Product Information Announcement

o New Release [● Revision] o Update o New Mail Code

Title

MCP/AS InfoExec™ Semantic Information Manager (SIM) Object Manipulation Language (OML) Programming Guide (8600 0163–102)

This announces a retitling and reissue of the *ClearPath HMP NX and A Series InfoExec™ Semantic Information Manager (SIM) Object Manipulation Language (OML) Programming Guide*. No new technical changes have been introduced since the HMP 1.0 and SSR 43.2, release in June 1996.

To order a Product Information Library CD-ROM or paper copies of this document

- United States customers, call Unisys Direct at 1-800-448-1424.
- Customers outside the United States, contact your Unisys sales office.
- Unisys personnel, order through the electronic Book Store at http://iwww.bookstore.unisys.com.

Comments about documentation can be sent through e-mail to **doc@unisys.com**.

Announcement only:	Announcement and attachments:		
	AS123	System:	MCP/AS
		Release:	HMP 4.0 and SSR 45.1
		Date:	June 1998
		Part number:	8600 0163–102

MCP/AS

UNISYS

InfoExec™ Semantic Information Manager (SIM) Object Manipulation Language (OML)

Programming Guide

HMP 4.0 and SSR 45.1 June 1998

Printed in USA

Priced Item 8600 0163–102

Contents

Section 5. Structuring Update Queries

Section 6. Writing Expressions for Retrieval and Update Queries

Figures

Tables

Examples

About This Guide

Purpose

This guide describes how to use the Object Manipulation Language (OML) to form retrieval and update queries against a Semantic Information Manager (SIM) database.

You can also use the information provided in this guide to retrieve data from the following:

- Data Management System II (DMSII) databases that have been processed by DMS.View
- Logic and Information Network Compiler II (LINC II) databases that have been processed by LINC.View
- Structured Query Language Database (SQLDB) databases
- Files defined in an Advanced Data Dictionary System (ADDS) data dictionary

This guide also discusses two methods for submitting queries: using the OML interface provided in the InfoExec Interactive Query Facility (IQF) and using the SIM library entry points.

Scope

This guide provides the information required to use OML to create a query and to submit the query against a database.

This guide also discusses two methods for processing queries. One method uses IQF, and the other method involves embedding the OML in a program you write that calls the SIM library.

Note: *You can use the SIM library interface to process SQLDB data manipulation language (DML) queries against an SQLDB database by embedding your SQLDB query text in a program you write that calls the SIM library.*

You can also submit queries against a SIM database by using one of the programmatic interfaces provided in ALGOL, COBOL74, and Pascal. For more information on this method, refer to the appropriate user language manual and the *InfoExec Semantic Information Manager (SIM) Programming Guide.*

Audience

This guide is addressed to the following audiences:

- Experienced users of IQF
- Programmers familiar with the concepts of a SIM database

Prerequisites

To use this guide, you need to understand the structure of SIM databases in terms of their entities, classes, and attributes. You should also be familiar with the concepts associated with querying against SIM databases. The *InfoExec Semantic Information Manager (SIM) Object Definition Language (ODL) Programming Guide* and the *InfoExec Semantic Information Manager (SIM) Technical Overview* provide the background information you might require.

How to Use This Guide

The first two sections of this guide introduce a SIM database and general information describing how to process a query. Sections 3 and 4 provide information on developing retrieval queries, and Section 5 is helpful when you are developing update queries. Section 6 provides more detailed reference material required for creating a query, and Sections 7 and 8 explain how to use the SIM library interface.

You should read this guide serially. If you are using the SIM library entry points to process your query, you should also read Sections 7 and 8; otherwise, you can ignore this material.

The examples provided in this guide are based on the ORGANIZATION database. The schema for this database is provided in Appendix C. Refer to this appendix when reading the examples in this guide.

Appendix F provides the syntax for all OML queries in quick-reference format.

Railroad diagrams are a method of depicting syntax. Appendix G explains how to read railroad diagrams.

Acronyms are spelled out on their first occurrence in the guide.

Organization

This guide is divided into eight sections and seven appendixes. In addition, an index appears at the end of this guide.

Section 1. Introduction to OML

This section provides an overview of the structure of a SIM database and to using OML to form retrieval and update queries as well as expressions. This section also discusses the methods of submitting queries against a SIM database.

Section 2. OML Language Conventions

This section describes the syntactical conventions that you must follow when you create a query. The discussion includes the rules for qualifying class and attribute names.

Section 3. Structuring Retrieval Queries

This section provides a skeleton format for generating a retrieval query and explains the purpose of each portion of the skeleton format.

Section 4. Formatting the Output from Retrieval Queries

This section describes the two forms of output that you can associate with a retrieval query: tabular and structured output. This section also describes how to select the order in which information is returned by a retrieval query.

Section 5. Structuring Update Queries

This section provides a skeleton format for generating an update query. The differences between the three types of update query—modification, deletion, and addition—are described. The effect of different types of attributes on the syntax and results of an update query also are discussed.

Section 6. Writing Expressions for Retrieval and Update Queries

This section describes the operators and functions that you can use in queries.

Section 7. Using the SIM Library Entry Points

This section provides a task-oriented description of the SIM library entry points.

Section 8. Describing the SIM Library Entry Points

This section provides details of the SIM library entry points in a reference format.

Appendix A. OML Error Messages

This appendix lists and explains some of the messages that might be returned by SIM when an OML query is processed.

Appendix B. Exception Fields and Categories

This appendix lists and explains the exception fields and categories you can return using the SIM library entry points.

Appendix C. Description of the ORGANIZATION Database

This appendix describes the schema of the database used in the examples in this guide.

Appendix D. Sample Retrieval Queries

This appendix presents retrieval queries that illustrate the use of some of the more complex functions supported in OML. The examples are based on the ORGANIZATION database.

Appendix E. Sample Update Queries

This appendix provides a description of the thought process and queries required to add a project, department, project manager, and several project employees to the ORGANIZATION database.

Appendix F. Quick-Reference Query Syntax

This appendix lists in quick-reference format the syntax for the different retrieval and update queries you can create using OML.

Appendix G. Understanding Railroad Diagrams

This appendix describes the conventions used in syntax diagrams.

Related Product Information

Unless otherwise stated, all documents referred to in this publication are MCP/AS documents. The titles have been shortened for increased usability and ease of reading.

The following documents are included with the software release documentation and provide general reference information:

- The *Glossary* includes definitions of terms used in this document.
- The *Documentation Road Map* is a pictorial representation of the Product Information (PI) library. You follow paths through the road map based on tasks you want to perform. The paths lead to the documents you need for those tasks. The Road Map is available on the PI Library CD-ROM. If you know what you want to do, but don't know where to find the information, start with the Documentation Road Map.
- The *Information Availability List* (IAL) lists all user documents, online help, and HTML files in the library. The list is sorted by title and by part number.

Communications Management System (COMS) Programming Guide

The guide explains how to write online, interactive, and batch application programs that run under COMS. This guide is written for experienced applications programmers with knowledge of data communication subsystems.

InfoExec Advanced Data Dictionary System (ADDS) Operations Guide

This guide describes InfoExec ADDS operations, such as creating and managing database descriptions. This guide is written for those who collect, organize, define, and maintain data and who are familiar with the Data Management System II (DMSII), the Semantic Information Manager (SIM), and the Structured Query Language Database (SQLDB).

InfoExec Capabilities Manual

This manual discusses the capabilities and benefits of the InfoExec data management system. This manual is written for executive and data processing management.

InfoExec DMS.View Operations Guide

This guide explains the InfoExec DMS.View utility and provides operating instructions for it. It is written for Data Management System II (DMSII) users who want to create a Semantic Information Manager (SIM) description and a Structured Query Language Database (SQLDB) description of a DMSII database.

InfoExec Interactive Query Facility (IQF) Operations Guide

This guide provides an overview of IQF and information about its inquiring, updating, and report painting capabilities. This guide is written primarily for IQF users with little programming knowledge, but also for programmers and database administrators.

InfoExec LINC.View Operations Guide

This guide provides operating instructions for the InfoExec LINC.View utility. It is written for Logic and Information Network Compiler II (LINC II) users who want to create a Semantic Information Manager (SIM) description and a Structured Query Language Database (SQLDB) description of a LINC II database.

InfoExec Semantic Information Manager (SIM) Object Definition Language (ODL) Programming Guide

This guide presents information on using ODL to define a SIM database. Each kind of ODL declaration that can be included in a schema is described. Use of the SIM Utility to create and maintain a SIM database is also described. This guide is written primarily for applications and systems developers responsible for defining and administering SIM databases.

InfoExec Semantic Information Manager (SIM) Programming Guide

This guide describes how to use language extensions to access InfoExec SIM databases from application programs written in COBOL74, Pascal, and ALGOL. This guide is written for programmers who know at least one of these programming languages thoroughly and who are familiar with SIM.

InfoExec Semantic Information Manager (SIM) Technical Overview

This overview describes the SIM concepts on which the InfoExec data management system is based. This overview is written for end users, applications programmers, database designers, and database administrators.

InfoExec Structured Query Language Database (SQLDB) Programming Guide

This guide details the A Series implementation of the Structured Query Language (SQL). The guide presents information on using SQLDB to define a relational database as well as using SQLDB to manipulate the data in a relational database that has been defined through SQLDB. This guide is written for applications programmers, database designers, and database administrators who are using SQLDB for managing relational databases.

System Messages Support Reference Manual

This manual presents operating system error messages and explains the most likely cause of each error and the most effective response to each message. This manual is written for operators and programmers responsible for system operation, and for the resolution of error conditions on those systems.

Section 1
Introduction to OML

The Object Manipulation Language (OML) is a high-level language you can use to construct retrieval and update queries. You can embed an OML query in a program that calls a Semantic Information Manager (SIM) library, or you can create and process the query, using the InfoExec query facility, Interactive Query Facility (IQF).

Understanding SIM Database Concepts

To create efficient queries, you must understand the concepts behind a SIM database. The following information overviews SIM concepts. The ORGANIZATION database is used as the basis for the examples provided in this manual. Appendix C provides a description of the database. For more SIM database conceptual information, refer to the *InfoExec SIM Technical Overview*.

When defining a query, you must be aware of three structures—entities, classes, and attributes.

Defining Entities

An entity is a representation of any object. For example, in the ORGANIZATION database, the objects described are people, projects, and departments. Each person, project, and department described in the database is considered to be an entity. Furthermore, all the information relating to a person, including any projects he or she is working on and the department he or she works in, is considered to be a part of the entity.

Defining Entities and the Roles They Play

In a SIM database, an entity is described by the role, or roles, it can take. For example, in the ORGANIZATION database, each person described can take one or more of the following roles:

- Person
- Previous employee
- Employee
- Manager
- Interim manager
- Project employee

Each one of these roles is referred to as a *class*.

Besides assigning one or more roles to an entity, you can also describe an entity, using *attributes*. Each attribute describes one characteristic of an entity. For example, age, current residence, personal identification number, and citizenship are all characteristics of a person.

Classes and attributes are described more fully later in this section.

Differentiating between Entities with Similar Descriptions

When a SIM database is populated, each entity is given a unique identifier called a *surrogate*. Even if all the attributes describing two entities have identical values, you can use the surrogate to differentiate between the two entities. For example, two new Rolls Royce cars are identical—they have the same engine, number of doors, color interior, and so on. If you enter information in your database for both cars, SIM supplies a surrogate value to enable you to differentiate between the two cars.

Note: *Because SIM provides the surrogate value, you can enter duplicate information in the database. Or, the designer of a database can prevent duplicate information from being entered by using the* required *and* unique *attribute options. For example, in the ORGANIZATION database each employee must be assigned a unique employee identification number when the employee entity is created.*

Defining Classes

A class is used to group related information that describes one role of an object. In the ORGANIZATION database, the objects described are people, projects, assignments, and departments. The classes used to group information about these objects are the following:

Object	Classes Describing
People	PERSON, PREVIOUS_EMPLOYEE, EMPLOYEE, MANAGER, INTERIM_MANAGER, and PROJECT_EMPLOYEE
Projects	PROJECT
Department	DEPARTMENT
Assignment	ASSIGNMENT

Understanding Superclass/Subclass Relationships

Because objects can take more than one role in a database, for example, a person can be both an employee and a manager, you must be able to define relationships between classes. In a SIM database, classes can have superclass/subclass relationships.

If a class has a subclass, then the subclass describes a subset of the objects described in the class. Consequently, the class is referred to as a superclass in relation to the subclass. For example, there are more employees described in the database than there are managers because while every manager is also an employee, not every employee is a manager. The subclass, then, provides additional information about the object.

The terms *generalization* and *specialization* are also used to define the relationship between classes and subclass. A class contains more *general* information than any of its subclasses. And, a class contains more *specialized* information than any of its superclasses.

In a SIM database, you can access all the information available in a class from that class or from any of the subclasses of that class. In the ORGANIZATION database, the PERSON class has the most general information about the people described in the database. Since all people have a marital status and a gender, you can access this information from the PERSON class. From the EMPLOYEE class, which is a subclass of the PERSON class, you can still get information about marital status and gender. You can also access information regarding employee identification number and salary, which is information that does not apply to all people—this information does not apply to previous employees and children of employees, so the information is not associated with the PERSON class.

There can be several layers of superclass/subclass relationships. For example, in the ORGANIZATION database, the class EMPLOYEE has the superclass PERSON and the subclasses PROJECT_EMPLOYEE and MANAGER. The term *hierarchy* refers to classes that appear in the same chain of superclass/subclass relationships.

Figure 1–1 illustrates the relationship between the PERSON, EMPLOYEE, MANAGER, and PROJECT_EMPLOYEE classes. The numbers in parentheses indicate the number of entities described by each class. The total number of entities described is the number of entities described by the PERSON class (100), as this class has no superclasses.

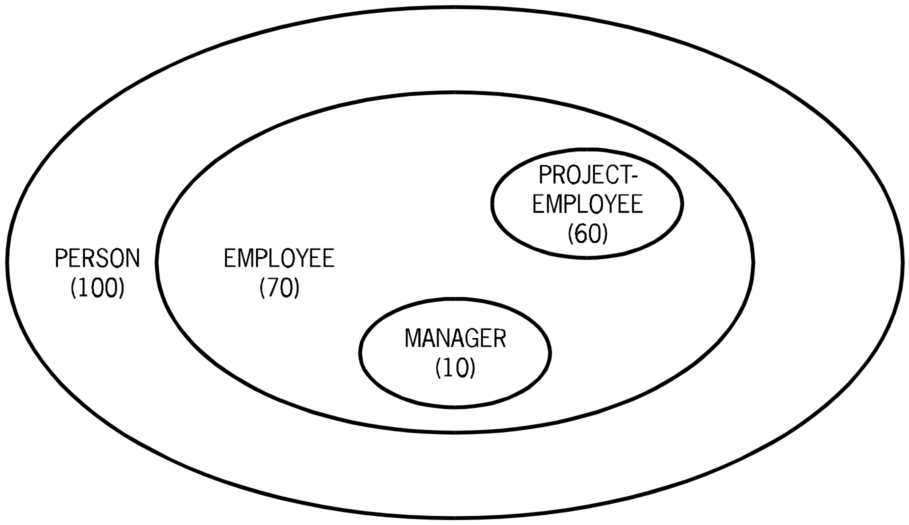


Figure 1–1. Class Relationships

All the details of an object available in a superclass are visible from any of its subclasses. Therefore, from the MANAGER class you can access all the information about a person as a manager, as an employee, and as a person.

Assume the PERSON class contains information about 100 people. Of these 100 people, 70 are employees and 30 are previous employees. Of the 70 employees, 10 are managers and 60 are project employees. The 60 project employees are employees and they are also people. The 10 managers are employees and they are also people.

In most retrieval queries, you work through a subclass to extract information from a superclass. However, you can also access information in a subclass from any of its superclasses using special syntax. The syntax for retrieval queries is provided in Section 3, “Structuring Retrieval Queries.”

Defining Attributes

Each object described in a SIM database has associated with it a number of descriptors, called attributes. Each attribute describes one facet of the object. For example, the attribute DEPT_TITLE describes the title of a department, and the attribute PROJECT_NO describes the number assigned to a project.

Figure 1–2 illustrates the type of information used to define the people described in the ORGANIZATION database. Note that because of the superclass/subclass relationships that exist, some information is associated with more than one class.

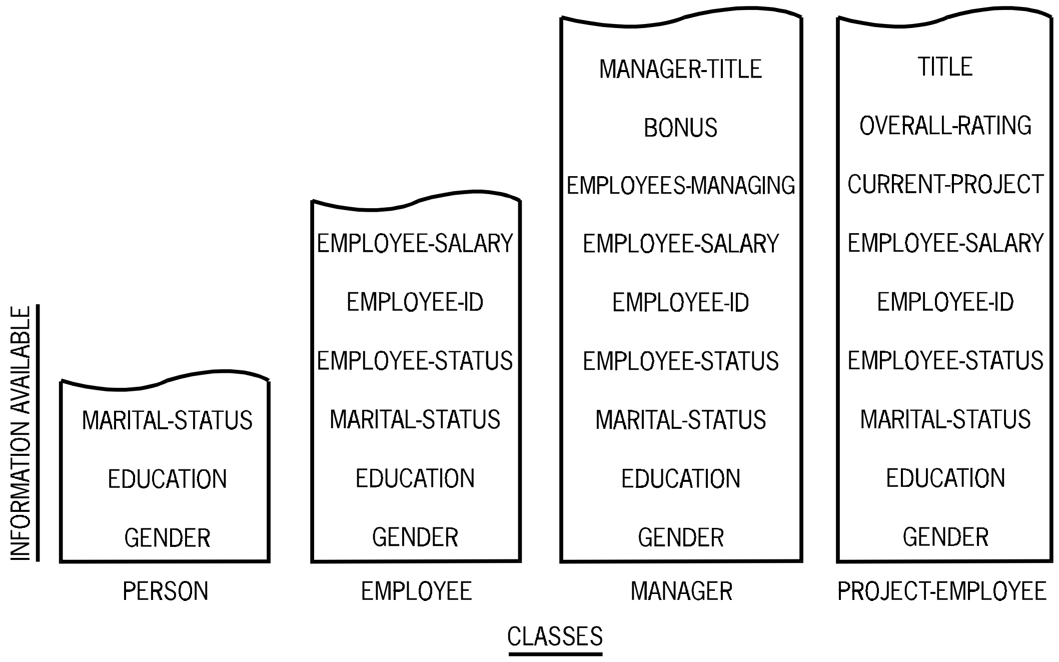


Figure 1–2. Attributes

When the database designer defines a SIM database, he or she assigns one or more of the following adjectives to each attribute:

- Data-valued or entity-valued
- Single-valued or multivalued
- Required
- Unique
- Simple or compound

In addition, attributes acquire characteristics based on their position in the database in relation to any particular class. In relation to a class, an attribute might be described as any one of the following:

- An immediate attribute
- An inherited attribute
- An extended attribute

The following information describes the characteristics associated with attributes.

