with an indicator in a passed status field and let the dialog end the run unit. If you do not follow this procedure, the CA ADS program may receive an error (DC174019) when it tries to save currencies for a run unit that no longer exists.
4. Return control to the CA ADS dialog by issuing one of the following DC RETURN statements:
   - If the program or one of its subroutines has issued a DC RETURN statement, issue a DC RETURN statement that specifies a next task code of 'ADSR'
   - If the program issues no DC RETURN statements, issue a DC RETURN statement that specifies no next task code

Example of a Subroutine Called by CA ADS

The program excerpt below is a subroutine called by a CA ADS dialog to perform data conversion functions.

Depending on the conversion code, it converts a Julian date to Gregorian or a Gregorian date to Julian.

```
WORKING-STORAGE SECTION.

01  CONVERT-CODES.
    05 JULGREG          PIC X    VALUE 'J'.
    05 GREGJUL          PIC X    VALUE 'G'.
01  GREGORIAN.
    10  MM            PIC 99   VALUE ZEROS.
    10  DD            PIC 99   VALUE ZEROS.
    10  YY            PIC 99   VALUE ZEROS.
01  JULIAN.
    10  JULIAN-YY         PIC 99   VALUE ZEROS.
    10  JULIAN-DDD         PIC 999  VALUE ZEROS.
```

```
LINKAGE SECTION.
*** DEFINE RECORDS THAT ARE PASSED FROM CA ADS ***
01 COPY IDMS SUBSCHEMA-CTRL.
01 COPY IDMS RECORD DATE-RECORD.
01 COPY IDMS RECORD DIALOG-REFERENCE-RECORD.
PROCEDURE DIVISION USING SUBSCHEMA-CTRL
                DATE-RECORD
                DIALOG-REFERENCE-RECORD.
   IF CONV-DIRECTION = JULGREG
     PERFORM A100-JULGREG
   ELSE
     IF CONV-DIRECTION = GREGJUL
       PERFORM A100-GREGJUL
   ELSE
     PERFORM A100-ERROR.
*** RETURN CONTROL TO CA ADS PROGRAM ***
   DC RETURN.
*** DATE CONVERSION AND ERROR PROCESSING ***
     .
     .
     .
```

Basic Mode

In basic mode, DC performs device-dependent data transfers between your program and the terminal. Your program must format the data and supply device-control characters based on the type of terminal in use; DC inserts the necessary line control information. For example, with 3270-type devices, you must send and receive data with device-control information that includes write control characters, orders, and buffer addresses.

The figure below shows a basic mode data transfer. DC appends framing characters to the input data stream and performs the required I/O.

```
Data stream as built by the user
 ┌────────────────────────────────────┐
 │ Data and device-control information │
 └────────────────────────────────────┘

Data stream as passed by basic mode request
 ┌──────────────┬────────────────────────────────────┬──────────────┐
 │ LINE CONTROL │ Data and device-control information │ LINE CONTROL │
 └──────────────┴────────────────────────────────────┴──────────────┘
```

For information on using basic mode to support System Network Architecture (SNA) protocols, see *CA IDMS DML Reference Guide for Assembler*.

I/O Requests Under Basic Mode

Basic mode supports synchronous and asynchronous read and write requests. The terms synchronous and asynchronous do not refer to line protocol for data transmission but rather to task processing during I/O operations. Synchronous and asynchronous I/O requests function in the following manner:

- Following a **synchronous** I/O request, control returns to DC, which places the issuing task in an inactive state. When the requested I/O operation is complete, DC places the task in a ready state and the task resumes processing according to its established dispatching priority.
- Following an **asynchronous** I/O request, the issuing task continues executing.

DC assumes that all I/O requests are synchronous unless a program explicitly requests asynchronous processing.

What You Can Do in Basic Mode

Using basic mode terminal management, you can perform the following functions:

- **Read data**-You can transfer data from the terminal to program variable storage.
- **Write data**-You can transfer data from program variable storage to the terminal.
- **Determine if I/O is complete**-You can check to determine if a previously issued asynchronous request is complete.

Reading Data from the Terminal

To transfer data from the terminal screen to program variable storage, issue either a READ TERMINAL or a WRITE THEN READ TERMINAL statement. This transfer begins when the user signals completion of the data entry by pressing an AID key. With 3270-type devices, data can optionally be transferred to the program without user intervention.

Note: WRITE THEN READ TERMINAL is not recommended because it is inherently conversational and holds resources.

Acquiring the Input Buffer

You must dynamically acquire the input buffer for record-element descriptions from the storage pool when the read operation is complete:

- If you specify WAIT, your program must acquire the input buffer by including a GET STORAGE parameter in the READ TERMINAL or WRITE THEN READ TERMINAL request. Your program is also responsible for releasing the acquired storage explicitly with a FREE STORAGE statement. If storage is not explicitly freed, DC releases all acquired buffers when the task terminates.
- If you specify NOWAIT, your program must acquire the input buffer by including a GET STORAGE parameter in the CHECK TERMINAL request.

Where to Define Data

Because storage is acquired by an explicit program request, you must define the associated data-item descriptions in the program's LINKAGE SECTION.

Writing Data to the Terminal

To transfer data from program variable storage to the terminal screen, issue a WRITE TERMINAL statement.

If the output buffer has been dynamically acquired, you can optionally release that area by including a FREE STORAGE parameter in the WRITE TERMINAL request. The associated storage is released when the write operation is complete.

Output buffers that are explicitly acquired and released must be defined in the program's LINKAGE SECTION. h2.Determining if Asynchronous I/O Is Complete

When your program issues an asynchronous I/O request, DC establishes an ECB that is posted only after the requested I/O is complete. Before performing further I/O operations, you must issue a CHECK TERMINAL statement to determine if the ECB has been posted. If the ECB is unposted, indicating that the I/O is not complete, DC places the task in an inactive state. When the operation is complete, DC reactivates the task according to its established dispatching priority.

The CHECK TERMINAL statement must be used following all asynchronous I/O requests, regardless of mode. That is, mapping mode and line mode output requests that specify NOWAIT must issue a CHECK TERMINAL statement before issuing any subsequent I/O requests.

Communicating with Database Procedures

Database procedures, which can be invoked before or after various DML functions, are defined in the schema by the DBA. For example, a data compression routine might be invoked before STORE and MODIFY; a decompression routine might be invoked after FIND.

Use of Database Procedures

Database procedures typically have more authority than application programs. For example, they can access all record elements of a schema-defined record and not just the fields defined in the subschema view. Therefore, if your program must provide more information than is provided by the DBMS itself, you can establish communications with a database procedure. Such instances are unusual; in most cases, you are not aware of the procedures called before or after various DML commands.

Steps to Establish Communication

To establish communications with a database procedure, add the following fields to program variable storage:

- **An 8-byte character literal** aligned on a fullword boundary. This field contains the name of the procedure to be called.
- **A 256-byte area** to which the procedure will be bound. This field defines the information to be passed.

Statements to Communicate with Database Procedures

The following statements enable your program to communicate with database procedures:

- BIND PROCEDURE establishes communication and passes data to the procedure
- ACCEPT PROCEDURE CONTROL LOCATION returns data from the procedure to program variable storage

These statements are explained below.

BIND PROCEDURE

The BIND PROCEDURE statement establishes communication between your program and a DBA-written database procedure. Additionally, the specified data in variable-storage is copied to the application program information block in the central version.

When to Use It

Consult with your DBA to determine when to issue the BIND PROCEDURE. After issuing a BIND PROCEDURE statement, you can modify fields in the 256-byte block without affecting communications with the procedure (for example, by using the ACCEPT PROCEDURE CONTROL LOCATION statement). The data passed is the information contained in the block at the time of the BIND PROCEDURE statement.

Example of the Definition in Variable Storage

The program excerpt below shows a sample 256-byte DBA-defined application program information block as listed in program variable storage.

```
DATA DIVISION
WORKING-STORAGE SECTION.
01  CHECKID            PIC X(8)     VALUE 'CHECKID'.
01  CHECKID-CTRL.
    05 CHECKID-DATE        PIC X(8).
    05 CHECKID-USER        PIC X(32).
    05 CHECKID-INFO        PIC X(216).
```

ACCEPT PROCEDURE CONTROL LOCATION

You can use the ACCEPT PROCEDURE CONTROL LOCATION statement to return a copy of the data bound to a database procedure to a specified location in program variable storage. A BIND PROCEDURE statement previously placed information into this block; this information may have been subsequently updated by the procedure.

ACCEPT PROCEDURE CONTROL LOCATION should be used by programs running under, but in a different region/partition from, the central version.

Example of Communicating With a Database Procedure

The program excerpt below shows the use of the BIND PROCEDURE and the ACCEPT PROCEDURE CONTROL LOCATION statements.

The BIND PROCEDURE statement is issued only once; the ACCEPT PROCEDURE CONTROL LOCATION statement is issued after STORE processing to return information from the user-written procedure. The database procedure itself is transparent to your application.

```
DATA DIVISION
WORKING-STORAGE SECTION.
01  CHECKID            PIC X(8)     VALUE 'CHECKID'.
01  CHECKID-CTRL.
    05 CHECKID-DATE        PIC X(8).
    05 CHECKID-USER        PIC X(32).
    05 CHECKID-INFO        PIC X(216).
PROCEDURE DIVISION.
       .
       .
    READ NEW-EMP-FILE-IN.
```

```
        AT END MOVE 'Y' TO EOF-SW.
*** ESTABLISH COMMUNICATION AND TRANSFER INFO TO ***
*** THE APPLICATION PROGRAM INFORMATION BLOCK    ***
    BIND PROCEDURE FOR CHECKID TO CHECKID-CTRL.
   PERFORM A300-STORE-EMP THRU 0300-EXIT
               UNTIL END-OF-FILE.
*** MOVE DATA FROM THE PROCEDURE TO ***
*** PROGRAM VARIABLE STORAGE        ***
    ACCEPT CHECKID-CTRL FROM CHECKID PROCEDURE.
   PERFORM U100-WRITE-PROC-INFO.
    FINISH.
    GOBACK.
 A300-STORE-EMP.
        .
*** ESTABLISHING CURRENCY AND INITIALIZATION FOR STORE ***
        .
    STORE EMPLOYEE.
    PERFORM IDMS-STATUS.
    PERFORM U500-WRITE-NEW-EMP-REPORT.
 A300-GET-NEXT.
    READ NEW-EMP-FILE-IN
        AT END MOVE 'Y' TO EOF-SW.
 A300-EXIT.
    EXIT.
 U100-WRITE-PROC-INFO.
    DISPLAY '**** STORE PROCEDURE INFORMATION ****'
          'DATE ' CHECKID-DATE
          'USER'  CHECKID-USER
          'INFO FOLLOWS: ' CHECKID-INFO.
```

Managing Queued Resources

Resources are objects that your program must explicitly ask for before it can do any work. Multiple resources may be required to perform a logical unit of work. For example, a database area is a resource that you ask for by issuing a READY statement; a database record occurrence is a resource that you ask for by issuing a FIND/OBTAIN statement.

Holding Resources

The number of resources that you hold and the way that you hold them affects other run units. For example, resources can be shared or exclusive.

You should adhere to the following guidelines when holding resources:

- **Free resources as soon as you are finished** in order that other run units can access them.
- **Hold resources for as short a time as possible**.
- **Acquire the lowest-level lock that you need**. For example, use shared locks instead of exclusive locks whenever possible.

Examples of Resources

Typical resources include:

- Database areas
- Database records
- Storage areas
- Common routines
- Queues
- Site-specific functions (for example, database update)

Meaning of Queued Resource

Your site may utilize queued resources. A queued resource is any resource that requires serial access. That is, only one program can access it at a time.

DC allows you to perform the following resource management functions:

- You can **test** to see if a resource is currently available.
- You can **acquire** a resource for exclusive use.
- You can **release** a previously acquired resource.

These functions are explained below, followed by a list of suggestions that you can use to avoid deadlocks.