Defining Data-Valued Attributes (DVAs) and Entity-Valued Attributes (EVAs)

Attributes can be data-valued or entity-valued.

Understanding DVAs

A DVA is a printable value that provides one piece of information about an entity. For example, the attribute EMPLOYEE_SALARY is a DVA in which each value describes the salary of one employee. A DVA can be any of the types described in Table 1–1.

Table 1–1. Data Types

Data Type	Explanation	Example
Boolean	The values true or false.	
Character	A single 8-bit character.	A, f, $, 3
Date	A date in the form MM/DD/YYYY, where MM denotes the month, DD denotes the day, and YYYY denotes the year. If the first two digits of the year are *19*, you can also use the form MM/DD/YY, where YY is the last two digits of the year. ***Note:*** *If you use IQF to process a query, you can also use the date formats DD/MM/YYYY and DD/MM/YY.*	12/25/1883, 5/19/87
Integer	A whole number.	23, 123, 9876, –27
Kanji	A set of Japanese characters. This unique character definition requires 16 bits.	
Number	A fixed-point number stored with an explicit number of digits before and after the decimal point.	2.34, 5.46
Real	A 48-bit floating-point number.	3E-12, 2.34E3, 3.9
String	A set of characters with a data type of either character or Kanji. When you use a string, it must be enclosed in double quotation marks.	"cabbages", "cars", "sealing wax"
Subrole	A read-only attribute that identifies subclasses. The value of the subrole is a class identifier.	PROJECT_EMPLOYEE, INTERIM_MANAGER
Symbolic	A set of unique related values.	Monday, Tuesday

Table 1–1. Data Types

Data Type	Explanation	Example
Time	A time in the form HH:MM:SS, where HH denotes the hour, MM denotes the minutes, and SS denotes the seconds. Seconds are optional.	12:22:34, 3:54
User-defined type (UDT)	A combination of any of the previously defined data types.	Degree could be defined as having the symbolic values HS, BA, BS, MA, MS, and PHD.

Understanding EVAs

An entity-valued attribute (EVA) acts as a connection between two or more entities described in the database. For example, you can use an EVA to describe a link between a manager and all his or her employees or to provide a link between an employee and his or her spouse.

When an EVA is defined, SIM automatically provides a reverse connection. For example, if the database designer defines an EVA that links all employees to their spouses, you can also use that link to connect each spouse to an employee. However, in some cases, your database designer might choose to create two EVAs to define the forward and backward relationships between two entities. Using two EVAs to explicitly define relationships simplifies the queries you write to extract or update information about related entities. For example, in the ORGANIZATION database, the EVA *EMPLOYEE_MANAGER* links each employee to a particular manager, and the EVA *EMPLOYEES_MANAGING* links each manager to his or her manager. Note that the EVA *EMPLOYEE_MANAGER* is single-valued, while the EVA *EMPLOYEES_MANAGING* is multivalued.

The database designer can use an EVA to connect entities that are

- In the same class
- In the same hierarchy
- Not in the same hierarchy

For example, in the ORGANIZATION database, the EVA *PARENT* connects entities in the same class—the PERSON class; the EVA *EMPLOYEE_MANAGER* connects entities in the same hierarchy—the EMPLOYEE and MANAGER classes; and the EVA *ASSIGNMENT_HISTORY* connects entities in different hierarchies—the ASSIGNMENT and PROJECT classes.

Defining Compound Attributes

If an attribute contains several pieces of information that describe one aspect of an entity, that attribute is called a *compound* attribute. For example, the attribute EDUCATION is a compound attribute with the components DEGREE_OBTAINED, YEAR_OBTAINED, and

GPA. Any component of a compound attribute can also be a compound attribute. In a retrieval query, you can choose to inquire about individual components or about all the components of a compound attribute.

Defining Single-Valued Attributes and Multivalued Attributes

Attributes can be single-valued or multivalued. AGE is a single-valued attribute because a person can have only one age. But CURRENT_PROJECT is multivalued because a project employee can be working on more than one project at a time.

Understanding Immediate Attributes

An immediate attribute is one directly associated with a particular class. For example, EMPLOYEE_ID is an immediate attribute of the EMPLOYEE class.

Figure 1–3 illustrates the immediate attributes associated with the classes PERSON, EMPLOYEE, MANAGER, and PROJECT_MANAGER. The immediate attributes of the classes are denoted by the shaded areas.

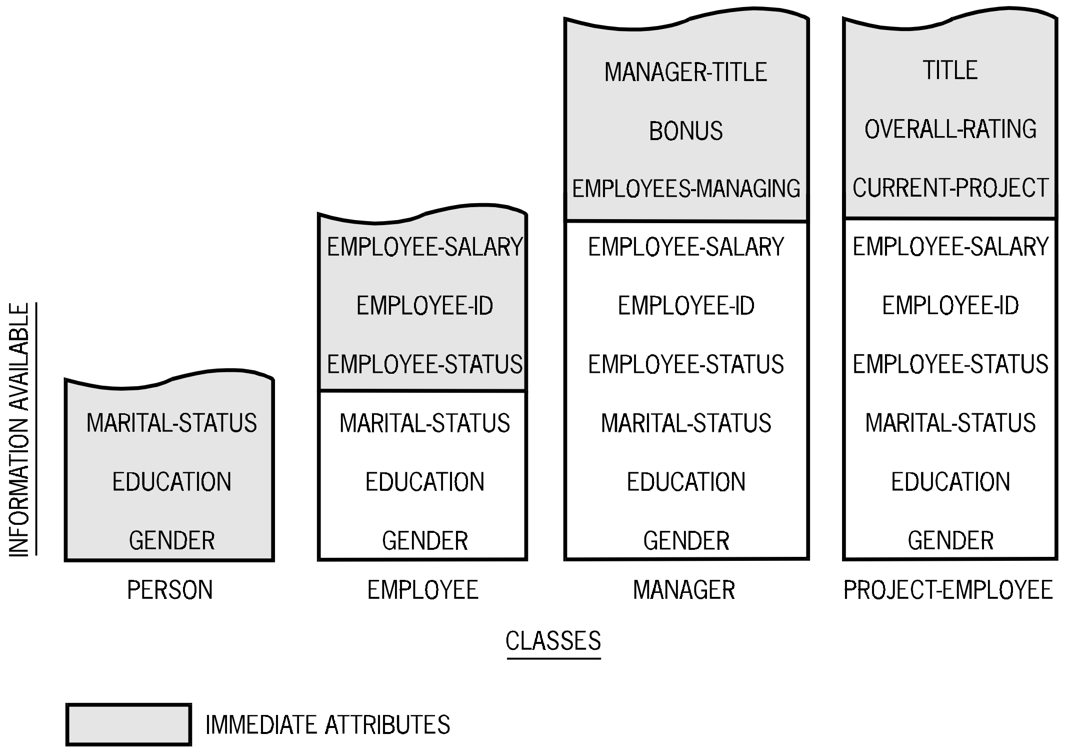


Figure 1–3. Immediate Attributes

Understanding Inherited Attributes

An inherited attribute is associated with a class because the class is defined as a subclass of another class. For example, the attribute EMPLOYEE_ID is an inherited attribute of the MANAGER class, but an immediate attribute of the EMPLOYEE class.

The advantage of inherited attributes is that common information need appear only once in the database.

Figure 1–4 illustrates the inherited attributes associated with the classes PERSON, EMPLOYEE, MANAGER, and PROJECT_MANAGER. The inherited attributes are denoted by the shaded areas in the figure.

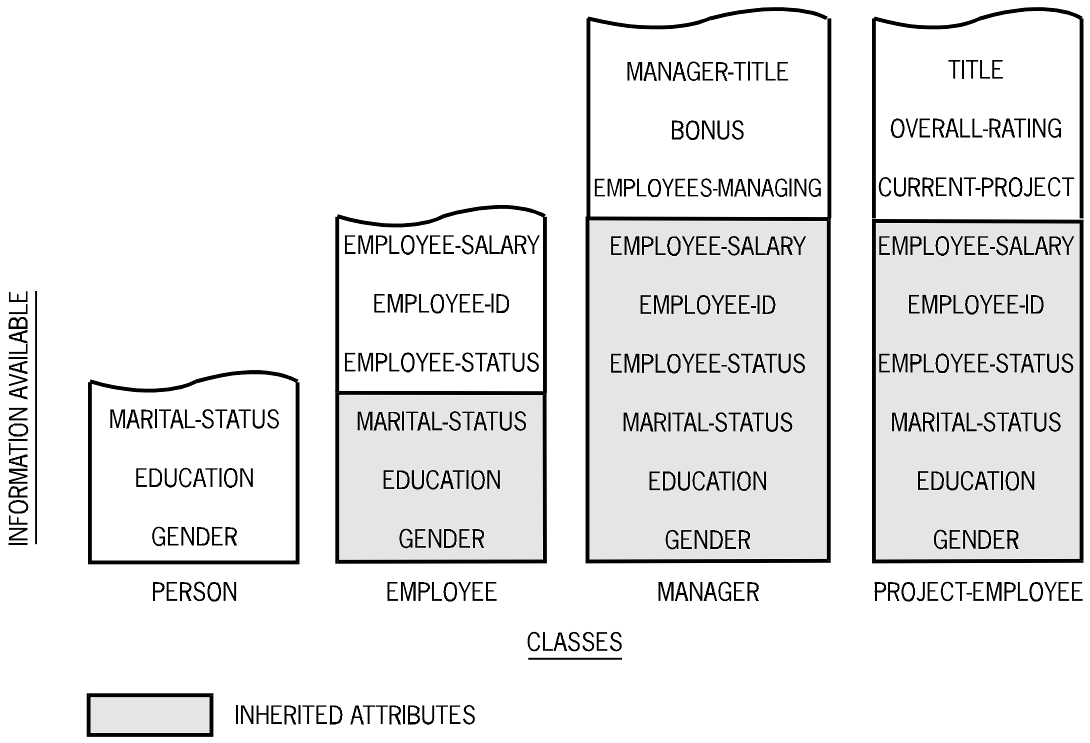


Figure 1–4. Inherited Attributes

Note: *When writing queries, immediate and inherited attributes are treated identically.*

Understanding Extended Attributes

An extended attribute is associated with a class because an EVA is defined that links the entities in the class you are looking at with the entities in the same or a different class. For example, in relation to the EMPLOYEE class, the immediate and inherited attributes of the MANAGER class are extended attributes because of the EVA *EMPLOYEE_MANAGER.*

Using OML

Using OML, you can create queries that either return information from a database or modify a database. The two types of queries are the following:

- Retrieval queries

  Retrieval queries return information from a database. The data can be returned in a designated order; however, to produce a finished report (graphic, tabular, or so on) you need to do additional formatting not supported by OML. You can do this additional formatting programmatically or by using IQF.

- Update queries

  Update queries modify a database in a user-determined manner. The modification can include adding information to, changing information in, or deleting information from a database.

Defining the Elements of a Retrieval Query

To define a retrieval query, you must provide the following three pieces of information:

- Information regarding the perspective of your query

  You can define the query perspective explicitly or implicitly. To define the query perspective explicitly, include a statement that provides the names of the perspective class or classes. If you do not include this statement in your query, the query perspective is extrapolated from other information you supply; this is called an implicit query perspective.

- Information detailing the data you want to retrieve from the database

  This information includes attributes and target attribute expressions. You can use target attribute expressions to manipulate data in the database to form temporary attributes.

- Conditions limiting the data you want to retrieve from the database

  Two types of condition are used to limit data retrieval: local filter expressions and global selection expressions.

  – Local filter expressions limit the amount of information returned for a particular entity. The expression does not affect the total number of entities about which information is returned. There is no limit on the number of local filter expressions that you can use in a retrieval query.

  – Global selection expressions limit the entities you want to retrieve from the database. You can define only one global selection expression for any retrieval query.

Defining the Elements of an Update Query

To define an update query, you must provide certain information on the entity or entities you want to insert, modify, or delete. The required information includes the following:

- The class in which the entity or entities belong
- The attributes you want to insert, modify, or delete
- A global selection expression to identify the entities you want to update

Selecting Your Perspective Class

When creating a retrieval query, you must first determine the aim of your query. That is, with what perspective do you want to extract data? For example, do you want to retrieve data about certain projects or certain employees? To make your query as meaningful as possible, select a single class as your perspective class. Make sure that all the data you want to retrieve is visible from the class as immediate, inherited, or extended attributes.

For example, to retrieve the following information, choose PROJECT as your perspective class:

- Project number
- Project title
- Project team
- Project manager
- Department assigned to the project

Similarly, choose MANAGER as your perspective class if you want to retrieve information about a manager's employees.

In some instances, the best perspective class might not be obvious. In these cases, refer to a database schema, such as the one shown in Figure C–1 (see Appendix C, “Description of the ORGANIZATION Database”). Using the schema, your knowledge of the superclass/subclass relationships, and your understanding of EVAs, trace a path through the database. Start from different classes until you find the route that is the simplest, yet enables you to retrieve all the information you want.

Writing Expressions

You can include three types of expressions in a retrieval query:

- Target attribute
- Local filter
- Global selection

In update queries you can include only the following types of expressions:

- Local filter, in the following manner:
  - As the right-hand side of an EVA assignment
  - As part of an aggregate function; that is, a function that counts the number of values or finds the minimum, maximum, sum, or average value
  - As part of a quantifier expression; that is, a function that determines whether all, some, or none of the values fit a requirement
- Global selection

Target Attribute Expression

You use a target attribute expression to create temporary attributes for a report. To create temporary attributes, you manipulate existing database attributes. For example, you can return the total salary expenditure for a project by totaling the salaries of all the project employees assigned to the project.

Local Filter Expression

You use a local filter expression to limit the number of times a particular attribute value appears in a report or to designate the values that should be considered when computing aggregate functions and quantifier expressions. To define a local filter, use the WITH clause. For more information on defining local filters, refer to “Using the WITH Clause to Create a Local Filter Expression” in Section 3, “Structuring Retrieval Queries.”

Global Selection Expression

You use a global selection expression to limit the entities that appear in a report or that are affected by an update query. To define the global selection expression, use the WHERE clause.

For more information on creating global selection expressions, refer to “Using the WHERE Clause to Create a Global Selection Expression” in Section 3, “Structuring Retrieval Queries.”

Writing Efficient Queries

The following text provides some guidelines to help you efficiently query a SIM database. Use these guidelines when you are creating new queries, modifying existing queries, and designing algorithms.

Although all the examples provided in this section are retrieval queries, the guidelines apply also to queries that modify the database. In addition, all the guidelines that relate to the WHERE clause of retrieval queries apply also to queries that delete data from the database.

The examples in this section are all based on the following schema. For information on the Object Definition Language (ODL) syntax used in this schema, refer to the *SIM ODL Programming Guide.*

```
CLASS ENTITY
  (WHEN_CREATED   : REQUIRED, UNIQUE
  ;PRIMARY_NAME
  ;NODE
  ;VERSION
  ;STATUS
  ;LOCKED
  ;CREATOR        : USER INVERSE IS CREATED, REQUIRED
  ;CHILDREN       : ENTITY_TO_ENTITY INVERSE IS PARENT_IS, MV
  ;PARENTS        : ENTITY_TO_ENTITY INVERSE IS CHILD_IS, MV
  );

CLASS RELATIONSHIP
  (WHEN_CREATED   : REQUIRED, UNIQUE
  ;DECLARED_ORDER
  ;RELATION_STATE
  ;CHANGE_INFO
  ;RELATION_DT    : DT INVERSE IS RELATION_FOR, REQUIRED
  );

SUBCLASS ENTITY_TO_ENTITY OF RELATIONSHIP
  (CHILD_IS       : ENTITY INVERSE IS PARENTS, REQUIRED
  ;PARENT_IS      : ENTITY INVERSE IS CHILDREN, REQUIRED
  );

CLASS DT
  (WHEN_CREATED   : REQUIRED, UNIQUE
  ;DT_TYPE_NAME
  ;DT_TYPE_VALUE
  ;RELATION_FOR   : RELATIONSHIP INVERSE IS RELATION_DT, MV
  );
```

```
CLASS USER
  (USER_NAME
  ;CREATED        : ENTITY INVERSE IS CREATOR, MV
  );

INDEX ENTSUR  ON ENTITY (WHEN_CREATED);

INDEX RELSUR  ON RELATIONSHIP (WHEN_CREATED);
INDEX RELORD1 ON RELATIONSHIP (DECLARED_ORDER);
INDEX RELORD2 ON RELATIONSHIP (DECLARED_ORDER, RELATION_STATE);
INDEX DTVALUES ON DT (DT_TYPE_VALUE);
INDEX DTSURS  ON DT (WHEN_CREATED);
```

Using Attributes Efficiently

Immediate attributes are the most efficient to retrieve since their values are present in the perspective class.

Inherited attributes are those that a subclass inherits from its superclasses. Depending on the mapping of the superclasses, the subclass, and the attributes, inherited attributes can be as efficient to retrieve as immediate attributes.

Extended attributes are always significantly less efficient to retrieve than either immediate attributes or inherited attributes. The extended attribute information is retrieved using an EVA, which involves accessing several additional storage structures that are not used when immediate attributes and inherited attributes are retrieved. However, if an EVA is not present (that is, has a null value), no additional costs are incurred. Note that some multivalued DVAs are treated like extended attributes.

Even though it is less efficient to retrieve EVAs, it is still more efficient to write one query that retrieves all the information you want than to write several queries that use only immediate attributes and inherited attributes to achieve the same task.

For more information on immediate attributes, inherited attributes, and extended attributes, refer to the *InfoExec SIM Technical Overview*.

Retrieving Additional Attributes from an Entity

The number of attributes you retrieve for any entity does not affect the efficiency of your query. For example, the following two queries incur the same performance cost:

Query 1

```
FROM ENTITY
RETRIEVE PRIMARY_NAME
WHERE WHEN_CREATED = #1
```

Query 2

```
FROM ENTITY
RETRIEVE PRIMARY_NAME, NODE, VERSION, STATUS, LOCKED
WHERE WHEN_CREATED = #1
```

There is no significant added expense since the entity being retrieved contains all of the single-valued DVAs. The bulk of the processing is in finding and accessing the entity, and not in finding and accessing the attributes that describe the entity.

Processing Additional Queries

Try to minimize the number of queries that your program has to make for a given transaction. If you can call SIM once instead of two or three times, then the overhead associated with making the additional query calls does not occur.

Example

For instance, as shown in this example, you can increase efficiency by replacing query 1 and query 2 by query 3.

Query 1

```
FROM RELATIONSHIP
RETRIEVE (WHEN_CREATED, LOCKED) OF CHILD_IS
WHERE WHEN_CREATED OF RELATIONSHIP = #1
```

Query 2

```
FROM RELATIONSHIP
RETRIEVE WHEN_CREATED OF PARENT_IS
WHERE WHEN_CREATED OF RELATIONSHIP = #1
```

Query 3

```
FROM ELATIONSHIP
RETRIEVE WHEN_CREATED OF PARENT_IS,
        (WHEN_CREATED, LOCKED) OF CHILD_IS
WHERE WHEN_CREATED OF RELATIONSHIP = #1
```

One significant exception to this guideline occurs when you know that some of the information is not needed most of the time. For example, if the information from the EVA *CHILD_IS* is needed all of the time, but the information accessed using the EVA *PARENT_IS* is needed only on discovering that the CHILD_IS entity is locked (and this occurs very infrequently) then you might want to use the two separate queries instead of the combined query.