Testing for Resource Availability

To determine if a resource or list of resources is currently available, perform the following steps:

1. Issue an ENQUEUE request that includes the TEST parameter.
2. Check for the following statuses:
3. **0000** indicates that all the tested resources were available and have now been enqueued for your task.
   - **3908** indicates that at least one of the tested resources is already owned by another task.
   - **3909** indicates that at least one of the tested resources is not yet owned by another task and is available to your task.

Acquiring Resources

To acquire and lock a resource or list of resources, perform the following steps:

1. Issue an ENQUEUE request that includes either the WAIT or the NOWAIT parameter.
2. Check for the following statuses:
   - **0000** indicates that all requested resources have been acquired and locked.
   - **3901** indicates that at least one of the tested resources cannot be enqueued immediately; to wait would cause a deadlock. No new resources have been acquired.
   - **3908** indicates that at least one of the tested resources is currently owned by another task. No new resources have been acquired.

Releasing Resources

After all processing is complete, release resources by issuing a DEQUEUE statement. You can release resources by name or all at once (by including the ALL parameter).

Avoiding Deadlock

One of the conditions of deadlock is that a program is holding resources while waiting for other resources. The following list explains techniques that your site can use to minimize this condition:

- **Request all required resources at the same time**. Whenever possible, you should try to ensure that your program isn't holding resources while waiting for other resources.
- **If you are denied access to a resource, you should release all previously acquired resources and start over**. After you release previously acquired resources, you can acquire all resources at the same time, as specified above.

- **Your site can follow a protocol of sequential order**. All programs follow a protocol that prescribes the order in which database records will be retrieved and updated.

  Note: This protocol will work only if every program in the system follows it.

  For example, all update applications that use an area sweep can agree to enter the database starting with the DEPARTMENT record rather than the OFFICE record.

  This protocol can also specify the order in which locks will be acquired and released.

Sharing Queued Resources Between CA IDMS Systems

In a data sharing environment, queued resources can be shared between CA IDMS systems that are members of a data sharing group. The benefit of sharing these resources is that access to them can be controlled between programs executing on any member of the group. Whether or not a specific queued resource is shared, is determined by specifications made by the CA IDMS system administrator. Programs accessing queued resources are not sensitive to whether or not a resource is shared, since the DML syntax is the same in either case.

Chapter 12: Testing

This chapter discusses the following topics related to the testing phase of program development:

- Preparing programs for execution-A discussion on precompiling, compiling, and link editing your program
- Selecting local mode or central version-A discussion on using local mode and central version in the test environment
- Overriding subschemas (Release 10.2)-A discussion on overriding the subschema at run time in both the batch and the online environments
- Setting up an online test application-A discussion on creating an online test environment

This section contains the following topics:

Preparing Programs for Execution (see page 309)
Selecting Local Mode or Central Version (see page 310)
Using SYSIDMS Parameters and DCUF SET Statements (see page 310)
Overriding Subschemas (Release 10.2) (see page 311)
Setting Up an Online Test Application (see page 314)

Preparing Programs for Execution

To prepare a CA IDMS load module, perform the following steps:

1. Execute the appropriate **precompiler** to obtain a source-language program. The precompiler is the preprocessor that creates expanded source code and copies any specified dictionary record descriptions or modules.
2. Execute the host-language **compiler** or **assembler** to obtain an object program.
3. Execute the **linkage editor** to obtain a load module (or phase). You should store CA IDMS load modules in a load (core-image) library defined to the test system.

For more information on this phase of testing, see the language-specific *CA IDMS DML* Reference *Guide*.

Selecting Local Mode or Central Version

Follow the guidelines listed below to determine whether to use local mode or central version in the test environment:

- Use **local mode** for testing batch programs. Be sure to back up the database before running any update applications.
- Use the **central version** mainly for online programs. Run batch jobs under the central version only to test aspects of central version processing (for example, update locks).

Using SYSIDMS Parameters and DCUF SET Statements

Using SYSIDMS parameters you can change the specification of these components of the physical environment in which your program executes without changing the program source:

- Database to be accessed
- Dictionary whose load area contains the subschema
- System to which the program should bind

To take advantage of this feature, the BIND RUN-UNIT statements in the program should not specify the DBNAME, DICTNAME, and NODENAME parameters. Any such hard-coded specification cannot be overridden at execution time.

Batch Execution

For a program executing in batch mode, you can make DBNAME, DICTNAME, and NODENAME specifications in the JCL using SYSIDMS parameters.

For documentation of SYSIDMS parameters, see *CA IDMS Common Facilities Guide*.

For sample JCL, see the language-specific *CA IDMS DML Reference Guide*.

Online Execution

For an online program, you can issue a DCUF SET statement to specify database, dictionary, and system. A DCUF SET statement can be submitted to the system by the user or by the program itself.

For more information about DCUF SET statements, see *CA IDMS System Tasks and Operator Commands Guide*.

User Session Attributes

When a program executes under the central version, the executing user is signed on the system automatically if the user is authorized and an explicit signon has not occurred. Signon processing establishes a set of attributes for the user session, including, for example, DBNAME and DICTNAME and the values assigned to them.

The program can access attribute information with a call to the IDMSIN01 entry point to the IDMS module. The program can use this feature to determine whether the user has the appropriate values assigned to the different components of the execution environment.

For more information about using calls to IDMSIN01, see *CA IDMS Callable Services Guide*.

Overriding Subschemas (Release 10.2)

You can override the program-specified subschema to access a test database that exists in a multiple-database environment. This allows you to perform testing without disrupting the production environment.

The procedures for overriding subschemas differ in the batch and the online environment.

Database Name Table

A database to be accessed by an application program with navigational DML must have one or more subschemas defined for it. The central version maintains this association in the database name table created prior to Release 12.0 where an entry exists for each database that can be accessed under that central version. A database name table entry includes the following information:

- The name of the database
- The names of the subschemas that map to the database
- For each subschema that maps to the database, the name of an equivalent subschema that provides the same database perspective (that is, the same record definitions) but maps to different page ranges (that is, different data)

Overriding a Batch Program's Subschema

To override the subschema named in a batch application program in the z/OS environment, perform the following steps:

1. Include an 01-level LINKAGE SECTION entry that names two subordinate data items:

```
01  RUNTIME-TEST-PARMS.

    05 PARM-LENGTH           PIC S9(4) COMP.
    05 RUNTIME-TEST-SUBSCHEMA  PIC X(8).
```

2. Include a USING statement in the PROCEDURE DIVISION heading:

```
PROCEDURE DIVISION USING RUNTIME-TEST-PARMS.
```

3. Before issuing any BIND statements, perform processing to determine if a subschema override has been included in the execution JCL:

```
IF PARM-LENGTH NOT EQ ZERO
   MOVE RUNTIME-TEST-SUBSCHEMA TO SUBSCHEMA-SSNAME.
```

4. Issue database BINDS and perform other processing as needed
5. At run time, include a PARM option in the JCL EXEC statement that specifies the name of the alternative subschema

Local Mode Considerations

The database and the appropriate subschemas must be defined in the database name table in the load (core-image) library.

Note: For more information about the database name table, see *CA IDMS Database Administration Guide*.

'Z/VSE users'. You can pass the alternative subschema name by using a SYSPARM:

// OPTION SYSPARM=*ssname*.

Example of Overriding a Batch Subschema

The program excerpt and JCL below illustrate the batch subschema override technique in the z/OS COBOL environment.

The PARM option on the EXEC statement specifies the name of a test subschema used to override the production subschema. If the parameter is passed, the application program moves the parameter to the SUBSCHEMA-SSNAME field in program variable storage before issuing the BIND RUN-UNIT command.

- **Source code:**
- SCHEMA SECTION.

```
DB EMPSS01 WITHIN EMPSCHM.
     '
01  SUBSCHEMA-SSNAME    PIC X(8)   VALUE 'EMPSS01'.
     '
LINKAGE SECTION.
01 RUNTIME-TEST-PARMS.
  05 PARM-LENGTH             PIC S9(4) COMP.
  05 RUNTIME-TEST-SUBSCHEMA    PIC X(8).
PROCEDURE DIVISION USING RUNTIME-TEST-PARMS.
   MOVE 'TESTPROG' TO PROGRAM-NAME.
   IF PARM-LENGTH NOT EQUAL TO 0 THEN
     MOVE RUNTIME-TEST-SUBSCHEMA TO SUBSCHEMA-SSNAME.
   BIND RUN-UNIT.
 .
 .
 .
```

- **Run-time JCL:**

```
//RUNJOB   EXEC PGM=TESTPROG,PARM='EMPSS01T'
```

Overriding an Online Program's Subschema

A program executing under the central version can be directed to access a specific database by using a DCUF SET DBNAME command. DCUF SET DBNAME establishes a default database for the current logical terminal and overrides the subschema named in the program's BIND RUN-UNIT statement. For example, to establish EMPTSTDB as the default database, either the user or the program can issue this CA IDMS command:

```
DCUF SET DBNAME EMPTSTDB
```

For more information on specifying a default database, see *CA IDMS System Operations* Guide.

'z/OS systems'. The specified database can be overridden by a specification in an IDMSOPTI module or SYSCTL file.

'Z/VSE systems'. The specified database can be overridden by a specification in an IDMSOPTI module.

Setting Up an Online Test Application

There are two typical online test configurations, although your site standards for online testing may be different. The two configurations are as follows:

- Online test programs are link edited into a **test load (core-image) library** that is defined to the DC system. This library is designated for test programs belonging to a specified user or group of users.
- Online test programs are link edited into a load (core-image) library that is defined to a DC system that contains a system dictionary and at least one **application dictionary**. This application dictionary should have been defined for testing in the DC system.

For information on using extended architecture (z/OS) to test programs above the 31-bit line, see the appendix XA Considerations.

Dynamically Defining Programs and Tasks

Before your application can execute under CA IDMS, you must ensure that all of its programs and tasks are defined to CA IDMS. You can define programs and tasks either at system generation or dynamically by issuing DCMT VARY DYNAMIC PROGRAM and DCMT VARY DYNAMIC TASK commands. For example, to dynamically define the DCADDEMP program and its associated task, issue the following DCMT statements:

```
DCMT VARY DYNAMIC PROGRAM DCADDEMP QUASIREENTRANT .

DCMT VARY DYNAMIC TASK ADDEMP INVOKES DCADDEMP INPUT .
```

'CLIST'. Because an application can consist of many programs and task codes, it is a good idea to define an application's dynamic program and task definition statements as a module in the dictionary. This module can then be invoked as a command list (CLIST) from the online DC system.

For more information on command lists, see *CA IDMS System Operations Guide*.

'NEW COPY'. You may need to redefine a recompiled program or map if NEW COPY is defined as MANUAL at system generation. To mark a previously defined program to new copy, issue the following online CA IDMS command:

```
DCMT VARY PROGRAM DCADDEMP NEW COPY
```

Using a Test Load Library (z/OS only)

To execute your test application in a DC system that uses a test load library for such applications, perform the following steps:

1. When coding is finished, compile the programs and link edit them into the load library that has been assigned the specified version number.
2. Define the programs to the DC system either at sysgen or by issuing DCMT VARY DYNAMIC PROGRAM commands.
3. Define the tasks to the DC system either at sysgen or by issuing DCMT VARY DYNAMIC TASK commands.
4. Establish the run-time test version number by issuing a DCUF TEST command.
5. Perform online application testing, as necessary.

Using an Application Dictionary

To execute your test application in a DC system that uses an application dictionary, perform the following steps:

1. Link edit all programs into a load (core-image) library that has been defined to the DC system.
2. Establish the application dictionary as the session default dictionary by issuing a DCUF SET DICTNAME command.
3. Define the programs to the DC system either at sysgen or by issuing DCMT VARY DYNAMIC PROGRAM commands.
4. Define the tasks to the DC system either at sysgen or by issuing DCMT VARY DYNAMIC TASK commands.
5. Perform online application testing, as necessary.