Using Indexes Instead of Linear Searches

Avoid linear searches whenever possible since they are generally less efficient than indexed searches. Linear searches are generally used when the query text does not include a WHERE condition. A WHERE condition that does not include any key attributes can also produce a linear search. A warning message is returned if your query includes a WHERE condition and produces a linear search.

However, there are two occasions when using linear searches can be more efficient than using index searches.

- If you are selecting most of the entities in a class
- If the class you are searching has a very low population

  For example, if the data you need can be accessed in either of the following ways, it might be better to use the linear search:

  - By a linear search of Class A, which contains approximately 100 entities
  - By an indexed search of Class B, which contains approximately 1,000,000 entities

Using an ORDERED BY Clause

The ORDERED BY clause, which is used to sort data, generally increases the processing time for a query. Using the ORDERED BY clause forces SIM to sort all the retrieved entities; the more keys that you include in the ORDERED BY clause, the longer the process. But using the ORDERED BY clause is sometimes necessary. If a large number of entities are retrieved, the following example illustrates a potentially expensive query:

```
FROM ENTITY
RETRIEVE PRIMARY_NAME, NODE
ORDERED BY ASCENDING PRIMARY_NAME,
           DESCENDING VERSION
WHERE USER_NAME OF CREATOR EQL #1
```

If there is an index specified in the database schema that is keyed on the attribute by which you want the data ordered, then SIM can retrieve the data using the index and no significant performance penalty is incurred. For example, the ORDERED BY clause in the following query does not impose a performance penalty because there is an index for the class RELATIONSHIP declared in the database schema that is keyed on the attribute DECLARED_ORDER:

```
FROM RELATIONSHIP
RETRIEVE WHEN_CREATED
ORDERED BY ASCENDING DECLARED_ORDER
WHERE RELATION_STATE EQL #1
```

If you omit the ORDERED BY clause from the query, SIM can use any one of the three indexes declared on the class RELATIONSHIP, depending on which keys are in the WHERE condition. Which index is used is determined by the SIM optimizer. The index the SIM optimizer chooses, because of other aspects of the query, might not be the most obvious index.

Another occasion when no performance penalty occurs for the ORDERED BY clause is when the query finds no data to return. If there is no data to return, a sort operation is not performed.

Despite the cost of using the ORDERED BY clause, you should keep in mind that in most cases SIM can return the sorted information to the program more efficiently than the program could retrieve and sort the information itself. If you really need the information sorted, then let SIM do it.

Selecting the Perspective Class

SIM analyzes a query and formulates the best plan for retrieving the data as quickly as possible. SIM breaks down the text of a query into a query tree, and then after further analysis, SIM develops a strategy tree, which might be identical to the query tree. You can review the query and strategy trees to determine if the decision made by SIM improved performance. If the performance is not improved, you might be able to rewrite your query to force SIM to make a different decision.

To access strategy and query trees, set the appropriate fields in the QUERY_INFO record when you process your query using the SIM library entry points. For more information on the SIM library entry points and the QUERY_INFO record, refer to Section 7, “Using the SIM Library Entry Points.”

Example

Assume you want to process the following query:

```
FROM ENTITY_TO_ENTITY
RETRIEVE WHEN_CREATED,
     DT_TYPE_NAME OF RELATION_DT,
     WHEN_CREATED OF CHILD_IS
ORDERED BY ASCENDING CHANGE_INFO,
     ASCENDING DECLARED_ORDER
WHERE WHEN_CREATED OF PARENT_IS EQL #1 AND
     DT_TYPE_VALUE OF RELATION_DT EQL #2
```

The query tree for the query looks as follows:

```
FROM ENTITY_TO_ENTITY
     TO RELATIONSHIP
        VIA ISA EVA
        TO DT
           VIA M:1 EVA RELATION_DT
     TO ENTITY
        VIA M:1 EVA CHILD_IS
     TO ENTITY
        VIA M:1 EVA PARENT_IS
```

The strategy tree for the query looks as follows:

```
FROM DT       (Index equals search)
     TO RELATIONSHIP
        VIA 1:M EVA RELATION_FOR
        TO ENTITY_TO_ENTITY
           VIA ASA EVA
           TO ENTITY
              VIA M:1 EVA CHILD_IS
           TO ENTITY
              VIA M:1 EVA PARENT_IS
```

In this example, when SIM forms the strategy tree, the perspective class changes from ENTITY_TO_ENTITY to DT. In the WHERE clause of the original query, the SIM optimizer identified the following two specific values on which to base its search:

- The attribute WHEN_CREATED of the class ENTITY (using the EVA *PARENT_IS*)
- The attribute DT_TYPE_VALUE of the class DT (using the EVA RELATION_DT)

A search can begin only in one place. The SIM optimizer chose to start in the DT class. The SIM optimizer would never choose to start in the class ENTITY_TO_ENTITY (the class ENTITY_TO_ENTITY is a subclass of the class RELATIONSHIP) because the WHERE clause does give a specific key value using any attribute in the class ENTITY_TO_ENTITY. Using the class ENTITY_TO_ENTITY as the query perspective forces a linear search. Whereas a linear search is not necessary if the perspective class is changed to DT.

However, in this instance the choice made by the SIM optimizer does not produce the most efficient query because, in the example application, thousands of entities are likely to meet the condition *DT_TYPE_VALUE EQL #2*. Rather than having to process all of these thousands of entities and deciding which of these also meet the first condition, it would be better if the SIM optimizer were to select the class ENTITY as the query perspective since only one entity meets the condition *WHEN_CREATED OF PARENT_IS EQL #2*.

A better strategy tree for the query is the following:

```
FROM ENTITY (Index equals search)
       TO ENTITY_TO_ENTITY
          VIA 1:M EVA CHILDREN    % inverse of PARENT_IS
          TO RELATIONSHIP
             VIA ISA EVA
             TO DT
                VIA M:1 EVA RELATION_DT
             TO ENTITY
                VIA M:1 EVA CHILD_IS
```

To force the SIM optimizer to choose the class ENTITY as the query perspective, you must include the less favorable perspective class in an OR condition in the WHERE clause as follows:

```
FROM ENTITY_TO_ENTITY
RETRIEVE WHEN_CREATED,
     DT_TYPE_NAME OF RELATION_DT,
     WHEN_CREATED OF CHILD IS
ORDERED BY ASCENDING CHANGE_INFO,
     ASCENDING DECLARED_ORDER
WHERE WHEN_CREATED OF PARENT_IS EQL #1 AND
          ( FALSE OR
           DT_TYPE_VALUE OF RELATION_DT EQL #2
          )
```

Although it seems logical that the query parser should discard the FALSE OR part of the WHERE clause, the optimizer avoids keying on one half of an OR condition and instead keys on the condition *WHEN_CREATED OF PARENT_IS EQL #1*. As a result, the query perspective now used by the SIM optimizer is the class ENTITY and the new query processes more quickly than the original query.

Note that you could also use the following WHERE clause to achieve the same effect. In this WHERE clause the FALSE value is passed to the query as a parameter.

```
WHERE WHEN_CREATED OF PARENT_IS EQL #1 AND
  ( #2 OR
  DT_TYPE_VALUE OF RELATION_DT EQL #3
  )
```

Documenting Queries

When you tailor queries for optimal performance, be sure to document them right next to the query text to explain your strategy. Someone could easily destroy your efforts if they see the elements such as FALSE OR in a query. Also the strategy might have to be reworked if the query is altered at a later date or if the database schema is altered.

Using the OML Interfaces

This guide discusses two methods for constructing queries. One method uses IQF; the other method involves embedding the OML in a user-written program that calls the SIM library.

Another method for submitting queries against a SIM database is not discussed in this guide: using the programmatic interfaces provided in ALGOL, COBOL74, and Pascal. For more information on this method, refer to the appropriate user language manuals and the *InfoExec SIM Programming Guide.*

Note: *You can use the compiler control options SET LIST and SET LISTQUERYTEXT to view the OML generated by a query that is submitted through a programmatic interface.*

The advantage of using IQF to create a query is that you have to write only the OML; you need not write a program to process the query. You can also use the screen interface for formatting the output of a retrieval query.

The advantage of using the SIM library interface instead of the compiler extensions is that the SIM library enables you to alter your query at run time. You can designate the file containing the query you want to process as well as the values to be used in the query at run time.

Querying Files, DMSII Databases, LINC II Databases, and SQLDB Databases

You can use OML to retrieve data from and update data in SIM databases and in DMSII databases that have been processed using DMS.View. You can also retrieve data from the following:

- Files defined in an Advanced Data Dictionary System (ADDS) data dictionary
- LINC II databases that have been processed using LINC.View
- Structured Query Language Database (SQLDB) databases

For more information on DMS.View, refer to the *InfoExec DMS.View Operations Guide.* For more information on LINC.View, refer to the *InfoExec LINC.View Operations Guide.* For more information on SQLDB, refer to the *InfoExec SQLDB Programming Guide.*

Querying Files

When files are stored in an ADDS data dictionary, they are treated as COBOL74 files. The components of the files are records, groups, and items. For information on defining a file in ADDS, refer to the *InfoExec ADDS Operations Guide.*

A file can contain only one class. The class can contain any number of compound attributes, and the file can have nested compound attributes.

Because occurring groups and fields do not correspond to any SIM construct, they are ignored.

Table 1–2 illustrates the conversion of an ADDS file structure to a SIM database.

Table 1–2. Mapping an ADDS File to a SIM Database

File Element	ADDS Definition	SIM Definition
Logical file name	File identifier	Database name
Physical file name	File name	Class name
Group	Group element	Compound attribute
Field	Item element	Component

Example

The file PHONEDIRECTORY, with the group EMPLOYEE, contains six items: LAST_NAME, FIRST_NAME, EXTENSION, ROOM, CC, and MANAGER. This file becomes the SIM database PHONEDIRECTORY with the class PHONEDIRECTORY, which has one compound attribute EMPLOYEE. The components of EMPLOYEE are the same as the six items.

Note: *When you use OML to query a file, you must always qualify the class name by the database name. For example, FROM PHONEDIRECTORY OF PHONEDIRECTORY.*

Querying DMSII Databases

DMSII databases that have been processed by DMS.View can be the subject of OML retrieval and update queries. You can also use DMSII applications to query and update the database.

For information on the mapping of a DMSII database to a SIM database, update restrictions and behavior patterns that are different from a native SIM database, and on using DMS.View, refer to the *InfoExec DMS.View Operations Guide.*

Querying LINC II Databases

LINC II databases that have been processed by LINC.View can be the subject of OML retrieval queries. However, you cannot update LINC II databases using OML; use LINC II applications to query and update the database.

Refer to the *InfoExec LINC.View Operations Guide* for information on using LINC.View.

Querying SQLDB Databases

SQLDB databases can be the subject of OML retrieval queries. You can also process OML retrieval queries and SQLDB dynamic DML queries against SQLDB databases, using the SIM library entry points. The SIM library entry points are described in Section 7, “Using the SIM Library Entry Points,” and Section 8, “Describing the SIM Library Entry Points.” For more information on the SQLDB dynamic DML, refer to the *InfoExec SQLDB Programming Guide.*

Querying Databases and Files with Different Query Languages

Table 1–3 details the types of queries and the query languages you can use to perform queries against different types of databases.

In the table, the term *update* refers to any form of query that adds, alters, or removes data from a database. The term *DML* refers to both the SQLDB dynamic DML and the SQLDB module language.

Table 1–3. Query Languages Available for Different Database Types

Database Type	Query Language	Query Types Allowed
Native SIM database	SIM OML	Retrieval, update
	DML	None
DMS.View database	SIM OML	Retrieval, update
	DML	Retrieval, update
LINC.View database	SIM OML	Retrieval
	DML	Retrieval
Files defined in ADDS	SIM OML	Retrieval
	DML	Retrieval
Native SQLDB database	SIM OML	Retrieval
	DML	Retrieval, update

Using the IQF Interface

IQF is a screen-based product you can use to construct queries against SIM databases. The databases need not be defined in an ADDS data dictionary. A full description of IQF and the procedures for accessing and using the product is provided in the *InfoExec IQF Operations Guide.*

When using IQF, you can choose to construct part or all of a retrieval or update query by entering the required OML expression directly. Alternatively, you can construct the query interactively, using a series of menus and screens. However, if you want to tailor the final formatting of an output report, you must use the interactive screens.

Opening a Database

If a database is not already open, the first step in creating a query is to open a database. This step is required whether you are using the interactive screen flow or entering the OML expression directly.

To open a database, use the Open a Data Base or File (OPEN) screen. You can reach the OPEN screen by entering *GO OPEN* in the Action field of any IQF screen or by using the OPEN option on the IQF home menu, DATA BASE AND FILE QUERY (QUERY).

More than one database can be open at a time.

Creating a Query Interactively

Using IQF, you can generate a query by stepping through a series of menus and screens. At any point in the process, you can exit the screen flow and enter the remainder of the query directly. For example, you could choose to use the Select Perspective Class (CLASS) and Select Target List Attributes/Expressions (TARGET) screens to define your perspective class and the attributes to be retrieved. You would then use the Data Manipulation Language (DML) screen to define any local filter, target attribute, and global selection expressions.

You can choose to enter your expressions on specially designated blank screens rather than on a screen that provides a skeletal layout of an expression. On the blank screen, you enter the OML text for the expression you want to create—but omit the WITH or WHERE keyword that would normally prefix the expression as the keyword is automatically supplied by IQF.

For more information on using IQF to create queries and to format reports, refer to the *InfoExec IQF Operations Guide.*

Creating a Query on the IQF DML Screen

Before you can view the DML screen, you must first open a database. To get to the DML screen, enter *GO DML* in the Action field of any screen. This action displays the DML screen, which is a series of blank pages on which to enter your query.

To save partial and completed queries, use the SAVE option on the DATA BASE AND FILE QUERY (QUERY) screen.

Note: *Before you try to process your query, make sure the database containing the appropriate attributes and classes is open.*

Reviewing a Query

You can save queries that you create by using the interactive screen flow, and you can save those that you create on the DML screen. You can then recall the saved queries either during the session in which you saved the query or in future IQF sessions.

You can review the OML generated by queries produced by using the interactive screens, as well as those entered on the DML screen. When you are learning OML, you might find it helpful to produce a query, using the interactive screens, and then review the generated query, using the DML screen.

To review a query, use the LOAD option on the QUERY screen. Then enter *GO DML* in the Action field of any screen.

You can rerun saved queries, and you can edit the saved queries.

Formatting Reports

A retrieval query entered on the DML screen only defines the information to be retrieved; it cannot format a report. IQF supplies a default format, or you can use the formatting facility provided on the Report Type and Destination Selection (REPORTTYPE) screen in IQF. For more information on report formatting, refer to the *InfoExec IQF Operations Guide.*

Using the SIM Interface

Instead of using IQF to create a query, you can write a program that calls the SIM library. Using the library entry points, you open a database, send SIM the text of the query, and process the query.

You must make a separate call on the SIM library for each entity that you want to retrieve. Only one call on the SIM library is required to change all the entities affected by an update query.

You can write the program, calling the SIM library in any A Series user language that supports a library interface. For more information on the SIM library entry points, refer to Section 7, "Using the SIM Library Entry Points," and Section 8, "Describing the SIM Library Entry Points," in this guide. For examples of user-written programs that illustrate the use of the SIM library entry points, refer to the sample programs included on the Mark release tapes.

Section 2
OML Language Conventions

When you write OML statements, you must follow rules for

- Syntax
- Qualifying elements
- Identifying attributes

Following Syntax Rules

Whether you enter the query text on an IQF screen or through a user-written program, you must adhere to the following syntax rules:

- Leave the appropriate blank spaces in your query text. Otherwise, a syntax error occurs because the lines of a query are concatenated by the system. For instance, if a word ends at the rightmost position of a line, you must leave a blank space in the leftmost position of the following line.
- Always enclose literal text in double quotation marks, for example, "Engineer".
- Always precede hexadecimal representations of characters by the numeral 4 and enclose the characters in double quotation marks. For example, *4"C481A381"* is the hexadecimal representation of the word *Data*.
- Always precede hexadecimal representations of numbers by the numeral 4 and enclose the numbers in single quotation marks. For example, *4'C4'* is the hexadecimal representation of *196* in the decimal counting system.
- Use parentheses to override the default precedence for given operators. OML expressions are computed from left to right according to the precedence given to any operators used in the expression. The exception to this rule is the evaluation of exponents (**). If an expression contains a series of exponents, the exponents are computed from right to left. The following two expressions are equivalent:

  ```
  3**4**5**6

  3**(4**(5**6))
  ```

  The default precedence for given operators is described in Section 6, “Writing Expressions for Retrieval and Update Queries.”
- Use hyphens (-) and underscores (_) interchangeably in OML queries if you wish. However, you cannot use hyphens when writing SQLDB DML queries.

- Use uppercase and lowercase letters interchangeably if you wish, except in literal text. For example, Employee and EMPLOYEE are the same; "Engineering" and "ENGINEERING" are not the same.
- Be sure to define attributes unambiguously. Attributes are qualified by defining how they relate to the perspective class. For example, the following statement relates the attribute AGE to the employees of a manager:

  ```
  AGE OF EMPLOYEES_MANAGING OF MANAGER
  ```

  You can also use the AGE attribute to retrieve the age of an employee, a child, a person, a project manager, a previous employee, and a manager.

  The following example designates that the project title is that of a project employee's current project:

  ```
  PROJECT_TITLE OF CURRENT_PROJECT OF PROJECT_EMPLOYEE
  ```

  A project title can also be related to a department, a manager, and an assignment.

  For more information on defining attributes, refer to "Qualifying Names in Retrieval Queries" later in this section.
- Always qualify the name of a component of a compound attribute by the name of the compound attribute (or names, if there are multiple levels of compound attributes) as well as by the name of the perspective class. In the following example, GPA is a component of the compound attribute EDUCATION, and EDUCATION is an immediate attribute of the class EMPLOYEE:

  ```
  GPA OF EDUCATION OF EMPLOYEE
  ```

  In the following example, START_DATE is a component of the compound attribute INTERIM_HISTORY, and INTERIM_HISTORY is an immediate attribute of the class MANAGER:

  ```
  START_DATE OF INTERIM_HISTORY OF MANAGER
  ```

Qualifying Names in Retrieval Queries

You can include attributes from several classes in a retrieval query; therefore, you must qualify the attribute names to avoid ambiguities. In contrast, in an update query, the only place in which ambiguities can occur is in the selection expression when you define the entity you want to update. Ambiguities cannot occur in the remainder of the update query because you can alter only the immediate and inherited attributes of the perspective class.

Qualifying Classes

You must qualify class names under either of the following conditions:

- You have more than one database open, and the same class name appears in at least two of the databases.
- The name of your database is also the name of a class in the database.