Chapter 13: Debugging

This chapter discusses the following topics related to the debugging phase of program development:

- Using the CA IDMS trace facility-To trace program execution
- Using the CA OLQ menu facility-To confirm database access
- Reading task dumps-To determine the contents of DC control blocks listed in a task dump
- Error checking-To inventory typical programming errors and possible solutions

Note: The online debugger is an additional facility for debugging online CA IDMS/DC and DC/UCF programs written in Assembler, COBOL, or PL/I. That facility is described in *CA* IDMS *Online Debugger Guide*.

This section contains the following topics:

Debugging Batch Programs with the Trace Facility (see page 317)
Using the CA OLQ Menu Facility (see page 320)
Reading Task Dumps (see page 321)
Error Checking (see page 327)

Debugging Batch Programs with the Trace Facility

You can use the CA IDMS trace facility to trace database calls in the following types of programs:

- Batch application programs that run either under the central version or in local mode
- CA IDMS utilities, compilers, and reports

The trace facility writes one line to the SYSLST file for each call to the IDMS module. This line contains the following:

- DML sequence number, if the DEBUG option was specified at compile time and mode is not DC-BATCH
- Database key
- Error status
- DML verb number
- DML verb name
- Record, set, or area name (if applicable)

What You Can Do

You can use the CA IDMS trace facility to:

- Help debug application programs
- Analyze and tune database navigation
- Analyze unfamiliar programs that have been assigned to you for maintenance

Activating the Trace Facility

To activate the trace facility, specify the SYSIDMS parameter DMLTRACE=ON. For example:

```
//SYSIDMS *
DBNAME=TSTDICT
DMLTRACE=ON
 .
 .
 .
```

This activates a trace of all DML calls made by the program.

Trace Facility Output

The following example shows CA IDMS trace facility output for a sample COBOL program:

```
Verb=59  BIND SUBSCHEMA-->EMPSS01     DBNAME=EMPDB     PROGRAM=CBDML04
Verb=59  BIND SUBSCHEMA-->IDMSNWKL    DBNAME=SYSTEM    PROGRAM=RHDCRUAL
Verb=37  READY Area Retrieval         AREA->DDLDCLOD
Verb=54  ACCEPT Current of Run-Unit
Verb=48  BIND Record                  REC-->LOADHDR-156        ADDR=8502AF0C
Verb=32  OBTAIN CALC                  REC-->LOADHDR-156
I D M S  SSCSTAT=0326  ERRREC=LOADHDR-156  ERRAREA=DDLDCLOD  DBKEY=20113:0
Verb=02  FINISH
Verb=59  BIND SUBSCHEMA-->IDMSNWKL    DBNAME=SYSTEM    PROGRAM=RHDCRUAL
Verb=37  READY Area Retrieval         AREA->DDLDCLOD
Verb=54  ACCEPT Current of Run-Unit
Verb=48  BIND Record                  REC-->LOADHDR-156        ADDR=8505C520
Verb=32  OBTAIN CALC                  REC-->LOADHDR-156
I D M S  SSCSTAT=0326  ERRREC=LOADHDR-156  ERRAREA=DDLDCLOD  DBKEY=20113:0
Verb=48  BIND Record                  REC-->LOADHDR-156        ADDR=8505C520
Verb=32  OBTAIN CALC                  REC-->LOADHDR-156
I D M S  SSCSTAT=0326  ERRREC=LOADHDR-156  ERRAREA=DDLDCLOD  DBKEY=20113:0
Verb=02  FINISH
Verb=59  BIND SUBSCHEMA-->IDMSSECU    DBNAME=SYSUSER   PROGRAM=RHDCRUAL
Verb=37  READY Area Retrieval         AREA->DDLSEC
Verb=54  ACCEPT Current of Run-Unit
Verb=48  BIND Record                  REC-->USER               ADDR=8009390C
Verb=32  OBTAIN CALC                  REC-->USER
I D M S  SSCSTAT=0370  ERRAREA=DDLSEC  DBKEY=8000006:0
Verb=02  FINISH
Verb=59  BIND SUBSCHEMA-->IDMSSECU    DBNAME=SYSUSER   PROGRAM=RHDCRUAL
Verb=37  READY Area Retrieval         AREA->DDLSEC
Verb=54  ACCEPT Current of Run-Unit
Verb=48  BIND Record                  REC-->PROFILE            ADDR=85024810
Verb=48  BIND Record                  REC-->ATTRIBUTE          ADDR=85024810
Verb=32  OBTAIN CALC                  REC-->PROFILE
I D M S  SSCSTAT=0370  ERRAREA=DDLSEC  DBKEY=8000006:0
Verb=02  FINISH
```

Turning Trace On and Off

You can use the DML trace facility selectively by adding logic to the program itself. You can switch the trace facility on and off within the program by issuing a call to the IDMSIN01 entry point of the IDMS module.

For information about how to call IDMSIN01 to manage the DML trace facility, see *CA IDMS Callable Services Guide*.

Using the CA OLQ Menu Facility

During the debugging phase of program development, you may need to determine if your application accessed the proper record occurrences or that database modifications were actually applied. You can use the CA OLQ menu facility to help you accomplish these tasks.

The CA OLQ menu facility can perform the following functions:

- Check the sequence of records retrieved by your program
- Test database navigation logic
- Confirm database access and modification

Retrieving Database Records

To retrieve database records, perform the following steps after signing on to the CA OLQ menu facility:

1. On the MENU screen, choose the RECORD option.
2. On the SIGNON screen, indicate the appropriate subschema.
3. On the RECORD SELECT screen, indicate which database records or logical record you want to retrieve.
4. On the FIELD SELECT screen, indicate which fields in the previously specified database records or logical record you want to display; optionally, specify selection criteria.
5. Press Enter. CA OLQ automatically generates retrieval paths and performs the database access.
6. Press Enter again. CA OLQ displays the retrieved data.

Note: The sequence listed above is the default sequence for the RECORD option. You need only press Enter after each step to continue to the next screen.

Storing a CA OLQ qfile

If you will be using the CA OLQ menu facility to perform the same database access repeatedly, you might want to store the logic in an qfile. To create an qfile, perform the following steps:

1. Perform steps 1 through 6 listed above.
2. On the MENU screen, select the EXPRESS ROUTINE option.
3. On the EXPRESS ROUTINE screen, specify the create option and a name.

Executing a CA OLQ qfile

To execute an qfile, perform the following steps:

1. On the MENU screen, select the EXPRESS ROUTINE option.
2. On the EXPRESS ROUTINE screen, specify the execute option and an qfile name.

For more information, see *CA OLQ User Guide*.

Reading Task Dumps

You can use a task dump to obtain task-related information that may not be available under the online debugger. For example, you can find out about the control blocks related to task or program definition by reading a task dump.

You should be familiar with dump reading and hexadecimal notation before trying to read a task dump.

Contents of a Snap Dump

The table below lists the order and contents of a DC formatted snap dump. All of the following information is listed if the ALL parameter of the SNAP command is specified.

Structure	Title starts with	Notes
Summary of all resources for all active tasks	SYSTEM PHOTO	Always listed with task and system snaps unless PHOTO disabled
System registers User registers	SYSTEM REGISTERS	Task and system snaps
System trace entries	TRACE ENTRIES, ORDERED OLDEST TO NEWEST	Task snaps that result from program checks and system snaps
Abend control element (ACE) including PSW, data at PSW, and registers	ABEND C.E.	Task snaps that result from program checks and system snaps
Maps of region and nucleus	MAP OF REGION	System snaps

Structure	Title starts with	Notes
Task's TCE	TASKS TCE ADDRESS	Task and system snaps
Task's DCE	TASKS DCE ADDRESS	Task and system snaps
Task's LTE	TASKS LTE ADDRESS	Task and system snaps
Task's PTE	TASKS PTE ADDRESS	Task and system snaps
Task's PLE	TASKS PLE ADDRESS	Task and system snaps
Task's resources	TASKS RESOURCE CHAIN	Task and system snaps
Options	OPTIONS ADDRESS	Task snaps that result from program checks and system snaps
CCE	CCE ADDRESS	Task snaps that result from program checks and system snaps
SVC parms	SVC PARMS ADDRESS	Task snaps that result from program checks and system snaps
ESE	ESE ADDRESS	System snaps
ERE area	ERE AREA ADDRESS	System snaps
CSA	CSA ADDRESS	Task snaps that result from program checks and system snaps
TCA header	TCA HEADER ADDRESS	System snaps
DCE area	DCE AREA ADDRESS	System snaps
TCE area	TCE AREA ADDRESS	System snaps
RCA header	RCA HEADER ADDRESS	System snaps
RLE area	RLE AREA ADDRESS	System snaps
RCE area	RCE AREA ADDRESS	System snaps

Structure	Title starts with	Notes
DPE area	DPE AREA ADDRESS	System snaps
Loader DCBs	LOADER DCBS	System snaps
LTERM table	LTERM TABLE ADDRESS	System snaps
PLE, Z/OS storage, PTEs, sets	PHYSICAL LINE ENTRY	System snaps
Task table	TASK TABLE ADDRESS	System snaps
Queue table	QUEUE TABLE ADDRESS	System snaps
Destination table	DEST TABLE ADDRESS	System snaps
STG headers (SCTs and SCEs)	STG TBL HDR ADDRESS	System snaps
All storage pools (0 through nnn)	STORAGE POOL NNN	System snaps
Program tables (PDTs and PDEs)	PGM TABLES ADDRESS	System snaps
All program pools present in the system	24 BIT PROGRAM POOL 24 BIT REENTRANT POOL 31 BIT PROGRAM POOL 31 BIT REENTRANT POOL	System snaps
Run unit table	SYS-RU-TAB ADDRESS	System snaps
Extent table	SYS-EXT-TAB ADDRESS	System snaps
DMCL table	DMCL TABLE ADDRESS	System snaps
Operating-system-dependent module	OSXX MODULE ADDRESS	System snaps
Nucleus modules	NUCLEUS ADDRESS	System snaps (reentrant systems only)
SVC module	SVC MODULE ADDRESS	System snaps (reentrant systems only)
Drivers	DRIVERS ADDRESS	
DBIO module	DBIO MODULE ADDRESS	System snaps (reentrant systems only)

Structure	Title starts with	Notes
DBMS module	DBMS MODULE ADDRESS	System snaps (reentrant systems only)

How to Use the Dump

This section tells you the items to look at first to use a DC formatted task dump efficiently.

Abend Message

The abend message that precedes the dump tells you the name of the abending program and the offset of the abend:

```
IDMS DC027001 V12 T3891 D003 PROGRAM CHECK IN SOC7TST AT OFFSET C8E
         PSW WAS 079D1E00 002C3C8E  DUMP OF TASK FOLLOWS
IDMS DC027009 V12 T3891 15:24:07 95.111 CURRENT TASK CODE IS TSK01
IDMS DC027010 V12 T3891 CURRENT LTE ID IS LT12008
IDMS DC027011 V12 T3891 CURRENT USER ID IS RKN
```

System Photo

The system photo (if provided) tells you the abending program's resource control element (RCE) address:

```
*** SYSTEM PHOTO WHEN *TASK* SNAP REQUESTED ***
 RELEASE:  CA IDMS nn.n   TAPE:  xxyymm  OP SYS:  z/OS
 TASK CODE:  *SYSTEM*  TASK ID:  00000000  DISP PRI: 00000255  PROGRAM:  *MASTER*  LTERM:   *N/A*
           RESOURCES:  RCE ADDR   RCE TYP   RESOURCE   RES ADDR      RESOURCE INFO
                  000EA724    STORAGE    STORAGE    001BFC80    LENGTH   00002380
                    .      .      .
                    .      .      .
                    .      .      .
 TASK CODE:  TSK01     TASK ID: 00003891  DISP PRI: 00000050  PROGRAM: SOC7TST   LTERM: LT12008
           RESOURCES:  RCE ADDR   RCE TYPE   RESOURCE   RES ADDR      RESOURCE INFO
```

```
000EB2F4   STORAGE   STORAGE   001C3540   LENGTH  00000040

000EC6EC   STORAGE   STORAGE   00125940   LENGTH  000000C0

000EAF04   STORAGE   STORAGE   00125000   LENGTH  00000940

000EBDA4   STORAGE   STORAGE   001B8E80   LENGTH  00000100

000EBBDC   STORAGE   STORAGE   001B8F80   LENGTH  00000080

000EC11C   STORAGE   STORAGE   001BECC0   LENGTH  00000080

000ED97C   QUEUE     QCE ADDR  001C3548   LENGTH  00000000

►►> 000EB0E4   PROGRAM   PROG ADD  002C3000   PDE ID  SOC7TST
```

Abend Control Element

The abend control element (ACE). Note the program status word (PSW), the data at the PSW, and the user mode registers (0-15):

```
ABEND C.E. ADDRESS IS 00125948

PSW AT TIME OF D003 ABEND 079D1E00 002C3C8E  ILC 6  INTC 04  ACEFLAG C0

DATA AT PSW 002C3C7E  58E0D210 D2036548 E000D203 E0006548

            96F0E003 D2036304 C0A8D203 6308C0A8

 R0/R8    R1/R9    R2/R10   R3/R11   R4/R12   R5/R13   R6/R14   R7/R15

002C4304 002C3BF8 002C3C7E 00000678 002C4174 5E2C43CE 001250A8 00125887

00125888 002C4384 00125008 00125008 002C3908 00125680 00000000 002C45B8
```

Note: The data begins X'10' bytes before the address pointed to by the PSW.