Qualifying Attributes

To ensure that there is no ambiguity in a retrieval query, you must qualify attribute names if both the following conditions are true:

- The attribute can belong to more than one class.
- The connection between the class that has the attribute as an inherited or immediate attribute and the perspective class is unclear.

In a retrieval query with a single perspective class, each attribute referenced must be connected to the perspective class either as an immediate or extended attribute.

Note: *When writing queries, immediate and inherited attributes are treated identically.*

To completely define the attributes in a query, you must define a path that relates the attribute to the perspective class. The process of forming the path is called qualifying, and the path formed is known as a qualification path. By qualifying an attribute, you restrict the occurrences of the attribute that appear in a report.

The qualification considerations for an attribute depend on the following two items:

- The relationship of the attribute to the perspective class
- The number of perspective classes you use in the query

Qualifying Immediate and Inherited Attributes

In a retrieval query, use the FROM clause to name the perspective class or classes. (Using the FROM clause is discussed in Section 3, "Structuring Retrieval Queries.") If you use a single perspective class, the immediate and inherited attributes of the class referred to in the RETRIEVE clause (the clause used to identify the details of an entity that are to be retrieved from the database) need not be qualified under either of the following conditions:

- The attributes are the first items in the RETRIEVE clause.
- The attributes are not preceded by a qualification that could also be applied to them.

For example, the following query retrieves the names and ages of all employees:

```
FROM EMPLOYEE
RETRIEVE NAME, AGE
```

If you also want to retrieve the name and age of each employee's manager, you can use the following query. The parentheses are used to group information that has a common qualification.

```
FROM EMPLOYEE
RETRIEVE NAME, AGE, (NAME, AGE) OF EMPLOYEE_MANAGER
```

Even though the same attribute names, NAME and AGE, are used, the second occurrence of each attribute is an extended attribute of the perspective class and, therefore, require additional qualification. In the example, the second occurrence of the NAME and AGE attributes are qualified by OF EMPLOYEE_MANAGER. Qualifying extended attributes is discussed more fully under "Qualifying Extended Attributes" later in this section.

The following query, however, results in the syntax error "Qualification is insufficient. Error precedes - NAME, AGE."

```
FROM EMPLOYEE
RETRIEVE NAME OF EMPLOYEE_MANAGER, AGE OF EMPLOYEE_MANAGER, NAME, AGE
```

The error occurs because the last two attributes in the RETRIEVE clause can be considered to be inherited or extended attributes of the perspective class, EMPLOYEE.

If the FROM clause is omitted from a query, the first attribute of the perspective class must be qualified with the construct *OF <class name>*. This qualification is automatically applied to all the other attributes in the RETRIEVE clause. For example, the following query retrieves the names and employee identification numbers of all employees:

```
RETRIEVE NAME OF EMPLOYEE, EMPLOYEE_ID
```

If a query has more than one perspective class, you must connect each attribute by immediate or extended qualification to one of the perspective classes. For example, the following query retrieves the names of all employees and the names of all managers. Note that if an employee is also a manager, the query retrieves his or her name twice.

```
FROM EMPLOYEE, MANAGER
RETRIEVE (NAME, AGE, EMPLOYEE_ID) OF EMPLOYEE, NAME OF MANAGER
```

If an attribute can belong to only one of the perspective classes, the qualification construct OF <class name> need not be added. For example, the following query retrieves the names of all employees and the titles of all projects:

```
FROM EMPLOYEE, PROJECT
RETRIEVE NAME, PROJECT_TITLE
```

The effect of selecting more than one perspective class is discussed in more detail under "Identifying the Effect of Multiple Perspective Classes on Qualification Rules" later in this section.

Note: *If the FROM clause is omitted from the query, each class named in the RETRIEVE clause is considered to be a perspective class unless the class is used in one of the following:*

- *Aggregate function*
- *Quantifier expression*
- *Role definition, using either the AS or ISA keyword*

For more information, refer to "Using Aggregate Functions," "Using Quantifiers," and "Using Subroles" later in this section. Additional information is located in Section 6, "Writing Expressions for Retrieval and Update Queries."

Usually an attribute that is an immediate or inherited attribute of the perspective class requires no qualification. The attribute must be qualified, however, in the following three cases:

- If the FROM clause is omitted from the query. In this case, the qualification identifies the perspective class.
- If there are multiple perspective classes and the attribute is an immediate or inherited attribute of more than one perspective class.
- If some form of implicit qualification causes the attribute to be associated with a class other than the perspective class. For more information on implicit qualification, refer to "Qualifying Names Implicitly" later in this section.

Note: *The elements of a retrieval query and the OML keywords are discussed fully in Section 3, "Structuring Retrieval Queries."*

Qualifying Extended Attributes

You can qualify extended attributes that are either EVAs or DVAs.

To qualify extended attributes, use one or more *OF <EVA>* constructs followed by an *OF <perspective class>* construct. If the perspective class has already been established, you can omit the *OF <perspective class>* construct.

The order of the *OF <EVA>* constructs is determined by the relationship between the following:

- The DVA or EVA being qualified
- The class owning the DVA or EVA
- The perspective class of the query

The path formed by the *OF <EVA>* constructs must be a continuous path from the DVA or EVA being qualified to the perspective class. You must be familiar with the schema of your database to create a continuous path and thereby qualify the DVA or EVA correctly.

The following examples are based on the ORGANIZATION database. Refer to Appendix C, “Description of the ORGANIZATION Database,” for an illustration of the ORGANIZATION database schema.

Examples of Correct Qualification Paths

The following examples show valid qualification paths based on the relationships illustrated in Figure C–1.

Example 1

The following query retrieves the name of the department of an employee’s manager:

```
FROM EMPLOYEE
RETRIEVE DEPT_TITLE OF MANAGERS_DEPT OF EMPLOYEE_MANAGER
```

The attribute DEPT_TITLE is an immediate attribute of the DEPARTMENT class. The EVA *MANAGERS_DEPT* links the MANAGER class to the DEPARTMENT class. The EVA *EMPLOYEE_MANAGER* links the EMPLOYEE class to the MANAGER class. This query therefore links from the perspective class (EMPLOYEE) to the DEPARTMENT class by way of the MANAGER class.

Example 2

The following query retrieves the name of the department manager assigned to a project:

```
FROM PROJECT
RETRIEVE NAME OF DEPT_MANAGERS OF DEPT_ASSIGNED
```

The attribute NAME is an inherited attribute of the MANAGER class. The EVA *DEPT_MANAGERS* links the DEPARTMENT class to the MANAGER class. The EVA *DEPT_ASSIGNED* links the PROJECT class to the MANAGER class. This query therefore links from the perspective class (PROJECT) to the MANAGER class by way of the DEPARTMENT class.

Example of an Incorrect Qualification Path

The following example shows a qualification chain that is incorrect:

```
FROM EMPLOYEE
RETRIEVE DEPT_TITLE OF DEPT_MANAGERS OF EMPLOYEE_MANAGER
```

The error returned by this query is "Invalid attribute chain at DEPT_MANAGERS. Error occurred at the end of the statement."

Because the EVA *DEPT_MANAGERS* belongs to the DEPARTMENT class and the EVA *EMPLOYEE_MANAGER* belongs to the EMPLOYEE class and because there is no direct relationship in the database schema between these two classes, this path is not continuous and therefore returns a syntax error.

If you want to retrieve an employee manager's department title, use the following query:

```
FROM EMPLOYEE
RETRIEVE DEPT_TITLE OF MANAGERS_DEPT OF EMPLOYEE_MANAGER
```

Examples of Qualification Paths That Include the Perspective Class

The following examples show the two different ways in which the perspective class can be included in the extended attribute qualification. Both queries return the name of the department manager assigned to each project. Note that if you declare one perspective class in a FROM clause, you must declare all the perspective classes in the FROM clause.

Example 1

```
FROM PROJECT
RETRIEVE NAME OF DEPT_MANAGERS OF DEPT_ASSIGNED
```

Example 2

```
RETRIEVE NAME OF DEPT_MANAGERS OF DEPT_ASSIGNED OF PROJECT
```

Note: *If the attribute being defined is a component of a compound attribute, the OF <EVA> constructs are preceded by the required OF <compound attribute> constructs.*

Qualifying Reflexive Attributes

An extended attribute can belong to the same class as the attribute that points to it; for instance, a manager can have a manager. This form of extended attribute is referred to as *reflexive*. The following query retrieves the name of a manager and the name of that manager's manager:

```
FROM MANAGER
RETRIEVE NAME, NAME OF EMPLOYEE_MANAGER
```

Qualifying by Database Name

You must qualify a class name with a database name under either of the following conditions:

- You have more than one database open, and the same class name appears in at least two of the databases.
- The class name is also the database name.

To qualify a class name with a database name, use the OF <database name> construct. If you omit the FROM clause from the query, you must add the OF <database name> construct to the end of the attribute qualifications.

The following two queries return the names and addresses of the people in the PERSON class of the ORGANIZATION database. The first example uses the FROM class to name the perspective class. The second example includes the perspective class as part of the target list.

Example 1

```
FROM PERSON OF ORGANIZATION
RETRIEVE NAME, CURRENT_RESIDENCE
```

Example 2

```
RETRIEVE NAME OF PERSON OF ORGANIZATION, CURRENT_RESIDENCE
```

Identifying the Effect of Multiple Perspective Classes on Qualification Rules

If a query has more than one perspective class, the entities in all the classes are compared with each other. For example, if you have two perspective classes, PERSON and DEPARTMENT and there are 10 entities in PERSON and 2 entities in DEPARTMENT, your query matches each of the 10 PERSON entities with each of the 2 DEPARTMENT entities to produce 20 groups of information. The result of this matching process is referred to as the *cross-product* or the *Cartesian product*.

If you want to limit the amount of data returned, use the WHERE clause to create a value-based link between the two classes. For example, the following query matches each employee against each manager. Therefore, if there are 100 employees and 10 managers, each of the 10 managers is matched against each of the 100 employees. The pairs of employees and managers earning the same salary are returned.

```
FROM MANAGER, EMPLOYEE
RETRIEVE NAME OF MANAGER, NAME OF EMPLOYEE
WHERE EMPLOYEE_SALARY OF MANAGER = EMPLOYEE_SALARY OF EMPLOYEE
```

Because a manager is also an employee, the minimum this query returns is information about each person who is both an employee and a manager.

Note: *Using more than one perspective class is most useful when you are retrieving data from more than one database.*

Example

The following query not only checks for employees and managers with the same salary, but also ensures that the employee and manager are not the same person. You can use the system-supplied surrogate values assigned to each entity to ensure that two different entities are being compared. The element MANAGER <> EMPLOYEE in the WHERE clause compares surrogate values. Potentially, this query might not return any data.

```
FROM MANAGER, EMPLOYEE
RETRIEVE NAME OF MANAGER, NAME OF EMPLOYEE
WHERE EMPLOYEE_SALARY OF MANAGER = EMPLOYEE_SALARY OF EMPLOYEE
   AND
   MANAGER <> EMPLOYEE
```

Qualifying Names Implicitly

You can omit the rightmost one or more EVAs and the perspective class from a query if the complete qualification path of a previously referenced attribute includes all the omitted EVAs. However, there must not be more than one possible path. For example, the following two queries are equivalent:

Example 1

```
RETRIEVE EMPLOYEE_ID OF PROJECT_EMPLOYEE,
         MANAGER_TITLE OF EMPLOYEE_MANAGER,
         START_DATE OF ASSIGNMENT_RECORD,
         PROJECT_NO OF PROJECTS_MANAGING,
         DEPT_LOCATION OF MANAGERS_DEPT
```

Example 2

```
RETRIEVE EMPLOYEE_ID OF PROJECT_EMPLOYEE,
         MANAGER_TITLE OF EMPLOYEE_MANAGER OF PROJECT_EMPLOYEE,
         START_DATE OF ASSIGNMENT_RECORD OF PROJECT_EMPLOYEE,
         PROJECT_NO OF PROJECTS_MANAGING OF EMPLOYEE_MANAGER OF
             PROJECT_EMPLOYEE,
         DEPT_LOCATION OF MANAGERS_DEPT OF EMPLOYEE_MANAGER OF
             PROJECT_EMPLOYEE
```

The following example shows a query that generates a SIM error because there is more than one possible qualification for one of the elements:

```
FROM PROJECT EMPLOYEE
RETRIEVE EMPLOYEE ID,
         EMPLOYEE ID OF EMPLOYEE_MANAGER
WHERE EMPLOYEE_SALARY > 30000
```

The implicit qualification of the first occurrence of the EMPLOYEE_ID attribute is OF PROJECT_EMPLOYEE. The second occurrence of the EMPLOYEE_ID attribute has the additional qualification OF EMPLOYEE_MANAGER. Attempting to qualify the EMPLOYEE_SALARY attribute in the global selection expression (the WHERE clause), causes a problem because you can complete the qualification by using either of the following two clauses:

- OF PROJECT_EMPLOYEE
- OF EMPLOYEE_MANAGER OF PROJECT_EMPLOYEE

To make the query unambiguous, you must make the global selection expression unambiguous. You can do this by changing the global selection expression to the following:

```
WHERE EMPLOYEE_SALARY OF PROJECT_EMPLOYEE > 30000
```

You can also factor out common elements of a query, using parentheses. For instance, the following two queries are equivalent. Both queries return the title of a manager and the salaries of that manager's employees.

Example 1

```
RETRIEVE MANAGER_TITLE OF MANAGER,
         EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF MANAGER
```

Example 2

```
RETRIEVE (MANAGER_TITLE, EMPLOYEE_SALARY OF EMPLOYEES_MANAGING)
           OF MANAGER
```

You can also nest elements of the qualification, as shown in the following example:

```
RETRIEVE (EMPLOYEE_ID, EMPLOYEE_HIRE_DATE,
            (MANAGER_TITLE,
              (DEPT_TITLE, DEPT_LOCATION
              )
              OF MANAGERS_DEPT
            )
            OF EMPLOYEE_MANAGER
         )
         OF EMPLOYEE
```

Understanding the Binding of Names Rule

When a query is processed, the information is returned iteratively; that is, the information relating to each entity is retrieved in turn. If an attribute is mentioned more than once in the query, the same value is returned for each occurrence. The multiple occurrences of an attribute are said to be *bound* to each other.

For example, the following query checks to see if the start date of a project employee's current assignment is January 1, 1988. If the start date fits this requirement, the employee identification number of the project employee and the title of his or her current project are retrieved.

```
RETRIEVE EMPLOYEE_ID OF PROJECT_EMPLOYEE,
         PROJECT_TITLE OF CURRENT_PROJECT OF PROJECT_EMPLOYEE
WHERE  START_DATE OF ASSIGNMENT_RECORD OF PROJECT_EMPLOYEE = 1/1/88
```

In this query, SIM establishes a reference variable, or pointer. This reference variable successively assumes the value of each entity in the PROJECT_EMPLOYEE class. The reference variable ensures that the same project employee is being referred to each time the class name appears.

If multivalued attributes are involved in the query, each occurrence of the attribute is dealt with as if it is a single-valued attribute. For example, the following query returns the employee identification number of a manager if any of his or her employees fits both of the following criteria:

- The employee has a title of *specialist*.
- The employee has a rating of 1.

```
FROM MANAGER
RETRIEVE EMPLOYEE_ID
WHERE TITLE OF PROJECT_TEAM OF PROJECTS_MANAGING = Specialist
   AND OVERALL_RATING OF PROJECT_TEAM OF PROJECTS_MANAGING = 1
```

The binding of names rule ensures that the two occurrences of the PROJECT_TEAM OF PROJECTS_MANAGING construct refer to the same department member.

Breaking the Binding of Names Rule

The binding of names can be broken in several ways:

- Using the CALLED keyword to name a reference variable
- Using the quantifiers SOME, ALL, or NO
- Using the aggregate functions AVG, COUNT, MAX, MIN, or SUM
- Using a path expression

Using a Reference Variable

You can break the binding rule, using the CALLED keyword to name a reference variable explicitly.

The following query returns the employee identification number for a manager who has at least one nonexempt employee earning less than $30,000 and at least one exempt employee earning more than $50,000:

```
FROM MANAGER
RETRIEVE EMPLOYEE_ID
WHERE EMPLOYEE_SALARY OF EMPLOYEES_MANAGING < 30000
      AND EMPLOYEE_STATUS OF EMPLOYEES_MANAGING = NON_EXEMPT
      AND EMPLOYEE_SALARY OF EMPLOYEES_MANAGING
          CALLED OTHER_EMPLOYEE > 50000
      AND EMPLOYEE_STATUS OF OTHER_EMPLOYEE = EXEMPT
```

If you do not include a reference variable, this query cannot return a true value because no single employee can earn less than $30,000 and more than $50,000.

However, because the CALLED keyword is used in the query, the first EMPLOYEE_SALARY OF EMPLOYEES_MANAGING attribute can refer to a different employee from the second EMPLOYEE_SALARY OF EMPLOYEES_MANAGING attribute.

You cannot use the CALLED keyword after a name that was created by using a CALLED keyword. The following example shows a query that generates a syntax error. The error returned is "Unknown name - OTHER_EMPLOYEE. Error precedes - EXEMPT."

The error occurs because an attempt is made to allocate a reference variable to the reference variable OTHER_EMPLOYEE.

```
FROM MANAGER
RETRIEVE EMPLOYEE_ID
WHERE EMPLOYEE_SALARY OF EMPLOYEES_MANAGING <= 30000
      AND EMPLOYEE_SALARY OF EMPLOYEES_MANAGING
          CALLED OTHER_EMPLOYEE >= 50000
      AND EMPLOYEE_STATUS OF EMPLOYEES_MANAGING = NON_EXEMPT
      AND EMPLOYEE_STATUS OF OTHER_EMPLOYEE
          CALLED EXEMPT_EMPLOYEE = EXEMPT
```

To correct the preceding example, replace the last line of the query with the following:

```
= EXEMPT
```

Following is the corrected query:

```
FROM MANAGER
RETRIEVE EMPLOYEE_ID
WHERE EMPLOYEE_SALARY OF EMPLOYEES_MANAGING <= 30000
      AND EMPLOYEE_SALARY OF EMPLOYEES_MANAGING
          CALLED OTHER_EMPLOYEE >= 50000
      AND EMPLOYEE_STATUS OF EMPLOYEES_MANAGING = NON_EXEMPT
      AND EMPLOYEE_STATUS OF OTHER_EMPLOYEE
          = EXEMPT
```

You can use the CALLED keyword after the class name in the FROM clause of a query, as in the following example:

```
FROM MANAGER CALLED BEST_MANAGER
```

The following query illustrates the use of the CALLED keyword in the FROM clause:

```
FROM MANAGER CALLED BEST_MANAGER
RETRIEVE EMPLOYEE_ID
WHERE EMPLOYEE_SALARY OF EMPLOYEES_MANAGING <= 30000
      AND EMPLOYEE_SALARY OF EMPLOYEES_MANAGING
          CALLED OTHER_EMPLOYEE >= 50000
      AND EMPLOYEE_STATUS OF EMPLOYEES_MANAGING = NON_EXEMPT
      AND EMPLOYEE_STATUS OF OTHER_EMPLOYEE
          = EXEMPT
```

Using Quantifiers

If you include multivalued attributes in a local filter or a global selection expression, you can use quantifiers to express whether some, all, or none of the values must meet the condition. Using a new quantifier always generates a new reference. You cannot use the CALLED keyword to name a reference variable. The following examples illustrate the effect of quantifiers on the binding of names:

Example 1

The following query returns the marital status of all of a manager's employees if both the following criteria are met:

- At least one of the manager's employees earns more than $30,000.
- None of the manager's employees earns less than $20,000.

```
FROM MANAGER
RETRIEVE MARITAL_STATUS OF EMPLOYEES_MANAGING
WHERE EMPLOYEE_SALARY OF SOME(EMPLOYEES_MANAGING) > 30000
      AND EMPLOYEE_SALARY OF NO(EMPLOYEES_MANAGING) < 20000
```

Example 2

The following query returns the marital status of all the manager's employees if any of his or her employees is over the age of 30. The employee identification numbers are returned regardless of the ages of the employees.

```
FROM MANAGER
RETRIEVE MARITAL_STATUS OF EMPLOYEES_MANAGING OF MANAGER
           WITH (AGE OF SOME(EMPLOYEES_MANAGING) > 30),
         EMPLOYEE_ID OF EMPLOYEES_MANAGING
```

For more information about quantifiers, refer to "Using Quantifiers" in Section 6, "Writing Expressions for Retrieval and Update Queries."