Program Definition Element

The abending program's program definition element (PDE). To locate the PDE, trace the chain of resources to find the abending program's RCE by using the RCE address noted above. The PDE immediately follows:

```
95111 15.24.14  TASK'S RESOURCE CHAIN.

95111 15.24.14  RCE IS AT 000EB2F4   01450001  00000F33  00000040  0 1C3540  00000000  0011B000

          .                       3376703 00000000   *...4....RHDCLTRMLT12008 ........*

          .                       3376703 00000000   *...4....RHDCLTRMLT12008 ........*
```

```
            .                              3376703 00000000   *...4....RHDCLTRMLT12008 ........*

95111 15.24.14  TASK'S PROGRAMS.

95111 15.24.14  RCE IS AT 000EB0E4   02690003  00000F33  00000002  0 2C3000  001DC390  00039BC0

95111 15.24.14  PDE AND PROGRAM TEXT

001DC380                              E2D6C3F7 E3E2E340  0014040 40404040  *           SOC7TST ..    *

001DC3A0   40404040 40404040 40400000 40000000    00000000 00000000  0000000 000F0694   *        .. ........... ......M*

001DC3C0   14000000 800E605C 000E605C 0000000A    00000005 00000000  0039BC0 00000000   *......-*..-*....................*

001DC3E0   00010000 00063488 01010000 28802000    01050000 00000000  0001EE8 002C3000   *.......H...................Y....*

001DC400   00000000 00005F48 E2D6C3F7 E3E2E340    0022240E 002E0023  1000000 000002E2   *........SOC7TST ...............S*

001DC420   001EE81E E8000000 98000100 00000000    00000000 00000000  01DC348 00000000   *..Y.Y...Q................C.....*

95111 15.24.14  COBOL EXTENSION AREA

001DC340                   00000290 00000908    00000678 04800000                  *       ................       *

002C3000   90ECD00C 185D05F0 4580F010 E2D6C3F7    E3E2E340 E5E2D9F1  700989F F02407FF   *.....).0..0.SOC7TST VSR1..Q.0...*

002C3020   96021034 07FE41F0 000107FE 002C4384    002C3000 002C3000  02C3908 002C3678   *O......0.......D................*

002C3040   002C3A40 002C4344 00000000 5E2C43EA    002C3A40 00000678  02C4174 5E2C43CE   *... ........

...... ........

...*

002C3060   002C30A0 002C387F 002C3880 002C4384    002C3000 002C3000  02C3908 002C3678   *......."......D................*

002C3080   0009F8DC 002C3888 F1F54BF2 F24BF1F3    C1D7D940 F2F16B40  1F9F8F6 00000000   *..8....H15.22.13APR 21, 1986....*

002C30A0   F0F0F0F0 00000000 40404040 00000000    00000000 00000000  0000000 00000000   *0000....    ....................*

002C30C0   00000000 00000000 00000000 00000000    00000000 00000000  0000000 00000000   *................................*

    CONTENTS THROUGH 002C30FF SAME AS ABOVE LINE.

002C3100   00000000 00000000 C5D5E3C5 D940C1D5    40C4C5D7 E340C9C4  0C1D5C4 40D7D9C5   *........ENTER AN DEPT ID AND PRE*

002C3120   E2E240C5 D5E3C5D9 405C5C40 C3D3C5C1    D940E3D6 40C5E7C9  3404040 40404040   *SS ENTER ** CLEAR TO EXIT      *

002C3140   40404040 40404040 40404040 40404040    40404040 40404040  4C5D7E3 60C9C440   *                      DEPT-ID *

002C3160   C5C9E3C8 C5D940D5 D6E340C5 D5E3C5D9    C5C440D6 D940D5D6  340D5E4 D4C5D9C9   *EITHER NOT ENTERED OR NOT NUMERI*

002C3180   C3404040 40404040 40404040 40404040    40404040 40404040  0404040 40404040   *C                              *

002C31A0   40404040 40404040 E2D7C5C3 C9C6C9C5    C440C4C5 D7C1D9E3  4C5D5E3 40C3D6E4   *      SPECIFIED DEPARTMENT COU*

002C31C0   D3C440D5 D6E340C2 C540C6D6 E4D5C440    40404040 40404040  0404040 40404040   *LD NOT BE FOUND                *
```

```
002C31E0   40404040 40404040 40404040 40404040    40404040 40404040  2D7C5C3 C9C6C9C5  *                SPECIFIE*
002C3200   C440C4C5 D7C1D9E3 D4C5D5E3 40C8C1E2    40D5D640 C5D4D7D3  6E8C5C5 E2404040  *D DEPARTMENT HAS NO EMPLOYEES  *
```

BL and DMAP Listings

If necessary, use the BL and DMAP listings from the COBOL compiler to locate questionable variable-storage values.

Additional Control Blocks

You may find it useful to look at the following control blocks:

- The common system area (CSA). You should check the following fields in the CSA:
  - CSAFLAG1 through CSAFLAG5 (starting at CSA + X'6C0') contain status flags.
  - CSATRCLO (CSA + X'370'), CSATRCHI (CSA + X'374'), and CSATRCNX (CSA + X'378') point to the first, last, and next available trace entries (this is useful if there is no system trace provided with your dump).
  - CSA + X'1134' lists the last 33 messages ordered newest to oldest; each doubleword contains the task ID, the message ID, and the severity code.
- The status flags in the task control element (TCE).
- The LTE contains fields that indicate the dictname, dictnode, dbname, whether the logical terminal has an autotask or CLIST, and the next task code.
- The dispatch control element (DCE) contains task information.

For more information, including a control block cross reference, see *CA IDMS DSECT Reference Guide*.

Error Checking

The table below presents a brief list of typical programming errors and possible solutions.

Problem	Language	Reason and/or action
Unconnected member records are showing up as set members.	All	Be sure to issue an IF MEMBER statement before OBTAIN OWNER in a set with the optional or manual set membership options.

Problem	Language	Reason and/or action
Skipping records in an area sweep.	All	Your processing may have taken you to another database page; be sure to issue a FIND CURRENT record-name before issuing an OBTAIN NEXT WITHIN area-name.
A program performs extra processing in addition to IDMS-STATUS.	COBOL	■ IDMS-STATUS is a COBOL SECTION; be sure to do one of the following: ■ Place IDMS-STATUS at the end of the program. ■ Ensure that the code following IDMS-STATUS is also a SECTION. ■ Always perform IDMS-STATUS THRU ISABEX.
Queue records written to the database are not being kept.	DC-BATCH	Be sure to issue either FINISH TASK or COMMIT TASK, or the queue records will be deleted at the end of the run unit.
Storage violation when initializing acquired storage fields.	DC	When you define acquired storage length by using the THROUGH option, CA IDMS does not acquire storage for the dummy byte. If you initialize fields on the group level, it may include the dummy byte, thus causing a storage violation.
Map displayed with no variables.	DC	Be sure to issue a BIND MAP statement for the map and BIND MAP RECORD statements for all map records.
The same errors occur despite repeated modification and recompilation.	DC	Be sure to issue a DCMT VARY PROGRAM NEW COPY statement following recompilation. Be sure to VARY the correct version of the program.

Appendix A: PL/I Considerations

This appendix explains:

- Passing parameters to PL/I programs in the TRANSFER CONTROL command, which differs from the procedure in the COBOL environment.
- Debugging a PL/I program using the CA IDMS Online Debugger

This section contains the following topics:

Transferring Control (see page 329)
Using the Online Debugger with PL/I (see page 330)

Transferring Control

:p In order to pass parameters in a LINK or XCTL statement, you must include the following PL/I declarative in your program:

```
DECLARE IDMSP ENTRY;
```

If you pass parameters to a PL/I program from a non-PL/I program (CA ADS dialog, or a COBOL or Assembler program), you must include special parameters in order to establish addressability to the passed data.

The program excerpt below shows the extra code necessary to transfer from a non-PL/I program to a PL/I program.

The parameters F1, F2, and F3 provide the addresses on which to base the structures that are passed.

```
TESTPROC: PROCEDURE (F1,F2,F3) OPTIONS (MAIN,REENTRANT);
DCL  (EMPSS01T SUBSCHEMA, EMPSCHEM SCHEMA) MODE (IDMS_DC) DEBUG;
DCL  IDMS ENTRY OPTIONS (INTER,ASM);
DCL  IDMSP ENTRY;
DCL  ADDR BUILTIN;
DCL  (F1,F2,F3) FIXED;
DCL  PASSED_FIELD_1 FIXED BIN(31) BASED (ADDR(F1));
INCLUDE IDMS (SUBSCHEMA_CTRL BASED (ADDR(F2)));
INCLUDE IDMS (RECORD_AA BASED (ADDR(F3)));
 .
 .
 .
```

For more information on passing parameters to a PL/I program from an Assembler program, see *CA IDMS DML Reference Guide for PL/I*.

Using the Online Debugger with PL/I

You can use the online debugger to detect, trace, and resolve programming errors in DC PL/I programs. To use the online debugger with PL/I, you should be familiar with both hexadecimal notation and hexadecimal arithmetic.

The phases of the debugging process are discussed below, followed by a sample PL/I online debugger session.