Using Aggregate Functions

Each time a class or multivalued attribute is used as the argument of an aggregate function, a new reference variable is generated. The aggregate functions supported in OML are as follows:

Function	Purpose
AVG(<attribute>)	Returns the average value of the aggregate attribute.
COUNT(<attribute>)	Returns the number of occurrences of the aggregate attribute.
MIN(<attribute>)	Returns the minimum value of the aggregate attribute.
MAX(<attribute>)	Returns the maximum value of the aggregate attribute.
SUM(<attribute>)	Returns the total value of the aggregate attribute.

You can use aggregate functions to check for null values. For example, the following two expressions are equivalent:

```
COUNT(AGE) = 0

NOT AGE EXISTS
```

For more information on the EXISTS operator, refer to Section 6, "Writing Expressions for Retrieval and Update Queries."

The following example illustrates the effect of an aggregate function on the binding of names. This query returns the identification number of any manager who earns more than the average manager salary.

```
FROM MANAGER
RETRIEVE EMPLOYEE_ID
WHERE EMPLOYEE_SALARY > AVG(EMPLOYEE_SALARY OF MANAGER)
```

When the query is processed, one reference variable points to the MANAGER whose employee identification number is being retrieved, while another reference variable ranges over the MANAGER class to calculate the average salaries of all managers.

For more information about aggregate functions, refer to "Using Aggregate Functions" in Section 6, "Writing Expressions for Retrieval and Update Queries."

Using Path Expressions

A path expression defines a set of values by stringing together a list of EVAs. Both quantifiers and aggregate functions can be used with path expressions; if one is used, it must be enclosed in parentheses. For example,

```
WHERE AGE OF ALL(EMPLOYEES_MANAGING OF DEPT_MANAGERS)
         OF DEPARTMENT > 25
```

This form of path expression is referred to as an *explicit* path expression. The path expression EMPLOYEES_MANAGING OF DEPT_MANAGERS returns a set of entities in the EMPLOYEE class. These entities are then quantified by the path expression class DEPARTMENT. The path expression class is the class containing the rightmost EVA in the path expression. Therefore, the preceding global selection expression selects the departments in which all employees of all that department's managers are over the age of 25.

Suppose the path expression is changed to the following:

```
WHERE AGE OF ALL(EMPLOYEES_MANAGING) OF DEPT_MANAGERS
      OF DEPARTMENT > 25
```

The path expression EMPLOYEES_MANAGING returns a set of entities in the EMPLOYEE class. These entities are then quantified by the path expression class MANAGER. Therefore, the preceding global selection expression selects those manager/department pairs in which all employees of the manager are over the age of 25.

If you use a path expression without a quantifier or aggregate function, an implicit quantifier of SOME is applied; for example, the following global selection expression selects any department that has one or more employees over the age of 25:

```
WHERE AGE OF EMPLOYEES_MANAGING OF DEPT_MANAGERS
      OF DEPARTMENT > 25
```

If a bound attribute appears in the global selection expression, the range of the path expression is limited; for example, in the following query, the occurrence of the DEPT_MANAGERS attribute in the global selection expression is bound to the occurrence in the RETRIEVE clause. Therefore, the only employee identifications retrieved are the ones for department managers that meet the global selection expression—at least one of the department manager's employees is over the age of 25.

```
FROM DEPARTMENT
RETRIEVE DEPT_TITLE,
         EMPLOYEE_ID OF DEPT_MANAGERS
WHERE AGE OF EMPLOYEES_MANAGING OF DEPT_MANAGERS > 25
```

Using Subroles

An entity in a database can play more than one role. For example, a person can be both a project employee and a manager. The AS clause enables you to retrieve information about an entity in a role that would not normally be visible from the perspective class.

The AS clause uses the attributes of type subrole. The use of the AS clause depends on whether the subrole attribute is single-valued or multivalued, and on whether the attribute is an immediate or inherited attribute of the perspective class.

Using the AS clause enables you to retrieve the following:

- Information that is common to all the entities in a class
- Information that is particular to a specific entity because of the subrole of the entity

For example, by using the AS clause you can retrieve the following kinds of information about the employees in your company:

- Information about the employees in their role as employees
- Information about the employees in their role as managers
- Information about the employees in their role as project employees

Using the AS Clause with Single-Valued Subrole Attributes

If class A has a single-valued subrole attribute that you can use to place an entity in either class B or class C, you can use the AS clause in the following ways:

- To find an entity that belongs to both class A and class B
- To find an entity that belongs to both class A and class C

If you use the AS clause to find an entity that belongs to both class B and class C, an error results.

For example, a person can be either a current or previous employee. Therefore, you can use the AS clause to retrieve information about a person who is also an employee and about a person who is also a previous employee. You cannot, however, retrieve information about an employee who is a previous employee, because no single person can be both an employee and a previous employee simultaneously.

Using the AS Clause with Multivalued Subrole Attributes

If class A has a multivalued subrole attribute that you can use to place an entity in either or both class B and class C, you can use the AS clause in the following ways:

- To find an entity that is in both class A and class B
- To find an entity that is in both class A and class C
- To find an entity that is in both class B and class C

The following query returns the title of an employee who is also a manager and is over the age of 30:

```
FROM PROJECT_EMPLOYEE
RETRIEVE MANAGER_TITLE OF PROJECT_EMPLOYEE AS MANAGER,
         EMPLOYEE_ID OF PROJECT_EMPLOYEE
WHERE AGE OF PROJECT_EMPLOYEE > 30
```

In the example, the parent class of PROJECT_EMPLOYEE is EMPLOYEE. The EMPLOYEE class has the multivalued subrole attribute, PROFESSION. The PROFESSION attribute can take one, or both, of the values, PROJECT_EMPLOYEE and MANAGER. Therefore, an employee can be both a project employee and a manager. Consequently, immediate attributes of the MANAGER class can be accessed even if the perspective class is PROJECT_EMPLOYEE.

Note: *In the preceding example, the query returns the employee identification number of all project employees over the age of 30. A null value is returned for MANAGER_TITLE when a project employee is not a manager.*

Section 3
Structuring Retrieval Queries

A retrieval query has four main elements, as follows:

- An element that determines the perspective of the retrieval query

  You can designate the perspective class or classes explicitly by using the FROM clause or implicitly by using the appropriate element qualifications in the target list. The target list consists of the attributes and target attribute expressions included in the RETRIEVE clause.

  For more information on using the FROM clause, refer to "Using the FROM Clause to Define the Perspective Class" in this section. For more information on qualifying items in the target list, refer to "Qualifying Names in Retrieval Queries" in Section 2, "OML Language Conventions."

- An element that defines the items to be retrieved from the database

  The items can include both attributes and target expressions. For more information on target expressions, refer to the *InfoExec IQF Operations Guide*. The items included in this element of the query are referred to as the target list. The element itself is referred to as the RETRIEVE clause.

  For more information on the RETRIEVE clause, refer to "Using the RETRIEVE Clause to Define the Target List" in this section.

- Two elements that limit the data returned by the query

  - The WITH clause, or local filter expression, limits the appearance of particular target list items for individual entities. The expression does not limit the number of entities about which information is retrieved.

    For example, to return the names of all your employees but the personal identification numbers for only those employees who are not US citizens, you must apply a local filter expression to the PERSON_ID attribute.

  - The WHERE clause, or global selection expression, limits the entities selected by the query.

    For example, to return the names and personal identification numbers for all your employees who are not US citizens, you must use a global selection expression.

  For more information on limiting the information retrieved from the database, refer to "Using the WITH Clause to Create a Local Filter Expression" and "Using the WHERE Clause to Create a Global Selection Expression" in this section.

Appendix D, "Sample Retrieval Queries," provides examples of retrieval queries.

The following syntax diagram illustrates the format of a retrieval query:

Retrieval Query

```
──┬──────────────────────────────────┬─ RETRIEVE ──────────────────→
  │            ┌←─────────── , ─────┐│
  └─ FROM ─────┴─<perspective class>─┴┘

→─┬──────────────────────────────┬─┌←──────── , ─────┐
  │                              │ └─<target list>──┴──────────────→
  ├─ TABULAR ─┬──────────────────┤
  │           └─ DISTINCT ───────┤
  └─ STRUCTURED ─────────────────┘

→─ ORDERED BY ──<order items>──────────────────────────────────────→

→─┬─────────────────────────────────────────┬──────────────────────┤
  └─ WHERE ──<global selection expression>──┘
```

<perspective class>

```
  ┌←──────────────── , ──────────────┐
──┴─<class>─┬──────────────────────┬─┴─────────────────────────────┤
            └─ OF ──<database>─────┘
```

<target list>

```
  ┌←──────────────── , ────────────────────┐
──┴─┬─<attribute definition>───────────────┬─┴───────────────────────┤
    └─<target attribute expression>────────┘
```

<attribute definition>

```
──┬─<attribute>─┬─┬─────────────────────────────────┬──────────────→
  └─ * ─────────┘ │ ┌←───────────────────────────┐   │
                  └─┴─ OF ──<compound attribute>─┴───┘

→─┬────────────────┬─┬──────────────────┬─┬─────────────────────┬──→
  │ ┌←───────────┐ │ └─ OF ──<class>────┘ └─ OF ──<database>────┘
  └─┴─ OF ──<EVA>─┴─┘

→─┬───────────────────────────────────┬────────────────────────────┤
  └─ WITH ──<local filter expression>─┘
```

<order items>

```
──┬───────────────┬─┬─────────────┬─┬─<attribute>──────────────────┬─┤
  ├─ ASCENDING ───┤ ├─ COLLATING ─┤ └─<target list item number>────┘
  └─ DESCENDING ──┘ ├─ BINARY ────┤
                    └─ ORDERING ──┘
```

The remainder of this section explains the use of the

- FROM clause to define the perspective class
- RETRIEVE clause to define the target list
- WITH clause to create a local filter expression
- WHERE clause to create a global selection expression

For information concerning the TABULAR, STRUCTURED, DISTINCT, and ORDERED BY options, refer to Section 4, “Formatting the Output from Retrieval Queries.”

You form target attribute expressions by manipulating data in the database. In a target attribute expression, you can use any existing attributes with any of the constructs discussed in Section 6, “Writing Expressions for Retrieval and Update Queries.”

Using the FROM Clause to Define the Perspective Class

You can define the perspective of a query either explicitly, using the FROM clause, or implicitly by embedding the class name in the RETRIEVE clause. If you choose to declare one perspective class in the FROM clause, you must declare all the perspective classes in the FROM clause. The syntax for the FROM clause is as follows:

FROM Clause

```
          ┌←──────────── , ────────────┐
── FROM ──┴─<class>─┬──────────────────┬─┴──────────────────────────────┤
                    └─ OF ──<database>─┘
```

For example, to define EMPLOYEE as your perspective class, use the following statement:

```
FROM EMPLOYEE
```

If you want more than one perspective class, list the classes after the FROM keyword and separate them with commas (,). For example, to define EMPLOYEE, DEPARTMENT, and PROJECT as your perspective classes, use the following statement:

```
FROM EMPLOYEE, DEPARTMENT, PROJECT
```

The order of the perspective classes in the FROM clause is not significant.

Note: *Queries are more efficient if they have only one perspective class. In most cases, you can extract all the information you require, using a single perspective.*

Using the RETRIEVE Clause to Define the Target List

Rather than define your perspective class explicitly, as in the previous examples, you can define the class implicitly in the RETRIEVE clause. To define the perspective class implicitly, you must follow the rules for qualifying names. For more information on qualification, refer to “Qualifying Names in Retrieval Queries” in Section 2, “OML Language Conventions.” The syntax for the RETRIEVE clause is as follows:

RETRIEVE Clause

```
── RETRIEVE ─┬──────────────────┬─────────────────────────────────────→
             └─<output options>─┘

  ┌←─────────────── , ────────────────┐
→─┴─┬─<attribute definition>──────────┴───────────────────────────────┤
    └─<target attribute expression>───┘
```

The series of attribute definitions and target attribute expressions you provide form the *target list.* The target list contains a series of expressions that define the aspects of an entity you are interested in. The simplest form of expression is an attribute name, for example, the title of a project. You must use a comma (,) to separate the items in the target list. If you want to retrieve the title and number of a project, you can use either of the following queries:

Example 1: Perspective Class PROJECT Explicitly Stated

```
FROM PROJECT
RETRIEVE PROJECT_TITLE, PROJECT_NO
```

Example 2: Perspective Class PROJECT Implicitly Stated

```
RETRIEVE PROJECT_TITLE OF PROJECT, PROJECT_NO
```

The target list can include immediate, inherited, and extended attributes of the perspective class. Under the following conditions, immediate and inherited attributes must be qualified by the OF <class name> construct:

- If the FROM clause is omitted from the query
- If there are multiple perspective classes

Extended attributes must be qualified to show their relationship to the perspective class. For instance, if you want to retrieve project employee and manager salary information as well as project title and project number information, you should write your query as follows:

```
FROM PROJECT
RETRIEVE PROJECT_TITLE, PROJECT_NO,
         EMPLOYEE_SALARY OF PROJECT_TEAM,
         EMPLOYEE_SALARY OF PROJECT_MANAGER
```

You can manipulate the data being retrieved from the database by including the attributes in expressions. For instance, instead of retrieving the salaries of the project employees, you could choose to retrieve the maximum and minimum salaries as shown in the following example:

```
FROM PROJECT
RETRIEVE PROJECT_TITLE, PROJECT_NO,
         MAX(EMPLOYEE_SALARY OF PROJECT_TEAM),
         MIN(EMPLOYEE_SALARY OF PROJECT_TEAM),
         EMPLOYEE_SALARY OF PROJECT_MANAGER
```

You can use an asterisk (*) to indicate that all the immediate and inherited DVAs of the perspective class are to be returned. For example, the following two queries are equivalent; both return the number and title of a project and the title, number, and location of the department assigned the project:

Example 1

```
FROM PROJECT
RETRIEVE * OF PROJECT, * OF DEPT_ASSIGNED
```

Example 2

```
FROM PROJECT
RETRIEVE PROJECT_NO, PROJECT_TITLE,
         (DEPT_TITLE, DEPT_NO, DEPT_LOCATION) OF DEPT_ASSIGNED
```

Note: *You can abbreviate RETRIEVE to RET.*

For more information on the functions you can use in queries, refer to Section 6, “Writing Expressions for Retrieval and Update Queries.”

Limiting the Data Retrieved

You can choose to retrieve the items in the target list for all the entities in the perspective class. Alternatively, you can choose to limit the data retrieved by using either or both of the following methods:

- Limiting the items retrieved for a particular entity
- Limiting the entities retrieved

To limit the information returned for a particular entity, use the WITH clause. To limit the entities about which information is returned, use the WHERE clause. You can use both the WITH and WHERE clauses in the same query.

Using the WITH Clause to Create a Local Filter Expression

Using the WITH clause enables you to limit the information returned for a particular entity. The WITH clause can be used only within the target list, aggregate functions, and quantifier arguments. If the item qualified by the WITH clause does not meet the required condition, SIM returns a null value for the item.

The WITH clause has the following syntax:

WITH Clause

```
——<item to be affected>—— WITH —— ( ——<qualifying expression>—— ) —————┤
```

The qualifying expression can contain any immediate, inherited, or extended attributes of the item being affected by the WITH clause. If the item affected is in the perspective class, the qualification *OF <perspective class>* must precede the WITH clause. Note that the qualification of the affected item is implicitly added to any attribute references in the WITH clause.

You can use multiple WITH clauses in a query, as follows:

- You can embed WITH clauses within a qualification path.
- You can nest WITH clauses.

The expressions created with the WITH clause are referred to as *local filter expressions*.

For more information on the purpose of the WITH clause, refer to the discussion on local filter expressions in the *InfoExec IQF Operations Guide*.

For more information on the expressions and functions you can use in constructing a WITH clause, refer to Section 6, “Writing Expressions for Retrieval and Update Queries.”

The following three examples illustrate uses of the WITH clause:

Example 1: A WITH Clause

The following query retrieves project titles and numbers. The query also returns the salaries of the project employees if more than 10 employees are assigned to the project.

```
FROM PROJECT
RETRIEVE PROJECT_TITLE,
         PROJECT_NO,
         EMPLOYEE_SALARY OF PROJECT_TEAM OF PROJECT
            WITH(COUNT(PROJECT_TEAM) > 10)
```

Example 2: A WITH Clause in a Qualification Chain

The following query illustrates the use of a WITH clause in a qualification chain. The query retrieves the salaries of employees under the following circumstances:

- The employee is over the age of 25.
- The employee's manager earns more than $50,000.
- The employee's manager is assigned to a project.

```
FROM PROJECT
RETRIEVE EMPLOYEE_SALARY OF EMPLOYEES_MANAGING WITH (AGE > 25)
                OF PROJECT_MANAGER WITH (EMPLOYEE_SALARY > 50000)
```

Example 3: Nested WITH Clauses

The following query illustrates nested WITH clauses. Nested WITH clauses are most useful when you use aggregate functions. The query retrieves the name of the department assigned to a project if the average salary of the project managers over the age of 30 is greater than $50,000.

```
FROM PROJECT
RETRIEVE DEPT_TITLE OF DEPT_ASSIGNED
              WITH (AVG (EMPLOYEE_SALARY OF DEPT_MANAGERS
                         WITH (AGE > 30)
                        ) > 50000
                   )
```

Using the WHERE Clause to Create a Global Selection Expression

To restrict the entities returned by the query, you must include a WHERE clause in your query. The expression used in the WHERE clause is referred to as a *global selection expression.*

You can include only one WHERE clause in a query. As shown in the following example, position the WHERE clause after the target list.

The following example query returns the project number, title, and department for projects assigned more than 10 employees:

```
FROM PROJECT
RETRIEVE PROJECT_NO, PROJECT_TITLE,
         DEPT_TITLE OF DEPT_ASSIGNED
WHERE COUNT(PROJECT_TEAM) > 10
```

If a target list includes a multivalued attribute, some, all, or none of its values can be retrieved, depending on whether or not the attribute is bound to the global selection expression, as follows:

- If the multivalued attribute is not bound to the selection expression, the query returns all the entities to which the multivalued attribute points as long as at least one value meets the condition.
- If the multivalued attribute is bound to the selection expression, the query returns only those entities that meet the condition.

The following example has one multivalued attribute bound to the global selection expression and one multivalued attribute that is not bound. Therefore, this query returns the titles of all the projects managed by any manager whose employees earn more than $25,000. However, the query returns the employee identification numbers of only those employees who earn more than $25,000.

```
FROM MANAGER
RETRIEVE EMPLOYEE_ID OF EMPLOYEES_MANAGING,
         PROJECT_TITLE OF PROJECTS_MANAGING
WHERE EMPLOYEE_SALARY OF EMPLOYEES_MANAGING > 25000
```

The EMPLOYEE_ID OF EMPLOYEES_MANAGING attribute is multivalued with respect to the perspective class MANAGER. The occurrence of the EMPLOYEE_ID OF EMPLOYEES_MANAGING attribute in the target list is bound to its appearance in the global selection expression. The PROJECT_TITLE OF PROJECTS_MANAGING attribute is also multivalued with respect to MANAGER, but it is not bound to the global selection expression because the WHERE clause does not include any reference to the PROJECT class.

For more information on global selection expressions, refer to the discussion on defining expressions in the *InfoExec IQF Operations Guide.*

For more information on the operators and functions you can use in a global selection expression, refer to Section 6, “Writing Expressions for Retrieval and Update Queries.”

Section 4
Formatting the Output from Retrieval Queries

When you create a retrieval query, you should be aware of two types of formatting: formatting the report and formatting the output.

Formatting the report involves the following elements:

- The output medium, for example, disk, printer, or terminal
- The display mechanism, for example, graph or table
- The data format, for example, the number of decimal places used for numeric data

You can format your report programmatically, or you can use the formatting options available in IQF. OML is not involved in formatting the report.

You can use OML to select the method and order to be used to extract the data you requested from the database. The following output formatting is supported in OML:

- Tabular output
- Structured output
- Ordered output

You can use ordered output with structured or tabular output, but you cannot use both structured and tabular output in the same query. The default value is tabular output if you do not designate either structured or tabular output.

This section provides information on how to request tabular, structured, and ordered output. The report formatting for the examples in this section is the default formatting for a retrieval query entered on the DML screen of IQF. These report outputs show the different sequences of item values returned by SIM for the various OML output formats. These sequences of values are the same whether IQF or the SIM library entry points are used.

To produce the report shown in the text, you must make the SIM library calls according to certain logical rules for retrieving records; these rules are discussed in Section 7, “Using the SIM Library Entry Points.”

Requesting Tabular or Structured Output

The following syntax diagram illustrates the position the output format request takes in a retrieval query:

Output Format Request

```
──┬────────────────────────────────────────────┬─ RETRIEVE ──────────→
  │          ┌←─────────── , ───────────────┐  │
  └─ FROM ───┴─<class>──┬───────────────────┴──┘
                        └─ OF ──<database>──┘

→─┬────────────────────┬─┬─┬─<attribute definition>─────────┬─┬────────────┤
  └─<output format>────┘ │ └─<target attribute expression>──┘ │
                         └←─────────────── , ─────────────────┘
```

Tabular output is the default output format for a query. To explicitly request tabular output, insert the keyword TABLE after the keyword RETRIEVE in the target list, as in the following example:

```
FROM PROJECT
RETRIEVE TABLE PROJECT_NO, PROJECT_TITLE
```

To request structured output, insert the keyword STRUCTURE after the keyword RETRIEVE in the target list, as in the following example:

```
FROM PROJECT
RETRIEVE STRUCTURE PROJECT_NO, PROJECT_TITLE
```

Prevention of Duplicate Output

If you use tabular output, you can prevent duplicate sets of information from being returned by using the keyword DISTINCT, as shown in the following example:

```
FROM PROJECT
RETRIEVE TABLE DISTINCT PROJECT_NO, PROJECT_TITLE
```

Effect of Target Attribute Expressions on the Output

The following discussion shortens the terms *target attribute expression* and *attribute definition* to *target expression* for convenience. Target expressions are all the attributes and expressions in the target list that follow the keyword RETRIEVE. Query output varies depending on the following:

- The type of output you request—tabular or structured
- The number of single-valued or multivalued target expressions included in your query
- The relationship between the target expressions in your query

Effect of Single-Valued Target Expressions

A target expression is considered to be single-valued under one of the following conditions:

- The target expression is a single-valued attribute that is immediate to or inherited by the perspective class.

  In the following example, PROJECT_NO is a single-valued target expression because it is an immediate, single-valued attribute of the perspective class PROJECT:

  ```
  FROM PROJECT
  RETRIEVE PROJECT_NO
  ```

- The target expression is an extended, single-valued attribute for which the entity-valued attribute (EVA) connecting the extended attribute to the perspective class is single-valued.

  In the following example, PROJECT_TITLE OF PROJECT_OF is a single-valued target expression because PROJECT_TITLE is a single-valued, extended attribute that is connected to the perspective class ASSIGNMENT through the single-valued EVA *PROJECT_OF*:

  ```
  FROM ASSIGNMENT
  RETRIEVE ASSIGNMENT_NO,
       PROJECT_TITLE OF PROJECT_OF
  ```

Two single-valued target expressions can be either dependent on or independent of each other based on the following conditions:

- The target expressions are dependent if they are in the same perspective class or are extended attributes connected by the same EVA to the perspective class. In the following example, ASSIGNMENT_NO and RATING are dependent on each other because they are both in the perspective class ASSIGNMENT. The target expressions PROJECT_NO OF PROJECT_OF and PROJECT_TITLE OF PROJECT_OF are dependent on each other because they are both extended attributes connected by the EVA *PROJECT_OF*.

  ```
  FROM ASSIGNMENT
  RETRIEVE ASSIGNMENT_NO, RATING,
       PROJECT_NO OF PROJECT_OF, PROJECT_TITLE OF PROJECT_OF
  ```

- The target expressions are independent if the two target expressions are extended attributes connected to the perspective class by different EVAs. In the following example, PROJECT_TITLE OF PROJECT_OF and TITLE OF STAFF_ASSIGNED are independent of each other because they are extended attributes, and each is connected to the perspective class through different EVAs: *PROJECT_OF* and *STAFF_ASSIGNED*.

  ```
  FROM ASSIGNMENT
  RETRIEVE ASSIGNMENT_NO,
       PROJECT_TITLE OF PROJECT_OF, TITLE OF STAFF_ASSIGNED
  ```

Queries that include single-valued target expressions produce output as follows:

- For either tabular or structured output, each retrieval returns the values for dependent single-valued target expressions in the same record.
- For structured output, each retrieval returns the values of two or more independent single-valued target expressions in separate records.

The following text describes examples of these previously described effects.

Example of Dependent Single-Valued Target Expressions

Assume you want to retrieve the project number and the name of a project. You would write the query as follows:

```
FROM PROJECT
RETRIEVE TABLE
    PROJECT_NO, PROJECT_TITLE
```

If you change the query to request structured output, the query would look as follows:

```
FROM PROJECT
RETRIEVE STRUCTURE
    PROJECT_NO, PROJECT_TITLE
```

PROJECT_TITLE and PROJECT_NO are single-valued target expressions because they are single-valued attributes in the perspective class PROJECT. Both target expressions are dependent on each other.

Table 4–1 illustrates the output generated for both queries. Both a tabular and a structured query produce the values of the dependent single-valued target expressions in the same record. Because the following report was produced by using IQF, values in the same record appear on the same line.

Table 4–1. Tabular Output for a Query Including Single-Valued Target Expressions

Project Number	Project Title
101	Camelot
102	Excalibur
103	Gallahad
201	Camelot1
202	Excalibur1
203	Gallahad1

Example of Independent Single-Valued Target Expressions

The following example includes two independent single-valued target expressions: PROJECT_TITLE and TITLE.

PROJECT_TITLE is a single-valued target expression because the single-valued EVA *PROJECT_OF* connects it to the perspective class ASSIGNMENT.

TITLE is a single-valued target expression because the single-valued EVA *STAFF_ASSIGNED* connects it to the perspective class ASSIGNMENT.

PROJECT_TITLE and TITLE are independent of each other because they are connected to the perspective class by different EVAs. Tabular output does not show this independent relationship; structured output does.

The following query returns tabular output:

```
FROM ASSIGNMENT
RETRIEVE TABLE
    ASSIGNMENT_NO,
    PROJECT_TITLE OF PROJECT_OF, TITLE OF STAFF_ASSIGNED
```

Table 4–2 illustrates the tabular output generated for this query. The query returns the values of all the target expressions in the same record. Because the following report was produced by using IQF, values in the same record appear on the same line. The blank values under Project Title and Staff Title show that the target expression values are null.

Table 4–2. Tabular Output for a Query Including Independent Single-Valued Target Expressions

Assignment Number	Project Title	Staff Title
2116218	Camelot	SENIOR
2118156	Camelot	
2113689	Camelot	SPECIALIST
2111365	Camelot	JUNIOR
2112153		
2212279		JUNIOR

The following query returns structured output:

```
FROM ASSIGNMENT
RETRIEVE STRUCTURE
    ASSIGNMENT_NO,
    PROJECT_TITLE OF PROJECT_OF, TITLE OF STAFF_ASSIGNED
```

Table 4–3 illustrates the structured output generated for this query. The query returns the values of the independent single-valued target expressions under Project Title and Staff Title in separate records. Because the following report was produced by using IQF, values in separate records appear on different lines. Thus, blank values do not always show that the target expression values are null. Blank values correspond to null values only as follows:

- Assignment numbers 2118156 and 2112153 have null values under Staff Title.
- Assignment numbers 2112153 and 2212279 have null values under Project Title.

Table 4–3. Structured Output for a Query Including Two Independent Single-Valued Target Expressions

Assignment Number	Project Title	Staff Title
2116218	Camelot	
		SENIOR
2118156	Camelot	
2113689	Camelot	
		SPECIALIST
2111365	Camelot	
		JUNIOR
2112153		
2212279		JUNIOR

Effect of Multivalued Target Expressions

A target expression is considered to be multivalued under one of the following conditions:

- The target expression is a multivalued attribute or a part of a multivalued compound attribute that is immediate to or inherited by the perspective class.

  In the following example, GPA OF EDUCATION is a multivalued target expression because GPA is part of the immediate compound attribute EDUCATION:

  ```
  FROM PERSON
  RETRIEVE LAST_NAME OF NAME,
       GPA OF EDUCATION
  ```

- The target expression is a multivalued extended attribute or is connected to the perspective class by a multivalued EVA.

  In the following example, although PROJECT_NO is a single-valued attribute, it is a multivalued target expression because it is connected to the perspective class PROJECT_EMPLOYEE by the multivalued EVA *CURRENT_PROJECT*:

  ```
  FROM PROJECT_EMPLOYEE
  RETRIEVE LAST_NAME OF NAME,
       PROJECT_NO OF CURRENT_PROJECT
  ```

Two multivalued target expressions can be either dependent on or independent of each other based on the following conditions:

- The target expressions are dependent if both are in the same compound attribute or if both are single-valued, extended attributes in the same class and are connected to the perspective class by the same multivalued EVA.

  In the following example, DEGREE_OBTAINED OF EDUCATION and GPA OF EDUCATION are dependent because DEGREE_OBTAINED and GPA are part of the same compound attribute EDUCATION. In addition, PROJECT_NO and PROJECT_TITLE are dependent on each other because both are single-valued extended attributes from the class PROJECT and both are connected to the perspective class PROJECT_EMPLOYEE by the multivalued EVA *CURRENT_PROJECT*.

  ```
  FROM PROJECT_EMPLOYEE
  RETRIEVE LAST_NAME OF NAME,
       DEGREE_OBTAINED OF EDUCATION, GPA OF EDUCATION,
       PROJECT_NO OF CURRENT PROJECT, PROJECT_TITLE OF CURRENT PROJECT
  ```

- The target expressions are independent if either of the two target expressions is declared as a multivalued attribute or if they are connected to the perspective class by different multivalued EVAs.

  In the following example, PROJECT_TITLE and LAST_NAME are independent because they are connected to the perspective class MANAGER by two different multivalued EVAs, *PROJECTS_MANAGING* and *EMPLOYEES_MANAGING*:

  ```
  FROM MANAGER
  RETRIEVE MANAGER_TITLE,
       PROJECT_TITLE OF PROJECTS_MANAGING,
       LAST_NAME OF NAME OF EMPLOYEES_MANAGING
  ```

Queries with multivalued target expressions produce output as follows until no more data exists to return:

- The value of only one multivalued target expression is returned in a record.
- The values of two or more dependent multivalued target expressions are returned in the same record. Each retrieval returns the next occurrence of the dependent expressions until none exist.
- The values of two or more independent multivalued target expressions are returned in separate records.

  That is, in a report produced by using IQF for two independent multivalued target expressions, each line shows a value of the first expression and a blank for the second expression until no value for the first expression exists. Then, each line shows a blank for the first expression and a value for the second expression until no values for the second expression exist.

For tabular output, each retrieval repeats single-valued target expressions for each occurrence of a multivalued target expression. For structured output, each retrieval returns only one occurrence of each single-valued target expression for each entity in the perspective class. Blank values are displayed for all other occurrences of multivalued target expressions.

The following text describes examples of the effects of multivalued target expressions.

Example of One Multivalued Target Expression

The following query includes the multivalued target expression EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF PROJECT_MANAGER. In this case, EMPLOYEE_SALARY is a single-valued attribute extended from the class EMPLOYEE, which is connected to the perspective class PROJECT. The connection is through the multivalued EVA *EMPLOYEES_MANAGING* by way of the single-valued EVA *PROJECT_MANAGER*.

```
FROM PROJECT
RETRIEVE TABLE PROJECT_TITLE, MANAGER_TITLE OF PROJECT_MANAGER,
               EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF PROJECT_MANAGER
```

Table 4–4 illustrates the tabular output generated for the query. The query repeats the values of the single-valued target expressions under Project Title and Manager Title once for each value in the multivalued target expression under Employee Salary.

Table 4–4. Tabular Output for a Query Including One Multivalued Target Expression

Project Title	Manager Title	Employee Salary
Excalibur	Supervisor	15000
Excalibur	Supervisor	17500
Excalibur	Supervisor	14000
Excalibur	Supervisor	20000
Camelot	Executive	50000
Camelot	Executive	60000
Camelot	Executive	45000

Table 4–5 illustrates the structured output generated for the query.

Table 4–5. Structured Output for a Query Including One Multivalued Target Expression

Project Title	Manager Title	Employee Salary
Excalibur	Supervisor	15000
		17500
		14000
		20000
Camelot	Executive	50000
		60000
		45000

Example of Two Dependent Multivalued Target Expressions

The following query uses two dependent multivalued target expressions: PROJECT_NO OF CURRENT_PROJECT and PROJECT_TITLE OF CURRENT_PROJECT.

Although PROJECT_NO and PROJECT_TITLE are single-valued attributes of the extended class PROJECT, they become multivalued target expressions because PROJECT is connected to the perspective class PROJECT_EMPLOYEE through the multivalued EVA *CURRENT_PROJECT*.

These two target expressions are dependent on each other because they are single-valued extended attributes from the same class, PROJECT, and are connected to the perspective class PROJECT_EMPLOYEE by the same multivalued EVA *CURRENT_PROJECT*.

The following query returns tabular output:

```
FROM PROJECT_EMPLOYEE
RETRIEVE LAST_NAME OF NAME,
    PROJECT_NO OF CURRENT_PROJECT,
    PROJECT_TITLE OF CURRENT_PROJECT
```

Table 4–6 illustrates the tabular output generated for the query. The query repeats each value of the single-valued target expression under Last Name for each pair of values in the dependent multivalued target expression under Project Number and Project Name.

Table 4–6. Tabular Output for a Query Including Two Dependent Multivalued Target Expressions

Last Name	Project Number	Project Name
Carlin	101	Camelot
Carlin	202	Excalibur1
Aquino	102	Excalibur
Aquino	202	Excalibur1
Reinholtz	202	Excalibur1
Reinholtz	103	Gallahad
Reinholtz	203	Gallahad1

Table 4–7 illustrates the structured output generated for the query. The query returns each value of the single-valued target expression under Last Name once for each pair of values in the dependent multivalued target expressions under Project Number and Project Name.

Table 4–7. Structured Output for a Query Including Two Dependent Multivalued Target Expressions

Last Name	Project Number	Project Name
Carlin	101	Camelot
	202	Excalibur1
Aquino	102	Excalibur
	202	Excalibur1
Reinholtz	202	Excalibur1
	103	Gallahad
	203	Gallahad1

Example of Two Independent Multivalued Target Expressions

In the following query, PROJECT_TITLE OF PROJECTS_MANAGING is a multivalued target expression because it is an attribute connected to the perspective class MANAGER by a multivalued EVA *PROJECTS_MANAGING*.

LAST_NAME OF NAME is a multivalued target expression because it is an attribute connected to the perspective class MANAGER by a multivalued EVA *EMPLOYEES_MANAGING*.

These two target expressions are independent of each other because they are connected to the perspective class by different EVAs.

The following query returns the output illustrated in Table 4–8:

```
FROM MANAGER
RETRIEVE MANAGER_TITLE,
         PROJECT_TITLE OF PROJECTS_MANAGING,
         LAST_NAME OF NAME OF EMPLOYEES_MANAGING
```

Table 4–8 illustrates the tabular output generated for this query. The query returns the values of the independent multivalued target expression under Project Title and Employee Last Name in separate records. Because the following report was produced by using IQF, values in separate records appear on different lines. Each value of the single-valued target expression under Manager Title is repeated once for each value of the multivalued target expressions.

Table 4–8. Tabular Output for a Query Including Two Independent Multivalued Target Expressions

Manager Title	Project Title	Employee Last Name
Dept_Manager	Excalibur	
Dept_Manager	Excalibur2	
Dept_Manager		Feverman
Dept_Manager		Smythe
Dept_Manager		Roget
Supervisor	Camelot	
Supervisor	Camelot1	
Supervisor		Carrey
Supervisor		Lani
Supervisor		Crawford
Supervisor		Sitar

Table 4–9 illustrates the structured output generated for the same query. The query returns the values of the independent multivalued target expression under Project Title and Employee Last Name in separate records. Because the following report was produced by IQF, values in separate records appear on different lines. Each value of the single-valued target expression under Manager Title is produced only once.