Computation Phase

Before beginning the debugging process, it is a good idea to determine the breakpoints you want to set and the storage locations you want to examine:

- To determine the hexadecimal offset of an **executable program instruction** at which you wish to set a breakpoint, perform the following steps:

  1. Examine the cross-reference table portion of your link-edit listing for an entry in the form *program-name*1. Record the hexadecimal offset listed under ORIGIN:

```
            CROSS REFERENCE TABLE

CONTROL SECTION              ENTRY
 NAME   ORIGIN  LENGTH         NAME   LOCATION
PLISTART    00     50
                        PLICALLA     6
PLIMAIN     50      8
*PLIPROG2    58    394
*PLIPROG1   3F0    EB4
                        PLI3PROG   3F8
IDMSPLI    12A8    284
```

  2. Examine the PL/I compiler portion of your listing and record the line number of the statement at which you wish to set the breakpoint:

```
133          WORK_LAST = EMP_LAST_NAME_0415;
134          WORK_FIRST = EMP_FIRST_NAME_0415;
                                  /*
           MAP OUT (DCTEST01) OUTPUT DATA YES
             MESSAGE (INITIAL_INSTRUCTIONS_MSG_1)
             LENGTH (25)
             DETAIL NEW KEY (DBKEY).
```

```
                    */
135     /* IDMS PL/I DML EXPANSION */    DO;
136     DML_SEQUENCE=0013;
137     DCCFLG1=0;
138     DCCFLG1=13;
139     DCCFLG2=16;
140     DCCFLG3=0;
141     DCCFLG4=4;
142     DCCFLG5=72;
143     DCCFLG6=0;
```

3. Examine the Assembler listing generated by the LIST option, locate the previously recorded PL/I line number, and record its corresponding hexadecimal displacement value:

```
* STATEMENT NUMBER  136
0006AA  41 80 7 21C        LA    8,SUBSCHEMA_CTRL.D
                 CCALIGN_AREA.FILLE
                 R0001
0006AE  58 40 3 124        L     4,292(0,3)
0006B2  50 40 8 008        ST    4,SSC_ERRSAVE_AREA
                 .DML_SEQUENCE
```

4. Add the origin offset and the breakpoint instruction's hexadecimal displacement to obtain the breakpoint address:

```
X'3F0' + X'6AA' = X'A9A'
```

- To determine the offset of **AUTOMATIC variables**, locate the variable storage map and record the displacement value for each variable you wish to examine during the debugging process:

```
MAP_WORK_REC        1    796    31C   AUTO

WORK_DEPT_ID        1    796    31C   AUTO
WORK_EMP_ID         1    800    320   AUTO
WORK_FIRST          1    804    324   AUTO
WORK_LAST           1    814    32E   AUTO
WORK_ADDRESS        1    829    33D   AUTO
WORK_STREET         1    829    33D   AUTO
WORK_CITY           1    849    351   AUTO
WORK_STATE          1    864    360   AUTO
WORK_ZIP            1    866    362   AUTO
WORK_DEPT_NAME      1    871    367   AUTO
```

  You locate AUTOMATIC variables at run time through Register 13.

- To determine the location of **STATIC INTERNAL variables**, examine the static internal storage map to find the hexadecimal offset for each variable you wish to examine during the debugging process.

  You locate STATIC INTERNAL variables at run time through Register 3.

Sample Online Debugger Session

To use the online debugger with a DC PL/I program, perform the following steps:

1. Compile the program with the LIST, OFFSET, XREF STORAGE, and MAP compiler options before defining it to the DC system.

2. Record breakpoint and storage displacements as explained in Computation Phase above.

3. Initiate the debugger session by entering the DEBUG task code from DC; the DEBUG> prompt is displayed, indicating that the debugger is in control:

   ENTER NEXT TASK CODE:
   debug

   DEBUG>

4. Indicate the program to be debugged by entering DEBUG followed by the program name; the debugger verifies the program name:

   DEBUG>
   debug pliprog

   DEBUG PLIPROG
   DEBUG> DEBUGGING INITIATED FOR PLIPROG VERSION 1
   DEBUG>

5. Establish breakpoints by issuing the AT command, followed by $, which signifies the base register, followed by the previously computed breakpoint address; the debugger verifies the establishment of the breakpoint:

   DEBUG>
   at $ + @a9a

   AT @A9A
   AT> @A9A ADDED
   DEBUG>

6. After all breakpoints have been set, leave the setup phase of the debugger session by issuing the EXIT command:

   DEBUG>
   exit

7. Initiate the run-time phase by issuing the task code that invokes the task that the program participates in:

   ENTER NEXT TASK CODE:
   deptmod

8. When a breakpoint is encountered at run time, the debugger assumes control and identifies the address, program, and the debugger expression that was used to establish the breakpoint:

```
AT OFFSET @A9A IN PLIPROG EXPRESSION @BDE
DEBUG>
```

9. You can now examine program variable storage by issuing LIST commands; indirect addressing is used based on the previously noted register and offset:

```
list %:r13 + @31c 32

LIST %:R13 + @31C 32
001DB7F4  F3F2F0F0 F0F0F0F4 C8C5D9C2 C5D9E340 *32000004HERBERT *
001DB804  4040C3D9 C1D5C540 40404040 40404040 *  CRANE        *
```

If your program contains any nested procedures or begin blocks, you will need to navigate the chain of dynamic storage areas (DSAs) to obtain the correct variable-storage base address. To navigate the DSA chain for nested procedures or begin blocks, list the contents of register 13 to determine the DSA for the current level of nesting:

```
list %:r13

LIST %:R13
001C7A30  84200000 001C7948 00000000 5E422A20 *D..........
...*
```

For subsequent levels of nesting, perform the following step:

a. List the absolute address contained 4 bytes off of the previously displayed line:

```
list @1c7948

LIST @1C7948
001C7948  84200000 001C74D8 00000000 4E4227EC *D......Q....+...*
```

b. When you have reached the final level of nesting, use the address 4 bytes off of the display as the base address to list AUTOMATIC variable-storage values:

```
DEBUG>
list 1c74d8 + @31c 32

LIST 1C74D8 + @31C 32
001C77F4  F3F2F0F0 F0F0F0F4 C8C5D9C2 C5D9E340 *32000004HERBERT *
001C7804  4040C3D9 C1D5C540 40404040 40404040 *  CRANE        *
```

To examine variables defined as BASED storage, perform the following steps:

c. Using indirect addressing, list the contents of the associated pointer variable:

```
DEBUG>
list %:r13 + @d4

LIST %:R13 + @D4
001499E0  00149AC8 00000000 00000000 00000000 *...H...........*
```

d. List the absolute address to display the BASED variable's values:

```
DEBUG>
LIST @149ac8 16
00149AC8  F1F1F1F1 C4C5D7E3 00000000 00000000 *1111DEPT........*
```

10. To continue program execution, enter the RESUME command:

```
DEBUG>
resume
```

11. To end a debugger session, enter the QUIT command from the DEBUG> prompt:

```
DEBUG>
quit

QUIT
QUIT DEBUGGER
ENTER NEXT TASK CODE:
```

Appendix B: Assembler Considerations

This appendix explains the following Assembler topics, which differ from the COBOL environment:

- Batch error checking-A discussion on coding error-checking routines for batch programs
- DC error checking-A discussion on coding error-checking routines for online programs

For more information related to using Assembler with DC, see *CA IDMS DML Reference* Guide *for Assembler*.

Checking the Status of Calls to DB

Assembler programs do not use the IDMS-STATUS block; you must explicitly code your own error-checking routines. You should check the ERRSTAT field after every DB DML call; if the DBMS returns an unexpected nonzero value, you should:

1. Display the following IDMS communications block fields:
   - PGMNAME
   - ERRSTAT
   - ERRORREC
   - ERRORSET
   - ERRAREA
   - RECNAME
   - AREANAME
   - DMLSEQ (if the DEBUG option is specified)

   You should also display any other relevant variable-storage fields.
2. Issue the @ROLLBAK command.
3. Terminate the program.

Checking the Status of Calls to DC

Assembler DC programs do not need to use the IDMS-DC communications block. You explicitly check the value returned to register 15 to determine the result of a DC call. If the call to DC included database access, you must check:

- Register 15 for return codes issued by DC
- The ERRSTAT field for status codes issued by DB

If DC returns an unexpected nonzero status, you should:

1. Save the register 15 value
2. Write a memory dump of the IDMS communications block and any other relevant variable-storage fields by using the #SNAP command
3. Terminate the program by using the #ABEND command

Testing for the return code in register 15 is not usually necessary because most Assembler DML commands have options that take action based on the return code value.

Appendix C: Batch Access to DC Queues and Printers

This appendix explains how a batch application program can use services of the CA IDMS central version.

DC-BATCH Mode

DC-BATCH allows your batch program to access DC queues and printers. Using DC-BATCH, your program can access the database, issue CA IDMS queue management commands, and transmit data to DC printers.

Note: DC-BATCH uses the IDMS communications block.

Batch Access to CA IDMS Queues

You can use DC-BATCH to establish a task within a batch application program. This allows your program to read data from queue records while performing normal database activities. Additionally, you can take advantage of DC facilities for locking queue records and performing recovery.

Perform the following steps to access queue records from a DC-BATCH program:

1. Link edit the program with the batch interface module, IDMS.
2. Specify a mode of DC-BATCH.
3. Initiate the DC task by issuing a BIND TASK statement before any other BIND statements. BIND TASK establishes communication with the DC system and allocates a packet-data-movement buffer to contain the queue data.
4. Issue retrieval and modification statements beginning with BIND RUN-UNIT and ending with FINISH. Within a task, you can code as many BIND/READY/FINISH sequences as required.

5. Issue GET QUEUE, PUT QUEUE, and DELETE QUEUE statements to access queue records. Queue access requests must fall between the BIND TASK and the FINISH TASK statements; they need not fall between BIND RUN-UNIT and FINISH.
6. Terminate the DC task by issuing a FINISH TASK statement. FINISH TASK relinquishes control over all database areas associated with the task and establishes an end-of-task checkpoint in the journal file for the queue areas that have been accessed by the task.

Within the task, you can issue COMMIT TASK and ROLLBACK TASK statements to write checkpoints and effect recovery coordinated with the CA IDMS run unit.

Note: Be sure to issue a BIND TASK statement and a FINISH TASK statement and to include the TASK parameter of the COMMIT and ROLLBACK statements.

Batch Access to DC Printers

To access DC printers from a batch application program, perform the following steps:

1. Link edit the program with the batch interface module, IDMS.
2. Specify a mode of DC-BATCH.
3. Issue a BIND TASK statement.
4. Issue WRITE PRINTER requests as needed to build a report and direct it to a printer.

   Note: Batch programs cannot issue WRITE PRINTER SCREEN requests.

   Terminate the DC task with a FINISH TASK statement.

Appendix D: XA Considerations

This appendix explains how your program can use XA features.

XA Support

CA IDMS supports XA for Assembler and VS COBOL II programs. To run your application in 31-bit mode, the following conditions must be met:

- Your program must be able to run above the 16-megabyte line. For more information, refer to the appropriate IBM documentation.
- Your DC system must contain at least one XA program pool, XA reentrant pool, and XA storage pool.
- You must link edit your program with the following options:

  RMODE=ANY,AMODE=31

  You must define the task, either at sysgen or by using a DCMT VARY DYNAMIC TASK command, with the LOCATION=ANY parameter.

Appendix E: Running a Program Under TCF

This appendix explains the processing that your program must perform in order to run under the Transfer Control Facility (TCF). TCF allows you to transfer from one online application to another without having to return first to DC.

This section contains the following topics:

Overview of TCF (see page 341)
Defining a TCF Task to the DC System (see page 343)
Using the UCE for Communication Under TCF (see page 344)
Determining if TCF Is Active (see page 345)
Starting a New Session (see page 346)
Resuming a Suspended Session (see page 346)
Processing a Pseudoconverse (see page 347)
Displaying Error Messages (see page 348)
Sample Application Under TCF (see page 348)

Overview of TCF

Using TCF, you can suspend a session of an online application, transfer directly to another online application, then transfer back and resume the suspended session.

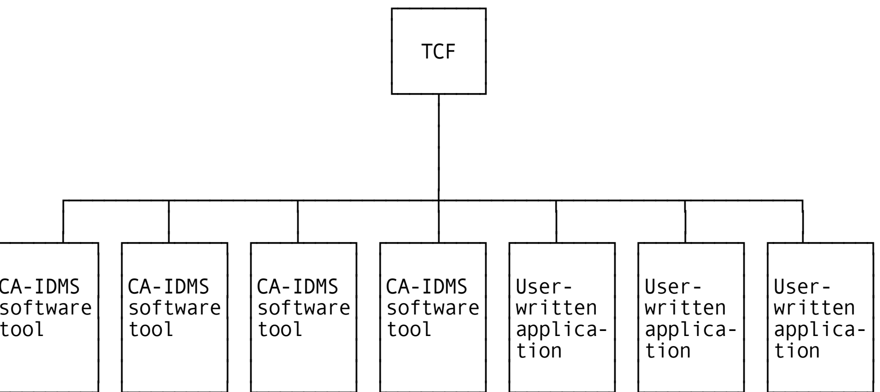
Before writing an application to run under TCF, you should be thoroughly familiar with TCF and the CA IDMS software tools that it invokes.

For more information on TCF, see *CA IDMS Common Facilities Guide*.