Table 4–9. Structured Output for a Query Including Two Independent Multivalued Target Expressions

Manager Title	Project Title	Employee Last Name
Dept_Manager	Excalibur	
	Excalibur2	
		Feverman
		Smythe
		Roget
Supervisor	Camelot	
	Camelot1	
		Carrey
		Lani
		Crawford
		Sitar

Effect of Multiple Perspective Classes on the Output

If your query includes more than one perspective class, the output is identical for structured and tabular output. In both cases, the query returns each piece of data several times. The data repetition occurs because a matrix is formed for the data retrieval. The requested items are retrieved from every possible combination of entities in the perspective classes, as long as that combination meets the global selection expression criteria. If there is no global selection expression, then the query returns a complete cross-product of entities.

The following query returns the output shown in Table 4–10. In this instance, if you request structured output, the query still returns the information shown in Table 4–10.

```
FROM MANAGER, PROJECT_EMPLOYEE
RETRIEVE MANAGER_TITLE OF MANAGER,
         TITLE OF PROJECT_EMPLOYEE
WHERE EMPLOYEE_SALARY OF MANAGER = EMPLOYEE_SALARY OF PROJECT_EMPLOYEE
```

Table 4–10. Effect of Multiple Perspective Classes on Tabular and Structured Output

Manager Title	Project Employee Title
Dept_Manager	Staff
Dept_Manager	Specialist
Dept_Manager	Junior
Div_Manager	Staff
Div_Manager	Specialist
Div_Manager	Junior
Executive	Staff
Executive	Specialist
Executive	Junior
Supervisor	Staff
Supervisor	Specialist
Supervisor	Junior
Dept_Manager	Staff
Div_Manager	Staff
Executive	Staff
Supervisor	Staff
Dept_Manager	Specialist
Div_Manager	Specialist
Executive	Specialist
Supervisor	Specialist
Dept_Manager	Junior
Div_Manager	Junior
Executive	Junior
Supervisor	Junior

Effect of the TRANSITIVE Function on the Output

You use the TRANSITIVE function to retrieve layered information, such as the names of managers in a branch of an organization. The presence of the TRANSITIVE function causes SIM to treat the query as if it were a sequence of retrieval queries. In the query sequence, replace the TRANSITIVE and END elements with a repetitive path expression. The path expression is repeated the number of times indicated by the END statement.

The following diagram illustrates the syntax of the TRANSITIVE function:

TRANSITIVE Function

```
–– TRANSITIVE –– ( ––<path expression>–– END –– LEVEL –– = ––<integer>––→

→– ) ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––┤
```

The following query returns information about the three levels of management above the manager whose last name is Lazer:

```
FROM MANAGER
RETRIEVE MANAGER_TITLE,
         NAME OF TRANSITIVE(EMPLOYEE_MANAGER END LEVEL=3)
WHERE LAST_NAME OF NAME OF MANAGER = "Lazer"
```

The preceding query is equivalent to the following sequence of queries:

```
FROM MANAGER
RETRIEVE MANAGER_TITLE, NAME OF (EMPLOYEE_MANAGER)
WHERE LAST_NAME OF NAME OF MANAGER = "Lazer"

FROM MANAGER
RETRIEVE MANAGER_TITLE, NAME OF (EMPLOYEE_MANAGER OF
                                 EMPLOYEE_MANAGER)
WHERE LAST_NAME OF NAME OF MANAGER = "Lazer"

FROM MANAGER
RETRIEVE MANAGER_TITLE, NAME OF (EMPLOYEE_MANAGER OF
                                 EMPLOYEE_MANAGER OF
                                 EMPLOYEE_MANAGER)
WHERE LAST_NAME OF NAME OF MANAGER = "Lazer"
```

The output from the query is shown in Table 4–11. From the table you can see that Lazer is managed by Susan Arne, Susan Arne is managed by John Paramo, and John Paramo is managed by Karen Perr. The query does not return the name of Karen Perr's manager because the END clause limits the request to three levels of management.

Table 4–11. Output from a Query Including the TRANSITIVE Function

Manager Title	First Name	Middle Initial	Last Name
Div_Manager	Susan	J	Arne
Dept_Manager	John	X	Paramo
Executive	Karen	D	Perr

If one of the queries does not return a manager's name, that is, returns a null value, the processing stops and the query returns the data extracted up until that point. If you omit the END clause from the query, the processing continues until a null value is returned.

You can use the TRANSITIVE function with both structured and tabular output, but it is more effective with tabular output. For more information on the TRANSITIVE function, refer to Section 6, "Writing Expressions for Retrieval and Update Queries."

Sorting Entities

Use the ORDERED BY clause to select the order in which the entities defined by your query are retrieved from the database. You can use the ORDERED BY clause with both tabular and structured output.

You can choose to sort information in either ascending or descending order. In ascending order, the lower values are retrieved first. In descending order, the data is returned with the highest values first. If you do not explicitly choose ascending or descending order, your output is sorted in ascending order, the default value.

In addition, you can choose to sort your string data, using any of the following sorting sequences:

- Binary

  An ordering sequence that sorts data according to the binary code value of the characters in the data.

- Ordering

  An ordering sequence that sorts data according to the ordering-sequence values assigned at your site. The same ordering-sequence value might be assigned to more than one character.

- Collating

  An ordering sequence that sorts data according to the ordering-sequence values and priority-sequence values assigned at your site.

Note: *Attributes with a base data type of string are ordered according to their collating sequence.*

If you do not explicitly choose binary, ordering, or collating sequencing, your output is sorted according to collating sequencing. For ASeriesNative strings, the binary, ordering, and collating sequences are identical and appear in binary sequence.

To sort your data, you append the ORDERED BY clause to the end of the target list (before the WHERE clause if one is included). The attributes you use in the ORDERED BY clause need not appear in the target list, but they must be related to the perspective class. You can sort the data by using the values of several attributes. For example, you can sort data by last name and then by first name. If you have more than one attribute in the ORDERED BY clause, the expression is evaluated from left to right.

The ORDERED BY clause has the following format:

ORDERED BY Clause

```
                  ┌←──────────────┐
── ORDERED ── BY ─┴─<order items>─┴──────────────────────────────────┤
```

<order items>

```
──┬──────────────┬─┬─────────────┬─┬─<attribute>──────────────────┬───────┤
  ├─ ASCENDING ──┤ ├─ COLLATING ─┤ └─<target list item number>────┘
  └─ DESCENDING ─┘ ├─ BINARY ────┤
                   └─ ORDERING ──┘
```

To include one of the attributes from the target list in your ORDERED BY clause, you can use its position in the target list rather than spelling out the attribute name or repeating the expression. Assume you use the following query:

```
FROM PROJECT_EMPLOYEE
RETRIEVE TITLE,
        (EMPLOYEE_SALARY /
           COUNT(PROJECT_TITLE OF CURRENT_PROJECT)
        )
ORDERED BY ASCENDING 2
WHERE PROJECT_TITLE OF CURRENT_PROJECT = "Excalibur"
```

The output from the query is ordered according to the results of the following expression:

```
EMPLOYEE_SALARY / COUNT(PROJECT_TITLE OF CURRENT_PROJECT)
```

If a multivalued attribute occurs in a query using ordered tabular output, the entities that the attribute refers to do not necessarily appear in consecutive order in the output. For example, the following query returns the output shown in Table 4–12:

```
FROM PROJECT
RETRIEVE PROJECT_TITLE, MANAGER_TITLE OF PROJECT_MANAGER,
         EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF PROJECT_MANAGER
ORDERED BY ASCENDING
      EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF PROJECT_MANAGER
```

Table 4–12. Ordered Tabular Output for a Query Including Multivalued Attributes

Project Title	Manager Title	Employee Salary
Fort Field	Supervisor	25000
Fort Field	Supervisor	35000
North Field	Executive	35500
Fort Field	Supervisor	36000
North Field	Executive	45000
North Field	Executive	60000
Fort Field	Supervisor	65000

If the ORDERED BY clause in the previous query is omitted, the output appears as shown in Table 4–13. In the table, all the entities with the same project title and manager title are grouped together.

Table 4–13. Unordered Tabular Output for a Query Including Multivalued Attributes

Project Title	Manager Title	Employee Salary
Fort Field	Supervisor	25000
Fort Field	Supervisor	35000
Fort Field	Supervisor	36000
Fort Field	Supervisor	65000
North Field	Executive	35500
North Field	Executive	45000
North Field	Executive	60000

Section 5 Structuring Update Queries

The two basic forms of update queries do either of the following:

- Alter entity information.
- Add and delete entities from classes.

The structure of an update query is different from that of a retrieval query in two ways:

- Only one perspective class is allowed.
- The perspective class is not declared in a FROM clause.

Appendix E, “Sample Update Queries,” provides a task-oriented description for performing update queries. Appendix F, “Quick Reference Query Syntax,” provides the syntax diagrams for update queries in a quick-reference format.

Understanding Restrictions on Modifications

The following restrictions apply to all types of modifications:

- You cannot alter extended attributes.
- You must ensure that the type of the expression used to denote a new value and the type of the attribute being altered are compatible. For example, you cannot assign a Boolean value to an attribute of type date, but you can assign an integer value to an attribute of type real.

  In addition, you must use the correct character set when you assign string data to attributes.
- You must ensure that the number of entities affected by the update query does not exceed the value designated by the LIMIT clause.
- You can update a maximum of one entity if the LIMIT clause is omitted from the query.
- You must include a WHERE clause in all update queries unless you are updating class attributes. You can use the WHERE TRUE statement if the update applies to all the entities.
- You must ensure that the result of a modification query is compatible with all the declared attribute options. Attribute options include details such as a range of allowable values, the maximum number of occurrences allowed, and whether the attribute is required.
- You can only use values in the database that existed before the start of your query; that is, the update query cannot see the results of its own work. For example, assume your query performs the following:
  - Changes the marital status of 10 employees to married
  - Alters the salaries of all married employees by 3 percent

  The salaries of the 10 employees whose marital status you change are not increased by 3 percent.

  Note: *Databases that have been processed by DMS.View have certain restrictions and behavioral patterns that differ from those of a native SIM database. For detailed information, refer to the* InfoExec DMS.View Operations Guide.

Modifying Entities

Use a query statement prefixed by the MODIFY keyword to add, alter, or delete attribute values from existing database entities.

The syntax you use depends on whether you want to add, modify, or delete information. Also, the syntax you use is determined by the types of attribute being affected by the update: single-valued or multivalued, compound, data-valued, or entity-valued. The following syntax diagram shows the basic format of an update query for altering existing entity information:

Basic Update Query

```
── MODIFY ─┬───────────────────────────┬─<class>─┬──────────────────┬──→
           └ LIMIT ── = ─┬─ ALL ───────┤         └ OF ──<database>──┘
                         └─<integer>───┘

            ┌←──────────────────────── , ─────────────────────────┐
→─ (────────┴─<attribute>──:= ─┬─────────────┬─┬────────────────┬─┴─) ────────→
                               ├─ EXCLUDE ───┤ └─<expression1>──┘
                               └─ INCLUDE ───┘

→─ WHERE ─┬─ TRUE ─────────────┬─────────────────────────────────────────┤
          └─<expression2>──────┘
```

Limiting the Number of Entities Changed

Use the LIMIT clause to designate the maximum number of entities that can be modified by a query. If the limit is exceeded, the query is rejected and the database is left unchanged. By default, a value of 1 is assigned to the LIMIT clause. Assigning the value ALL to the LIMIT clause enables you to alter an unlimited number of entities.

The following query increases the salary of the employee Joe Hyams by 10 percent. Because the LIMIT clause is omitted from the query, the default value of 1 is used. Consequently, no change is made to the database if there is more than one employee called Joe Hyams.

```
MODIFY EMPLOYEE
       (EMPLOYEE_SALARY:= EMPLOYEE_SALARY * 1.1)
WHERE LAST_NAME OF NAME = "Hyams" AND FIRST_NAME OF NAME = "Joe"
```

The following query gives all employees a 10 percent pay raise:

```
MODIFY LIMIT=ALL EMPLOYEE
       (EMPLOYEE_SALARY:= EMPLOYEE_SALARY * 1.1)
WHERE TRUE
```

The following query gives a pay raise of 10 percent to all the project employees assigned to the project Camelot. When the query is processed, a check is made to ensure that the salary change is limited to 15 entities.

```
MODIFY LIMIT=15 PROJECT_EMPLOYEE
       (EMPLOYEE_SALARY:= EMPLOYEE_SALARY * 1.1)
WHERE PROJECT_TITLE OF CURRENT_PROJECT = "Camelot"
```

Notes:

- *The symbol := is used to assign values to attributes.*
- *The symbol = is used to assign a limit or, in an expression, to equate values.*

Adding Values to Attributes

The syntax for adding values to single-valued attributes and to multivalued attributes is different. It is described in the following text.

Adding Values to Single-Valued Attributes

Use the following syntax to add a value to a single-valued attribute:

```
── MODIFY ─┬──────────────────────────┬─<class>─┬───────────────────┬───→
           └─ LIMIT ── = ─┬─ ALL ─────┤         └─ OF ──<database>──┘
                          └─<integer>─┘

            ┌←──────────────────, ───────────────┐
→─ (───┴─<attribute>──:= ──<expression1>─┴─) ── WHERE ────────────────→

→─┬─ TRUE ─────────┬──────────────────────────────────────────────────┤
  └─<expression2>──┘
```

Use expression1 to assign values to the attribute. Also, expression1 can contain attributes.

Note: *If you use an attribute whose values are changing, use the existing values. As seen in the following example, if the existing value of EMPLOYEE_SALARY is 30000, all occurrences of EMPLOYEE_SALARY on the right-hand side of an assignment statement equate to 30000 even though the first statement in the query alters the EMPLOYEE_SALARY value to 33000:*

```
MODIFY LIMIT=ALL EMPLOYEE
       (EMPLOYEE_SALARY:= EMPLOYEE_SALARY * 1.1,
        EMPLOYEE_ID:= EMPLOYEE_SALARY
       )
WHERE TRUE
```

Adding Values to Multivalued Attributes

Use the INCLUDE clause to add values to multivalued attributes. The change does not affect any of the existing values of the attribute. If, however, the change would cause too many values to be assigned to the attribute or if any other attribute option would be invalidated, the update query is rejected. If the update of the multivalued attribute is rejected, the other changes requested in the query are not carried out even if they are valid.

The following is the syntax for adding a value to a multivalued attribute:

```
── MODIFY ─┬──────────────────────────┬─<class>─┬──────────────────┬─ (──→
           └─ LIMIT ── = ─┬─ ALL ─────┘         └─ OF ──<database>─┘
                          └─<integer>─┘

  ┌←──────────────────────────────────── , ─────────────────────────────┐
  │                                               ┌←────── , ─────┐     │
→─┴─<multivalued attribute>── := ── INCLUDE ── (──┴─<expression1>─┴─) ──┴──→

→─) ── WHERE ──┬─ TRUE ────────┬───────────────────────────────────────────┤
               └─<expression2>─┘
```

In your update query, expression1 can be replaced by a single value or by a series of values separated by commas. If the attribute is a compound multivalued attribute, expression1 can also be replaced by one or more assignments to the components of the attribute.

The following example adds a Master's degree to the information about the employee Fred Mercer; it also changes his title to specialist:

```
MODIFY PROJECT_EMPLOYEE
      (EDUCATION:= INCLUDE (DEGREE_OBTAINED:= MS,
                           YEAR_OBTAINED:= 1/1/88,
                           GPA:= 3.9),
      TITLE:= Specialist)
WHERE LAST_NAME OF NAME = "Mercer" AND FIRST_NAME OF NAME = "Fred"
```

Note: *When you process a DMSII database using DMS.View, all occurrences of a multivalued attribute might be assigned a default value during the processing. If this situation occurs and you try to add a value to the multivalued attribute, an error occurs. To avoid this problem, first delete one occurrence of the multivalued attribute for each value that you want to add.*

For more information on using DMS.View, refer to the InfoExec DMS.View Operations Guide.

Deleting Values from Attributes

Use the EXCLUDE clause to delete values from both single-valued and multivalued attributes. The following text provides detailed information on using the EXCLUDE clause.

Deleting Values from Single-Valued Attributes

Use the following syntax to delete values from single-valued DVAs or EVAs:

```
── MODIFY ──┬──────────────────────────┬─<class>─┬────────────────────┬───→
            └─ LIMIT ── = ─┬─ ALL ─────┤         └─ OF ──<database>───┘
                           └─<integer>─┘

           ┌←──────────────── , ────────────────┐
→─ ( ──┴─<attribute>── := ── EXCLUDE ──┴─ ) ── WHERE ─┬─ TRUE ─────────┬───┤
                                                      └─<expression>───┘
```

When you delete a single-valued attribute, its value becomes null. You cannot delete the value of a required attribute unless you also delete the entity. For example, the DEPT_NO attribute is a required attribute of the DEPARTMENT class. Therefore you can only delete a department number if you also delete the remainder of the information that relates to the department.

Deleting Values from Multivalued Attributes

To delete all the values associated with a multivalued attribute, use the same syntax as that for a single-valued attribute.

In an update query, you can designate two different types of LIMIT clauses as follows:

- A LIMIT clause that designates the maximum number of entities that can be altered.

  To achieve this effect, place the LIMIT clause immediately after the MODIFY keyword. If you omit this statement, your query can update only one entity.

- A LIMIT clause that designates the number of occurrences of a particular multivalued attribute that can be deleted.

  To achieve this effect, place the LIMIT clause after the EXCLUDE keyword. You can use the LIMIT clause once for each attribute affected by the update. If you omit this statement, your query can update only one occurrence of each attribute.

Use the following syntax to delete values from a multivalued attribute:

```
── MODIFY ─┬────────────────────────────┬─<class>─┬──────────────────┬───→
           └─ LIMIT ── = ─┬─ ALL ──────┤          └─ OF ──<database>─┘
                          └─<integer>──┘

→─ (──<expression defining occurrences>──) ── WHERE ───────────────→

→─┬─ TRUE ──────────┬─────────────────────────────────────────────────┤
  └─<expression1>───┘
```

<expression defining occurrences>

```
  ┌←─────────────────────────────── , ───────────────────────────────┐
──┴─<attribute>── := ── EXCLUDE ─┬─────────────────┬─┬──────────────────┬┴─┤
                                 └─<limit clause>──┘ └─<local filter>───┘
```

<limit clause>

```
── LIMIT ── = ──<integer>───────────────────────────────────────────┤
```

<local filter>

```
──<attribute>── WITH ── (──<expression2>──) ────────────────────────┤
```

If you omit the *LIMIT* = *<integer>* construct, you can delete only one occurrence of the attribute. You can use the *WITH <expression2>* construct to designate the occurrences that you want to delete.

The following examples illustrate the different ways to remove values from a multivalued attribute.

Example 1: Removing All Values from a Multivalued Attribute

For the employee with the identification number 12345678, the following query removes all information relating to education:

```
MODIFY EMPLOYEE
       (EDUCATION:= EXCLUDE LIMIT=ALL)
WHERE EMPLOYEE_ID = 12345678
```

In this query, the multivalued attribute is qualified by a LIMIT clause. Using the LIMIT clause ensures that the query is not rejected if the multivalued attribute occurs more than once.