TCF Internal Processing

You should be aware of TCF internal processing:

- TCF invokes an application through a TRANSFER CONTROL LINK function. This allows TCF to regain control after every pseudoconverse and every DC RETURN statement. The figure below shows a typical TCF program structure.



- All communication between an application and TCF occurs through the universal communications element (UCE). The UCE is the link between an application and TCF. Each application and TCF set fields in the UCE that indicate specific actions to be taken.

  The record layout of the UCE is presented below.

  Note: Be sure to copy **version 2** of the UNIVERSAL-COMMUNICATIONS-ELEMENT. Depending on the language, you may need to define a synonym with a shorter name.

```
01  UNIVERSAL-COMM-ELEMENT.
    03  UCE-IDENT-02            PIC XXXX.
    03  UCE-DBNAME-02           PIC X(8).
    03  UCE-NODE-NAME-02        PIC X(8).
    03  UCE-DICT-NAME-02        PIC X(8).
    03  UCE-DICT-NODE-02        PIC X(8).
    03  UCE-SCHEMA-NAME-02      PIC X(8).
    03  UCE-SCHEMA-VER-02       PIC 9999   USAGE COMP.
    03  UCE-SUBSCHEMA-NAME-02   PIC X(8).
    03  UCE-SUBSCHEMA-VER-02    PIC 9999   USAGE COMP.
    03  UCE-INPUT-POINTER-02    PIC S9(8)  USAGE COMP.
    03  UCE-INPUT-LENGTH-02     PIC S9(8)  USAGE COMP.
    03  UCE-OUTPUT-POINTER-02   PIC S9(8)  USAGE COMP.
    03  UCE-OUTPUT-LENGTH-02    PIC S9(8)  USAGE COMP.
    03  UCE-ENTITY-OCCURRENCE-02 PIC X(32).
    03  UCE-ENTITY-OCCUR-VER-02 PIC 9999   USAGE COMP.
    03  FILLER                  PIC XX.
    03  UCE-ACTION-CODE-02      PIC XXXX.
    03  UCE-RETURN-CODE-02      PIC S9(8)  USAGE COMP.
    03  UCE-MSG-CODE-02         PIC 9(7)   USAGE COMP-3.
    03  UCE-MSG-TEXT-POINTER-02 PIC S9(8)  USAGE COMP.
    03  FILLER                  PIC X(32).
    03  UCE-SYS-INIT-TIME-02    PIC S9(8)  USAGE COMP.  DO NOT MODIFY
```

```
03 UCE-FROM-TASK-02       PIC X(8).          DO NOT MODIFY
03 UCE-ACTIVE-TASK-02     PIC X(8).          DO NOT MODIFY
03 UCE-NEXT-TASK-02       PIC X(8).          DO NOT MODIFY
03 UCE-ENTRY-TASK-02      PIC X(8).          DO NOT MODIFY
03 UCE-PT-LIST-POINTER-02  PIC S9(8)  USAGE COMP.  DO NOT MODIFY
03 UCE-NBR-TASKS-02       PIC S9999  USAGE COMP.  DO NOT MODIFY
03 UCE-NBR-SESSIONS-02     PIC S9999  USAGE COMP.  DO NOT MODIFY
03 UCE-QUEUE-ID-02        PIC S9(8)  USAGE COMP.  DO NOT MODIFY
03 UCE-SESSION-DESCR-02    PIC X(16).
03 UCE-CURR-TASK-FLAG-02   PIC X.
              88 UCE-SUSPEND-02  VALUE 'S'.
              88 UCE-END-02      VALUE 'O'.
              88 UCE-CONVERSE-02 VALUE 'P'.
03 UCE-NEXT-TASK-FLAG-02   PIC X.
              88 UCE-NEW-02     VALUE 'N'.
              88 UCE-RESUME-02 VALUE 'O'.
```

- Your program is responsible for saving its own variable storage in the form of a queue or a scratch record when performing **suspend** processing. Suspend processing can be either implicit or explicit:
  - A TCF user **implicitly** suspends an application by switching to another TCF application.
  - A TCF user **explicitly** suspends an application by issuing a SUSPEND command.

    Additionally, a TCF user can suspend an entire TCF session by issuing a SWITCH SUSPEND command.

Defining a TCF Task to the DC System

To make your task eligible to run under TCF, you need to use the system generation TASK statement to define the task code that invokes your program under TCF. Include the following parameters:

- TCF TASK IS TCF- Enables your task to run under TCF
- PRODUCT CODE IS- Identifies a generic TCF task code for your task

For example:

```
TASK EMPTSKT INVOKES EMPPRG INPUT SAVE
   TCF TASK IS TCF
   PRODUCT CODE IS EMPTSK.
```

TCF tasks must be defined at system generation; they cannot be defined dynamically.

For more information about system generation, see *CA IDMS System Generation Guide*.

Using the UCE for Communication Under TCF

You use fields in the UCE:

- To communicate with TCF
- To communicate with other applications running under TCF

Communicating with TCF

When your program begins, it checks certain UCE fields to determine invocation conditions. You perform processing based on how your program is invoked.

When your program ends, it sets UCE fields to tell TCF what to do next; for example, whether to switch to another application, perform a pseudoconverse, suspend the TCF session, or end the TCF session.

Use the following fields in the UCE to communicate with TCF:

- UCE-IDENT-02 indicates whether your program is currently running under TCF. If it is, this field is equal to the value UMBR.
- UCE-SESSION-DESCR-02 is used as a queue or scratch record ID. Use this ID to retrieve the variable storage from your application's previously suspended TCF session.
- UCE-NEXT-TASK-02 specifies the task to which TCF should switch.
- UCE-CURR-TASK-FLAG-02 is the flag you set to indicate to TCF whether to suspend a session, end a session, or begin a pseudoconverse.
- UCE-NEXT-TASK-FLAG-02 is the flag you set to indicate to TCF and to other applications whether to begin a new session or restart an old session.
- UCE-MSG-CODE-02, UCE-RETURN-CODE-02, and UCE-MSG-TEXT-POINTER-02 are used to process errors under TCF.

Communicating with Other Applications

You can pass data to and receive data from other applications that run under TCF. For example, you could pass a schema name, a subschema name, syntax, or input parameters.

The UCE provides different fields for different kinds of data:

- To pass a schema name, use the UCE-SCHEMA-NAME-02 field. Optionally, include a version number by using the UCE-SCHEMA-VER-02 field.
- To pass a subschema name, use the UCE-SUBSCHEMA-NAME-02 field. Optionally, include a version number by using the UCE-SUBSCHEMA-VER-02 field.

- To pass large amounts of data (33 bytes or more), use the UCE-INPUT-POINTER-02 field. UCE-INPUT-LENGTH-02 specifies the input data length.

  You can also use the UCE-OUTPUT-POINTER-02 field. UCE-OUTPUT-LENGTH-02 specifies the output data length.

- To pass small amounts of data (32 bytes or less), use the UCE-ENTITY-OCCURRENCE-02 field. In some situations, you may need to include a version number by using the UCE-ENTITY-OCCUR-VER-02 field.

Determining if TCF Is Active

If TCF has invoked the task, the UCE-IDENT-02 field contains the literal UMBR. If UMBR is not present, the session was not invoked by TCF.

COBOL Example

For example, in COBOL:

```
PROCEDURE DIVISION USING UNIVERSAL-COMM-ELEMENT.
  IF UCE-IDENT-02 NOT = 'UMBR'
    THEN GO TO A100-NON-TCF-SESSION.
```

Assembler Example

Assembler programs check register 1 to determine if they are being invoked under TCF. Register 1 points to a one-entry parameter list that points to the UCE. If TCF has invoked the task, the first four bytes of the UCE contain the literal UMBR. For example:

```
    LTR   R1,R1                  UNDER TCF?
    BZ    NONTCF                 NO, GO ON
    L     R2,0(R1)               MAYBE
    CLC   0(4,R2),=CL4'UMBR'     CHECK FOR 'UMBR'
    BNE   NONTCF                 NO, GO ON
+  *** TCF PROCESSING ***
```

Starting a New Session

To start a new session under TCF, perform the following steps:

1. Check the following fields for data:

   ■ UCE-INPUT-POINTER-02 typically points to syntax to be used to start a new session of your application. To determine the data length, refer to UCE-INPUT-LENGTH-02.

   ■ UCE-ENTITY-OCCURRENCE-02 contains an entity name to be used to start a new session of your application. To determine the version number, refer to UCE-ENTITY-OCCUR-VER-02.

   If either of these fields contains data, you should start a new session as specified by the passed data.

2. Check the UCE-NEXT-TASK-FLAG-02 field and perform processing as follows:

   ■ If **N** (new) is specified, you should start a new session using the DBNAME and NODENAME fields from the UCE.

   ■ If **O** (old) is specified, you should resume the previously suspended session as explained in Resuming a Suspended Session.

3. Perform processing, as required.

4. When processing is complete, move 'P' (pseudoconverse) to UCE-CURR-TASK-FLAG-02 to indicate to TCF that a pseudoconverse is to take place.

5. Issue a DC RETURN statement.

Resuming a Suspended Session

To resume a previously suspended session, perform the following steps:

1. Use the session descriptor (UCE-SESSION-DESCR-02) as the queue or scratch ID to retrieve the variable storage needed to resume the session. If the storage area cannot be found, perform the following steps:

   a. Move +4 to UCE-RETURN-CODE-02.

   b. Issue a DC RETURN statement.

   This returns control to TCF and displays an error message on the TCF Error Message screen.

2. Perform processing, as required.

3. When processing is complete, move 'P' (pseudoconverse) to UCE-CURR-TASK-FLAG-02 to indicate to TCF that a pseudoconverse is to take place.

4. Issue a DC RETURN statement.

Processing a Pseudoconverse

At the beginning of a pseudoconverse, you should check a user-defined map field for the following:

1. Does the TCF user want to **suspend** the session?
2. Does the TCF user want to **quit** the session?
3. Does the TCF user want to **switch** to another application that runs under TCF?

If none of the above is specified, you should perform processing, as required.

Suspend Processing

If the TCF user wants to suspend a session, perform the following steps:

1. Move the name of the session descriptor to UCE-SESSION-DESCR-02.
2. Save program variable storage as either a scratch or a queue record using the session descriptor as the scratch or queue ID.
3. Move 'S' (suspend) to UCE-CURR-TASK-FLAG-02.
4. Issue a DC RETURN statement.

End Processing

If the TCF user wants to end a session, perform the following steps:

1. Move 'O' (off) to UCE-CURR-TASK-FLAG-02.
2. Issue a DC RETURN statement.

Switch Processing

If the TCF user issues any form of the SWITCH command, perform the following steps:

1. Move the name of the session descriptor to UCE-SESSION-DESCR-02.
2. Save program variable storage as either a scratch or a queue record using the site-standard session descriptor as the scratch or queue ID.

3. Perform the following steps:
   - **If no task code or product code is specified**, move spaces to UCE-NEXT-TASK-02.
   - **If a task code or product code is specified**, move that code to UCE-NEXT-TASK-02.

     If the TCF user specifies the NEW option of the SWITCH command, move 'N' (new) to UCE-NEXT-TASK-FLAG-02; otherwise move 'O' (old) to UCE-NEXT-TASK-FLAG-02.

     Note: If the TCF user specifies new, you may need to pass data to the switched-to task using either the UCE-INPUT-POINTER-02 or UCE-ENTITY-OCCURRENCE-02 fields in the UCE or some other site-standard convention.
4. Issue a DC RETURN statement.

Displaying Error Messages

At times, you may want to intentionally abend your task by issuing a WRITE LOG statement with a given severity code. To do this under a TCF, perform the following steps:

1. Move the message number to UCE-MSG-CODE-02.
2. Move -1 to UCE-RETURN-CODE-02.
3. Issue a GET STORAGE statement to acquire the storage that is to contain the message text.
4. Issue a WRITE LOG statement. Include the RETURN TEXT option and specify the previously acquired storage.
5. Move the address of the message text into UCE-MSG-TEXT-POINTER-02.

   Note: COBOL programs can do this by calling an Assembler subroutine.
6. Issue a DC RETURN statement.

   This returns control to TCF and displays the error message on the TCF Error Message screen.

Sample Application Under TCF

The program below performs processing that enables it to run under TCF.

This program checks TCF-related fields in the UCE and performs TCF processing before performing any application-specific processing.

```
WORKING-STORAGE SECTION.
01  WS-START                  PIC X(10)  VALUE '*WS START*'.
01  USER-IDENT.
   05 USER-ID-FIRST-EIGHT     PIC X(8).
   05 USER-ID-REST            PIC X(24).
01  TASK-ID                   PIC X(8).
01  SESSION-DESC-WORK.
   05 SDW-1                   PIC X(8).
   05 SDW-2                   PIC X(8).

01  TCF-REC.
   02  TCF-REC-COMMLINE       PIC X(7).
                  88  SUS-COMMAND VALUE 'SUS'
                    'SUSP' 'SUSPE' 'SUSPEN'
                    'SUSPEND'.
                  88  BYE-COMMAND VALUE 'BYE'
                    'QUIT' 'QUI' 'END'.
                  88  SWITCH-COMMAND VALUE 'SWI'
                    'SWIT' 'SWITC' 'SWITCH'.
   02  TCF-REC-QUIT           PIC X VALUE '_'.
   02  TCF-REC-SUSPEND        PIC X VALUE '_'.
   02  TCF-REC-SWITCH         PIC X VALUE '_'.
   02  TCF-REC-HELP           PIC X VALUE '_'.
   02  TCF-REC-SWI-TASK       PIC X(8).
   02  TCF-REC-OLDNEW         PIC X.
                  88  SWI-OLD VALUE 'O'.
                  88  SWI-NEW VALUE 'N'.

01  DATA-REC.
   02  DATA-REC-FIELD1        PIC X VALUE '_'.
   02  DATA-REC-FIELD2        PIC X VALUE '_'.
   02  DATA-REC-FIELD3        PIC X VALUE '_'.
   02  DATA-REC-FIELD4        PIC X VALUE '_'.
01  WS-END                    PIC X(8)  VALUE '*WS END*'.
LINKAGE SECTION.
01  COPY IDMS UNIVERSAL-COMM-ELEMENT VERSION 2.
01  UNIVERSAL-COMM-ELEMENT.
    03  UCE-IDENT-02          PIC XXXX.
    03  UCE-DBNAME-02         PIC X(8).
```

```
    03 UCE-NODE-NAME-02      PIC X(8).
    03 UCE-DICT-NAME-02      PIC X(8).
    03 UCE-DICT-NODE-02      PIC X(8).
    03 UCE-SCHEMA-NAME-02    PIC X(8).
    03 UCE-SCHEMA-VER-02     PIC 9999   USAGE COMP.
    03 UCE-SUBSCHEMA-NAME-02 PIC X(8).
    03 UCE-SUBSCHEMA-VER-02  PIC 9999   USAGE COMP.
    03 UCE-INPUT-POINTER-02  PIC S9(8)  USAGE COMP.
    03 UCE-INPUT-LENGTH-02   PIC S9(8)  USAGE COMP.
    03 UCE-OUTPUT-POINTER-02 PIC S9(8)  USAGE COMP.
    03 UCE-OUTPUT-LENGTH-02  PIC S9(8)  USAGE COMP.
    03 UCE-ENTITY-OCCURRENCE-02
                    PIC X(32).
    03 UCE-ENTITY-OCCUR-VER-02
                    PIC 9999   USAGE COMP.
    03 FILLER               PIC XX.
    03 UCE-ACTION-CODE-02    PIC XXXX.
    03 UCE-RETURN-CODE-02    PIC S9(8)  USAGE COMP.
    03 UCE-MSG-CODE-02       PIC 9(7)   USAGE COMP-3.
    03 UCE-MSG-TEXT-POINTER-02
                    PIC S9(8)  USAGE COMP.

    03 FILLER               PIC X(32).
    03 UCE-SYS-INIT-TIME-02  PIC S9(8)  USAGE COMP.
    03 UCE-FROM-TASK-02      PIC X(8).
    03 UCE-ACTIVE-TASK-02    PIC X(8).
    03 UCE-NEXT-TASK-02      PIC X(8).
    03 UCE-ENTRY-TASK-02     PIC X(8).
    03 UCE-PT-LIST-POINTER-02 PIC S9(8)  USAGE COMP.
    03 UCE-NBR-TASKS-02      PIC S9999  USAGE COMP.
    03 UCE-NBR-SESSIONS-02   PIC S9999  USAGE COMP.
    03 UCE-QUEUE-ID-02       PIC S9(8)
                    USAGE COMP.
    03 UCE-SESSION-DESCR-02  PIC X(16).
    03 UCE-CURR-TASK-FLAG-02 PIC X.
                  88 UCE-SUSPEND-02 VALUE 'S'.
                  88 UCE-END-02 VALUE 'O'.
                  88 UCE-CONVERSE-02 VALUE 'P'.
    03 UCE-NEXT-TASK-FLAG-02 PIC X.
                  88 UCE-NEW-02 VALUE 'N'.
                  88 UCE-RESUME-02 VALUE 'O'.
PROCEDURE DIVISION USING UNIVERSAL-COMM-ELEMENT.
MAIN-LINE.
```

```
***                          CHECK FOR TCF SESSION
    IF UCE-IDENT-02 NOT = 'UMBR'
      THEN GO TO C100-SESSION.
***                          MOST LIKELY PSEUDO-CONV
    IF UCE-CONVERSE-02
      THEN GO TO A100-PSEUDOCONVERSE.
***                          NOT PCONV, DATA SENT?
    IF UCE-INPUT-POINTER-02 NOT = 0 OR
      UCE-ENTITY-OCCURRENCE-02 NOT = SPACES
    THEN
      GO TO A100-START-WITH-DATA.
***                          NEW SESSION SPECIFIED?
    IF UCE-NEW-02
      THEN GO TO A100-START-NEW-SESSION
***                          ELSE DEFAULT TO OLD
    ELSE
      GO TO A100-START-OLD-SESSION.

 A100-PSEUDOCONVERSE.
    BIND MAP TCFMAP01.
    BIND MAP TCFMAP01 RECORD TCF-REC.
    BIND MAP TCFMAP01 RECORD DATA-REC.
    ACCEPT USER ID INTO USER-IDENT.
    ACCEPT TASK ID INTO TASK-ID.
***                      MENU OR COMMAND-LINE SUSPEND
    IF (TCF-REC-SUSPEND NOT = '_')
     OR SUS-COMMAND
    THEN
       MOVE USER-ID-FIRST-EIGHT TO SDW-1.
       MOVE TASK-ID          TO SDW-2.
       MOVE SESSION-DESC-WORK   TO UCE-SESSION-DESCR-02.
***                      USE SESS-DESCRIPTOR FOR QID
       PERFORM U100-SAVE-STORAGE
       MOVE 'S' TO UCE-CURR-TASK-FLAG-02
       DC RETURN.

***                      MENU OR COMMAND-LINE QUIT
    IF (TCF-REC-QUIT NOT = '_')
     OR BYE-COMMAND
    THEN
       MOVE 'O' TO UCE-CURR-TASK-FLAG-02
       DC RETURN.
```

```
***                    MENU OR COMMAND-LINE SWITCH
    IF (TCF-REC-SWI-TASK NOT = SPACES)
     OR SWITCH-COMMAND
    THEN
      PERFORM B100-SWITCH
    ELSE
      MOVE 'P' TO UCE-CURR-TASK-FLAG-02
      GO TO C100-SESSION.
*
 A100-START-WITH-DATA.
*** START SESSION USING THE DATA PASSED IN        ***
*** UCE-INPUT-POINTER-02 OR UCE-ENTITY-OCCURRENCE-02 ***
*
 A100-START-NEW-SESSION.
*** START A NEW SESSION, MOVE 'P' ***
*** TO UCE-CURR-TASK-FLAG-02      ***
*
 A100-START-OLD-SESSION.
*** RESTART OLD SESSION, GET PREVIOUS VARIABLE ***
*** STORAGE FROM SCRATCH OR QUEUE AND MOVE 'P' ***
*** TO UCE-CURR-TASK-FLAG-02.                  ***
*** IF UCE-SESSION-DESCR-02 IS EMPTY, OR IF    ***
*** GET QUEUE/SCRATCH FAILS, MOVE +4 TO        ***
*** UCE-RETURN-CODE-02 AND ISSUE A DC RETURN   ***
*
    IF UCE-SESSION-DESCR-02 = SPACES
      MOVE +4 TO UCE-RETURN-CODE-02
      DC RETURN.
    GET QUEUE ID UCE-SESSION-DESCR-02
          FROM WS-START
          TO WS-END
          RETENTION 7
      ON ANY-ERROR-STATUS
        MOVE +4 TO UCE-RETURN-CODE-02
        DC RETURN.
    MOVE 'P' TO UCE-CURR-TASK-FLAG-02.
    GO TO C100-SESSION.

*
 B100-SWITCH.
    MOVE USER-ID-FIRST-EIGHT TO SDW-1.
    MOVE TASK-ID           TO SDW-2.
    MOVE SESSION-DESC-WORK   TO UCE-SESSION-DESCR-02.
```

```
*                        USE SESS-DESCRIPTOR FOR QID
    PERFORM U100-SAVE-STORAGE.
    IF TCF-REC-SWI-TASK = SPACES
      THEN MOVE SPACES TO UCE-NEXT-TASK-02
          DC RETURN.
    MOVE TCF-REC-SWI-TASK TO UCE-NEXT-TASK-02.
    IF SWI-NEW THEN
      MOVE 'N' TO UCE-NEXT-TASK-FLAG-02
    ELSE
      MOVE 'O' TO UCE-NEXT-TASK-FLAG-02.
    DC RETURN.
*
 C100-SESSION.
*** PROGRAM PROCESSING ***
*
 U100-SAVE-STORAGE.
*** SAVE WORKING STORAGE FROM WS-START TO WS-END ***
*** IN THIS EXAMPLE, ITS A QUEUE RECORD          ***
    PUT QUEUE ID UCE-SESSION-DESCR-02
            FROM WS-START
            TO WS-END
            RETENTION 7.
 IDMS-ABORT.
 IDMS-ABORT-EXIT.
    EXIT.
    COPY IDMS IDMS-STATUS.
```

Appendix F: Services Batch Interface

This section contains the following topics:

About the 10.2 Services Batch Interface (see page 355)

About the 10.2 Services Batch Interface

Batch programs that require CA IDMS 10.2 services only can use the optional 10.2 services batch interface to access a later release.

Since only 10.2 features are available through this interface, later release features such as SYSIDMS parameters and SQL access are not supported through this interface.

This appendix describes the requirements for using the 10.2 services batch interface.

CA IDMS Installation

The 10.2 services batch interface requires two load modules supplied by the CA IDMS installation process:

- IDML
- B102STUB

Usage

Before you can make use of the interface, z/OS users must copy the B102STUB load module from the appropriate release level CAGJLOAD load library into a load library which will be included in their run-time STEPLIB concatenation and rename the module to IDMSB102.

z/VSE users need to copy the B102STUB.PHASE from their installed CA IDMS sub-library into a sub-library which will be included in their run-time LIBDEF concatenation, and then rename the module to IDMSB102.PHASE.

To use the 10.2 services batch interface, the following conditions must be met:

- The z/OS batch job JCL includes a STEPLIB that contains IDMSB102
- The z/VSE batch job JCL includes a LIBDEF concatenation which contains the IDMSB102 PHASE.
- The batch program is linked with either IDMS (Release 10.2) or IDML
- If the batch program has been relinked with a later version of the IDMS module, it must be relinked with either the 10.2 IDMS module or the IDML module

  Note: A link with IDMSINTB is supported for upward compatibility but is neither required nor recommended for using the 10.2 batch services interface.
- DBNAME must be specified. Since SYSIDMS parameters are not supported through this interface, you can do one of the following:
  - Modify the BIND RUN-UNIT statements in the program to specify the DBNAME parameter; for example:

    ```
    BIND RUN-UNIT DBNAME EMPDEMO.
    ```
  - Update the DBTABLE within the central version to utilize subschema mapping and default dbname parameters; for example:

    ```
    DBNAME *DEFAULT
     SUBSCHEMA EMPSS?? MAPS TO EMPSS?? USING DBNAME EMPDEMO;
    ```

    For more information about updating the DBTABLE, see the *CA IDMS Database Administration Guide*.

Batch Execution Considerations

Be aware of these considerations when preparing a program to use one of the batch interfaces:

- If the batch program is linked with either IDMS (Release 10.2) or IDML, and IDMSB102 is in a batch program JCL STEPLIB (z/OS) or LIBDEF (z/VSE), the 10.2 services batch interface is *always* used.
- If the batch program is linked with any later version of the IDMS module, the 10.2 services batch interface will *never* be used, even if IDMSB102 is in a batch program JCL STEPLIB (z/OS) or LIBDEF (z/VSE).

- To use a later version of the batch interface with an existing 10.2 program, be sure that IDMSB102 is not in a batch program JCL STEPLIB (z/OS) or LIBDEF (z/VSE).
- If the COBOL program has been compiled with the DYNAM option, you must rename the IDML module to IDMS and place it in a separate library. This library must be the first library after the STEPLIB concatenation.
- If signon security is in effect, a valid user ID must be provided for the batch user exit BTCIDXIT, which allows specification of the user ID to be checked by security. A sample BTCIDXIT may be found in the distribution source library.
- If the batch program is linked with either IDMS (Release 10.2) or IDML, then IDMSIN01 functions are not supported and the new items of the delivered IDMS-STATUS routine (dbkey, page group and dbkey format) are not displayed.

Index

A

ABEND • 290
abend control element • 324
ACCEPT (DC) • 262
ACCEPT statements
- BIND ADDRESS • 140
- DATABASE-STATISTICS • 171
- db-key • 135
- general discussion • 134
- page information • 139
- PROCEDURE CONTROL LOCATION • 303

access modes • 180
AID • 198, 257, 300
APPC • 13
application dictionary • 15
application server • 13
area
- description • 43, 47
- locks • 180
- sweeping • 123
- usage modes • 181

Assembler • 335
- batch error checking • 335
- DC error checking • 335

asynchronous • 299
attribute byte
- definition • 193

automatic set membership • 145

B

basic mode • 299
batch execution • 310, 355
batch mapping compiler • 193
batch, 10.22 services batch interface • 355
BDAM blocks • 48
bill-of-materials
- description • 42
- retrieving • 161
- storing • 160

BIND statements
- BIND PROCEDURE • 302
- BIND RUN-UNIT • 310, 355
- BIND TASK • 337

BL listing • 324
BTCIDXIT user exit • 355

C

CA ADS • 297
CA IDMS
- about • 13

CA OLQ • 320
CALC location mode • 56
central version • 310
- access modes • 180
- area locks • 180
- concurrent area use • 181
- database name table • 311
- description • 14
- implicit record locks • 181

chain set linkage options • 23
CHANGE PRIORITY • 288
CHECK TERMINAL • 301
checkpoints • 183
COBOL
- VS COBOL II • 339

command list • 314
COMMIT
- frequency of use • 183
- implicit • 183
- TASK • 249, 337

common system area • 324
common work area • 236
compiling • 309
CONNECT • 155
control statements
- COMMIT • 183

currency • 112
- queue records, in • 249
- scratch records, in • 241

D

Data Definition Language (DDL) • 44
Data Manipulation Language (DML) • 14
data structure diagram • 59
database
- collecting statistics • 171
- concepts • 17
- description • 44
- name table • 311

physical attributes • 47
transactions • 183
database key
deadlocks • 291
description • 49
page information • 110
retrieving records • 128
run unit currencies • 112
saving • 135
set relationships • 55
transmitting screen data • 253
database procedures
ACCEPT PROCEDURE CONTROL LOCATION • 303
BIND PROCEDURE • 302
DBNAME • 310
DC
abend processing • 290
Assembler considerations • 335
current time and date • 275
data integrity • 264
DC-BATCH • 337
event control blocks • 294
journal file • 277
messages • 281
modifying task priority • 288
nonterminal tasks • 288
programming techniques • 255
statistics • 279
table management • 272
task priority • 288
task-related information • 262
terminal management • 193, 299
writing to a printer • 285
DC RETURN • 257
DC-BATCH • 249, 289, 337
DCMT
VARY DYNAMIC PROGRAM • 272, 314
VARY DYNAMIC TASK • 314
VARY PROGRAM NEW COPY • 314
VARY QUEUE • 249
DCUF statements • 310
SET DBNAME • 313
SET DICTNAME • 314
TEST • 314
DDLDCMSG • 281
DDLDCRUN • 249
DDLDCSCR • 241
deadlock prevention • 304
debugging • 317
CA OLQ menu facility • 320
dump reading • 321
error checking • 327
PL/I • 330
using the trace facility • 317
DICTNAME • 310
DIRECT location mode • 59
storing • 145
with FIND/OBTAIN DB-KEY • 128
DISCONNECT • 156
dispatch control element • 324
DMAP listing • 324
DMCL • 47
dump reading • 321
dynamic program definition • 314
dynamic task definition • 314

E

ERASE • 150
error checking • 327
error messages
for maps • 204
suppressing display of • 204
event control block • 294, 301

F

files
description • 48
FIND/OBTAIN statements
CALC • 116
general discussion • 115
OWNER • 126
USING DB-KEY • 128
WITHIN AREA • 123
WITHIN SET • 117
WITHIN SET USING SORT KEY • 119
FINISH TASK • 339

G

generic key searches • 119, 131
GET TIME • 275

H

hierarchical relationships • 32

I

identical data

testing for • 205
IDML load module • 355
IDMS statistics block • 183
IDMSB102 load module • 355
IDMSIN01 entry point • 310, 317
IF
IF EMPTY • 141
IF MEMBER • 142
index key • 29
INDEX pointer • 31
indexed sets
description • 29
linkage options • 31
retrieval commands • 131
INQUIRE MAP statement • 198
DIFFERENT parameter • 205
IDENTICAL parameter • 205
Integrated Data Dictionary (IDD) • 15

J

journal file • 277
junction record • 41

K

KEEP • 166
KEEP LONGTERM • 264
explicit locks • 265
monitoring records • 268

L

line mode • 223
link editing • 309
load library
test • 314
local mode • 310
area locks • 180
description • 14
location mode • 56
locks
area • 180
DC • 264
effect on run units • 175
exclusive • 175
explicit • 175
explicit (online) • 265
implicit • 175, 183
longterm locks • 268
record • 175
shared • 175
logical terminal element • 233, 236, 241, 324

M

manual set membership • 142, 155
MAP IN • 198
MAP OUT • 196
MAP OUTIN • 203
map request block • 195
mapping mode
asynchronous output requests • 301
detail area • 207
footer area • 207
general discussion • 193
header area • 207
housekeeping statements • 195
INQUIRE MAP • 198
MAP IN • 198
MAP OUT • 196
MAP OUTIN • 203
MODIFY MAP • 202
maps
error messages for • 204
messages • 281
with maps • 196
mixed page groups • 128
modified data tag • 208
retrieval example • 212
update example • 214
MODIFY • 148
MODIFY MAP • 202
multiple set ownership • 35

N

navigational DML programming
currency • 112
networks • 38
NEXT pointer • 23
NODENAME • 310
nonterminal tasks
ATTACH • 288
external request • 289
queue threshold • 290
SET TIMER • 289

O

occurrences
description • 17

online debugger
PL/I considerations • 330
sample PL/I session • 332
online mapping • 193
automatic editing • 198
operating modes
DC-BATCH • 289
optional set membership • 142, 155
OWNER pointer • 23

P

PAGE-INFO parameter • 128
pages
description • 47
PL/I
online debugger • 330
TRANSFER CONTROL • 329
precompiler • 309
PRIOR pointer • 23
program definition element • 324
program management • 256
program pools
tables • 272

Q

queue management • 249
COMMIT TASK • 249
deleting • 249
header record • 249
locking • 249
retention period • 249
retrieving • 249
storing • 249
queued resources • 304
queues
print • 285

R

record locks
description • 175
online environment • 265, 268
set implicitly by DML verbs • 168
records
CALC location mode • 56
data structure diagram • 59
description • 19
DIRECT location mode • 59
junction record • 41
location mode • 56
storing • 145
structure • 50
VIA location mode • 57
relationships
as sets • 32
description • 17
Release 10.2 • 355
restricting access
area usage mode • 181
KEEP • 166
KEEP LONGTERM • 265
retrieval statements
FIND/OBTAIN CALC • 116
FIND/OBTAIN OWNER • 126
FIND/OBTAIN USING DB-KEY • 128
FIND/OBTAIN WITHIN AREA • 123
FIND/OBTAIN WITHIN SET • 117
FIND/OBTAIN WITHIN SET USING SORT KEY • 119
general discussion • 115
indexed records • 131
indexed sets • 131
RETURN • 131
RETURN • 131
ROLLBACK • 335
TASK • 337
run unit
general discussion • 173
terminated from db-key deadlocks • 291

S

saving I/O
FIND/OBTAIN DB-KEY • 128
IF • 140
RETURN • 131
saving page information • 139
schemas • 44
scratch area • 241
scratch management • 241
logical terminal element • 241
record ID • 241
task control element • 241
scratch record ID • 241
secondary dictionary • 314
see=sessionattributes attributes • 310
see=sessionattributes profile attributes • 310
segments
description • 47

SEND MESSAGE • 284
session attributes • 310
SET ABEND EXIT • 293
sets
- data structure diagram • 59
- description • 20
- hierarchies • 32
- linkage • 55
- linking new members • 24
- membership options • 31
- mode options • 23
- multiple set ownership • 35
- networks • 38
- relationships • 32, 55
- sorted • 119

signon processing • 310
signon security • 355
sort keys
- contiguous • 119
- noncontiguous • 119
- retrieving sorted records • 119
- RETURN • 131

statistics
- database • 171
- DC • 279

storage management • 229
- shared • 235
- shared kept • 236
- user • 231
- user kept • 233

STORE • 145
subroutines
- ACCEPT BIND ADDRESS • 140

subschema
- default usage modes • 181
- mapping • 355
- overriding • 311

symbolic key
- contiguous • 119
- noncontiguous • 119

synchronous • 299
SYSIDMS parameters • 310, 317
system dictionary • 15
System Network Architecture • 299

T

task code • 262
task control element • 229, 233, 241
TCP/IP • 13
teleprocessing monitor • 13
terminal management • 193
- basic mode • 299
- line mode • 223
- mapping mode • 193

test load library • 314
trace facility • 317
transaction statistics block • 279
TRANSFER CONTROL
- LINK • 260
- LINK from CA ADS • 297
- LINK under TCF • 341
- PL/I considerations • 329
- XCTL • 258

transfer control facility • 341
- sample program • 348

types
- description • 17

U

universal communications element • 341
updating the database
- connecting records • 155
- disconnecting records • 156
- erasing records • 150
- general discussion • 145
- modifying records • 148
- storing records • 145

usage modes
- area • 181
- ready options • 181

V

VIA location mode • 57
VSAM
- CONNECT restriction • 155
- control intervals • 48
- DISCONNECT restriction • 156
- MODIFY (ESDS and KSDS) • 148
- restrictions with FIND/OBTAIN DB-KEY • 128
- restrictions with IF • 140
- STORE (RRDS) • 145

W

walking a set • 117
write control character • 202
- definition • 193

WRITE JOURNAL • 277
WRITE LOG • 281
WRITE PRINTER • 285
    DC-BATCH • 337

XA considerations
    assembler • 339
    VS COBOL II • 339