The query also fails if there is more than one employee with the identification number 12345678 because, by default, only one entity can be affected by an update query. To alter more than one entity, you must include a LIMIT clause immediately after the MODIFY keyword.

Example 2: Removing a Single Value from a Multivalued Attribute

The employee with the identification number 12345678 has the following education record:

- Bachelor's degree, GPA 3.5
- Master's degree, GPA 4.0
- Doctorate, GPA 3.0

The following query removes the education information relating to the employee's doctorate. The remainder of the employee's information is not affected by the query.

```
MODIFY EMPLOYEE
       (EDUCATION:= EXCLUDE EDUCATION WITH (GPA = 3.0))
WHERE EMPLOYEE_ID = 12345678
```

Example 3: Removing Multiple Values from a Multivalued Attribute

The following query removes up to five occurrences of the education information for the employee who has the identification number 12345678. If the employee has more than five occurrences of the education information with a GPA of less than 3.5, no changes are made and an error is returned.

```
MODIFY EMPLOYEE
       (EDUCATION:= EXCLUDE LIMIT=5 EDUCATION WITH (GPA < 3.5))
WHERE EMPLOYEE_ID = 12345678
```

Replacing Attribute Values

The syntax required for changing an attribute value depends on whether the attribute is single-valued, multivalued, entity-valued, or a compound. The following text provides the syntax required to alter values for different types of attributes.

Replacing Values in Single-Valued Attributes

To replace a value for a single-valued DVA or EVA, use the same syntax as that for adding a value:

```
── MODIFY ─┬──────────────────────────────┬─<class>─┬────────────────────┬───→
           └─ LIMIT ── = ─┬── ALL ───────┤          └─ OF ──<database>──┘
                          └─<integer>───┘

             ┌←───────────────── , ──────────────────┐
→─ ( ──┴─<attribute>── := ──<expression1>──┴─) ── WHERE ─────────────────→

→─┬── TRUE ────────┬──────────────────────────────────────────────────┤
  └─<expression2>─┘
```

Replacing Values in Multivalued Attributes

To replace a value for a multivalued attribute, you must first use the EXCLUDE clause to delete the value and then use the INCLUDE clause to add the new value. For more information on the EXCLUDE and INCLUDE clauses, refer to “Deleting Values from Multivalued Attributes” and “Adding Values to Multivalued Attributes” earlier in this section.

Replacing Values in Compound Attributes

Use the following syntax to replace values in compound attributes:

```
── MODIFY ─┬──────────────────────────────┬─<class>── (──<compound>── := ─→
           └─ LIMIT ── = ─┬── ALL ───────┤
                          └─<integer>───┘

→─<EXCLUDE clause>── , ──<compound>── := ──<INCLUDE clause>──) ────────→

→─ WHERE ─┬── TRUE ───────┬──────────────────────────────────────────────┤
          └─<expression>─┘
```

<EXCLUDE clause>

```
── EXCLUDE ──<compound>── WITH ── (──<component expression>──) ───────┤
```

<INCLUDE clause>

```
── INCLUDE ── (──────────────────────────────────────────────────────────→

  ┌←──────────────────────────── , ─────────────────────────────┐
→─┴─<component>── := ─┬─<value>──────────────────────────────┬─┴─) ─────────┤
                      │  ┌←─────────────── , ──────────────┐ │
                      └──┴─<component>── := ──<value>──┴──┘
```

The following query shows how to change values for a compound attribute. In the example, the compound attribute EDUCATION is altered by replacing information relating to a Master of Science (MS) degree with information relating to a Master of Arts (MA) degree.

```
MODIFY EMPLOYEE
       (EDUCATION:= EXCLUDE EDUCATION WITH (DEGREE_OBTAINED = MS),
        EDUCATION:= INCLUDE (DEGREE_OBTAINED:= MA,
                              YEAR_OBTAINED:= 1/1/89,
                              GPA:= 3.7)
       )
WHERE EMPLOYEE_ID = 12345678
```

The WITH clause in the query can refer to any element in the compound. If the compound attribute has a nested compound attribute, the values for the nested compound must be grouped together in parentheses. Assume that in the preceding example, GPA is a compound attribute that contains the GPA at the end of each year. The update query would look as follows:

```
MODIFY EMPLOYEE
       (EDUCATION:= EXCLUDE EDUCATION WITH (DEGREE_OBTAINED = MS),
        EDUCATION:= INCLUDE (DEGREE_OBTAINED:= MA,
                              YEAR_OBTAINED:= 1/1/89,
                              GPA:=
                               (YEAR_1:= 3.7,
                                YEAR_2:= 3.6
                               )
                             )
       )
WHERE EMPLOYEE_ID = 12345678
```

In order to change values in multivalued attributes, you must first exclude the old values and then include the new values.

When you assign values to a new multivalued compound attribute, SIM assigns a null value to any components not explicitly assigned values. If the multivalued compound attribute is a required attribute, you must assign a nonnull value to at least one of the attribute components. If you use the EXISTS operator on a compound attribute, the query returns a true value if at least one of the components is assigned a value. For more information, refer to "Using the EXISTS Operator" in Section 6, "Writing Expressions for Retrieval and Update Queries."

When you modify a compound attribute that is single-valued, if you use the INCLUDE keyword, components that you do not assign new values become null. If you do not use the INCLUDE keyword, components retain their old values if you do not assign new values to them.

Associating Entities by Using Entity-Valued Attributes (EVAs)

Use the following syntax to replace or add a single-valued EVA value:

```
── MODIFY ──<class>─┬─────────────────────┬─ ( ──<EVA>── := ─────────────→
                    └─ OF ──<database>─┘

→─<class pointed to>── WITH ── ( ──<expression1>── ) ── ) ── WHERE ───→

→─<expression2>───────────────────────────────────────────────────────┤
```

To delete a value from a single-valued EVA, use the following syntax:

```
── MODIFY ──<class>─┬─────────────────────┬─ ( ──<EVA>── := ── EXCLUDE ───→
                    └─ OF ──<database>─┘

→─<EVA>─┬──────────────────────────────────┬─ ) ── WHERE ──<expression>─┤
        └─ ( ── WITH ──<expression1>── ) ─┘
```

Use the INCLUDE and EXCLUDE clauses to add or remove an association between two or more entities. In addition, include an expression in the query to designate the entities that are to be affected.

Example 1

The following query associates the manager Anne Bartlett with the employee Jerry Munks. Because the EVA *EMPLOYEE_MANAGER* is single-valued, the keyword INCLUDE is not required.

```
MODIFY EMPLOYEE
       (EMPLOYEE_MANAGER:=   MANAGER
                              WITH (LAST_NAME OF NAME = "Bartlett"
                              AND FIRST_NAME OF NAME = "Anne")
       )
WHERE LAST_NAME OF NAME = "Munks"
      AND FIRST_NAME OF NAME = "Jerry"
```

Example 2

The following query associates the employee Jerry Munks with the manager Anne Bartlett. Because the EVA *EMPLOYEES_MANAGING* is multivalued, the keyword INCLUDE is required.

```
MODIFY MANAGER
       (EMPLOYEES_MANAGING:= INCLUDE EMPLOYEE
                              WITH (LAST_NAME OF NAME = "Munks"
                              AND FIRST_NAME OF NAME = "Jerry")
       )
WHERE LAST_NAME OF NAME = "Bartlett"
      AND FIRST_NAME OF NAME = "Anne"
```

Example 3

The following query removes the association between the employee Jerry Munks and the manager Anne Bartlett:

```
MODIFY MANAGER
       (EMPLOYEES_MANAGING:= EXCLUDE EMPLOYEES_MANAGING
                              WITH (LAST_NAME OF NAME = "Munks"
                              AND FIRST_NAME OF NAME = "Jerry")
       )
WHERE (LAST_NAME OF NAME = "Bartlett"
      AND FIRST_NAME OF NAME = "Anne")
```

Example 4

The following query removes the association between the manager Anne Bartlett and the employee Jerry Munks. Note that because the EVA *EMPLOYEE_MANAGER* is single-valued, the keyword EXCLUDE is required, but not the keyword INCLUDE.

```
MODIFY EMPLOYEE
       (EMPLOYEE_MANAGER:= EXCLUDE
       )
WHERE (LAST_NAME OF NAME = "Munks"
      AND FIRST_NAME OF NAME = "Jerry")
```

Note: *If you modify an EVA, the reverse modification is automatically made. For example, if you assign a manager to a project employee, using the EVA* EMPLOYEE_MANAGER, *the information is automatically assigned to the EVA* EMPLOYEES_MANAGING *because EMPLOYEE_MANAGER is declared as the inverse of EMPLOYEES_MANAGING in the Object Definition Language (ODL) for the database.*

Replacing Class Attributes

Use the MODIFY statement without a WHERE clause to update class attributes. The perspective class for the update query is the class (or subclass of the class) that owns the attributes being altered. You cannot use the same MODIFY statement to alter both class and entity attributes.

The syntax for replacing class attributes is shown in the following diagram:

```
── MODIFY ─┬──────────────────────────┬─<class>─┬───────────────────┬───→
           └─ LIMIT ── = ─┬─ ALL ─────┤         └─ OF ──<database>──┘
                          └─<integer>─┘

         ┌←─────────────────── , ────────────────────┐
→─ ( ──┴─<attribute>── := ─┬───────────┬─┬──────────────────┬┴─ ) ──────┤
                           ├─ EXCLUDE ─┤ └─<expression1>────┘
                           └─ INCLUDE ─┘
```

Adding Entities and Roles

You can add a role to an entity by inserting an entity from a class into a subclass in the same hierarchy. For example, you could insert an entity from the EMPLOYEE class to the PROJECT_EMPLOYEE class.

You can add an entity to the database by adding an entity that does not already exist in the database.

The syntax for assigning new values to the attributes is the same as that for the MODIFY statements with the exception that the EXCLUDE statement cannot include a WITH expression.

Adding an entity to a subclass automatically adds the entity to the superclasses of that class. The values for the attributes of type subrole are generated automatically. However, adding an entity to a superclass does not add information to the subclasses of that class.

Adding an Entity from Another Class

The syntax for inserting an entity from a class to one of its subclasses is as follows:

```
—— INSERT ——<subclass>—— FROM ——<superclass>—— WHERE ——————————————————→

→—<expression defining entity to be transferred>—— ( ——————————————————→

→—<attributes being assigned values>—— ) ——————————————————————————————┤
```

For example, if you want to promote John Lemon to a manager and assign him a new title and some employees, you could use any one of the following three queries:

Example 1

```
INSERT MANAGER FROM EMPLOYEE
WHERE LAST_NAME OF NAME = "Lemon" AND FIRST_NAME OF NAME = "John"
(MANAGER_TITLE:= Supervisor,
  EMPLOYEES_MANAGING:=
                    INCLUDE EMPLOYEE WITH (LAST_NAME OF NAME = "Franks"
                                     AND FIRST_NAME OF NAME = "Joshua"),
  EMPLOYEES_MANAGING:=
                    INCLUDE EMPLOYEE WITH (LAST_NAME OF NAME = "Truman"
                                     AND FIRST_NAME OF NAME = "Liza"),
  EMPLOYEES_MANAGING:=
                    INCLUDE EMPLOYEE WITH (LAST_NAME OF NAME = "Singer"
                                      AND (FIRST_NAME OF NAME =
                                      "Stephanie")
)
```

Example 2

```
INSERT MANAGER FROM EMPLOYEE
WHERE LAST_NAME OF NAME = "Lemon" AND FIRST_NAME OF NAME = "John"
(MANAGER_TITLE:= Supervisor,
  EMPLOYEES_MANAGING:= INCLUDE EMPLOYEE WITH
                    (
                     (LAST_NAME OF NAME = "Franks"
                     AND FIRST_NAME OF NAME = "Joshua")
                     OR
                     (LAST_NAME OF NAME = "Truman"
                     AND FIRST_NAME OF NAME = "Liza")
                     OR
                     (LAST_NAME OF NAME = "Singer"
                     AND FIRST_NAME OF NAME = "Stephanie")
                    )
)
```

Example 3

```
INSERT MANAGER FROM EMPLOYEE
WHERE LAST_NAME OF NAME = "Lemon" AND FIRST_NAME OF NAME = "John"
(MANAGER_TITLE:= Supervisor,
  EMPLOYEES_MANAGING:= INCLUDE
                        (EMPLOYEE WITH
                          (LAST_NAME OF NAME = "Franks" AND
                           FIRST_NAME OF NAME = "Joshua"),
                         EMPLOYEE WITH
                          (LAST_NAME OF NAME = "Truman" AND
                           FIRST_NAME OF NAME = "Liza"),
                         EMPLOYEE WITH
                          (LAST_NAME OF NAME = "Singer" AND
                           FIRST_NAME OF NAME = "Stephanie")
                        )
)
```

You can supply values only for the immediate attributes of the entity. If you assign subroles to the entity you create, you can also assign values to the immediate attributes that describe the entity in the designated subrole.

If you want to change any of the extended attributes of the entity, you must write a separate modification statement.

The immediate attributes not assigned values are assigned initial or null values. Required attributes must be assigned values; therefore, in the preceding examples, MANAGER_TITLE is assigned a value because it is a required attribute of the MANAGER_CLASS.

If two subclasses have a common superclass that has a multivalued attribute of type subrole, you can add a new role to an entity so that its information can be accessed from both of the subclasses and from the superclass. The two subclasses need not have a superclass-subclass relationship. For example, because the PROJECT_EMPLOYEE and MANAGER classes have the same superclass—EMPLOYEE, you can add the role of MANAGER to a PROJECT_EMPLOYEE entity, and you can add the role of PROJECT_EMPLOYEE to a MANAGER entity. If you attempt to add a role to an entity when the common superclass only has a single-valued subrole attribute, the update fails and the system returns an error. You cannot, however, add the role of ASSIGNMENT to a PROJECT entity or the role of PROJECT to an ASSIGNMENT entity, because the PROJECT and ASSIGNMENT classes are in different hierarchical structures; that is, the PROJECT and ASSIGNMENT classes do not share a common superclass.

In a hierarchy, the inserted superclass can be many levels above the inserted subclass. If the entity being added does not already exist in the levels between the inserted subclass and the inserted superclass, SIM creates the entity as part of the INSERT statement.

In the ASSIGNMENT statements, you can refer to attributes in the created entities. If there is a required attribute for the created entity, the INSERT statement must assign a value to the required attribute. The INSERT statement cannot refer to attributes in the inserted superclass or in any class between the inserted superclass and the inserted subclass where the entity exists.

Note: *To insert an entity from a class into a subclass with multiple superclasses, all the superclass entities of the inserted subclass must already exist. If one or more of the superclasses do not exist, the message "Super Class entity missing <superclass name>" displays.*

Except when creating a new entity and altering class attributes, you must always include a WHERE clause in an update query. When you insert an entity from one class into another, the WHERE clause must be used only to select a single entity.

Adding a New Entity to the Database

The syntax for inserting an entity into the database is as follows:

```
── INSERT ──<class>──┬───────────────────┬── ( ──<assignment statements>── ) ─┤
                     └─ OF ──<database>──┘
```

For example, to add a new project to the database, you can use the following query:

```
  INSERT PROJECT
         (PROJECT_NO:= 12345,
          PROJECT_TITLE:= "Jeronimo",
          PROJECT_TEAM:= INCLUDE
                         (PROJECT_EMPLOYEE WITH
                          (LAST_NAME OF NAME = "Colt"),
                          PROJECT_EMPLOYEE WITH
                          (LAST_NAME OF NAME = "Dewey"),
                          PROJECT_EMPLOYEE WITH
                          (LAST_NAME OF NAME = "Old"),
                          PROJECT_EMPLOYEE WITH
                          (LAST_NAME OF NAME = "Dust")
                         ),
          DEPT_ASSIGNED:= DEPARTMENT WITH (DEPT_NO = 34)
         )
```

You cannot include a WHERE clause when creating a new entity.

Deleting Entities

Use the deletion statement to remove entities from a class. The deletion statement removes the designated entities from the requested class and from any subclasses of the requested class. The deletion statement does not remove the entity from any superclasses. For example, if you remove an entity as an employee, the entity still exists as a person, but no longer exists as a manager or project employee. But if you remove an entity as a person, the entity is removed from all classes in the database.

Include a LIMIT clause in the deletion statement to indicate the maximum number of entities you want to delete. If you exceed the limit, the deletion process ends without any changes being made to the database. The default value for the LIMIT clause is 1.

Note: *You cannot delete an entity if it is the target of the only occurrence of a required EVA. For example, if a project is assigned as the only current project of a project employee, that project cannot be deleted because the requirement that each project employee must be assigned to at least one project would no longer be fulfilled.*

The syntax for the deletion statement is as follows:

```
── DELETE ─┬───────────────────────────┬─<class>─┬──────────────────┬────→
           └─ LIMIT ── = ──<integer>──┘          └─ OF ──<database>─┘

→─ WHERE ──<expression defining entities to be deleted>────────────────┤
```

The WHERE clause must be present in the deletion statement.

Example

To remove the projects Fort Field and North Field from the database, you could use the following query:

```
DELETE LIMIT=2 PROJECT
WHERE PROJECT_TITLE = "Fort Field"
      OR
      PROJECT_TITLE = "North Field"
```

Section 6
Writing Expressions for Retrieval and Update Queries

In a query, you can include three categories of expressions:

- Target attribute
- Local filter
- Global selection

In Section 3, “Structuring Retrieval Queries,” the purpose of each expression is discussed. This section provides information on the operators and functions you can use to construct expressions.

Applying General Rules for Expression Writing

When you form an expression, you should keep the following factors in mind:

- Attributes can be assigned null values. Null values produce well-defined but not necessarily obvious results in expressions. Use the EXISTS operator to check for null values. Refer to “Using the EXISTS Operator” in this section for more information.
- In SIM databases, a Boolean item can take one of three values: true (T), false (F), or null (?). Refer to “Using Boolean Operators” in this section for more information.
- Expressions containing compound attributes can be tested to ensure that the individual components of the compound attribute are assigned nonnull values. A test carried out only on the compound attribute ensures that at least one component of the attribute is assigned a value. For example, if you test the attribute NAME for the existence of values, a true value is returned if you assign a value to FIRST_NAME, LAST_NAME, or MID_INITIAL.

  Use the EXISTS operator to check for null values. Refer to “Using the EXISTS Operator” in this section for more information.
- Expressions containing multivalued attributes are usually evaluated to true if at least one occurrence meets the criterion in the expression.

  However, if the condition containing the multivalued attribute is logically combined with another condition, the result is affected by the binding of names rule. For more information on binding of names, refer to “Understanding the Binding of Names Rules” in Section 2, “OML Language Conventions.”

You can use the quantifiers ALL, SOME, and NO to state explicitly the conditions you want to impose on multivalued attributes. For more information, refer to “Using Quantifiers” in this section.

Using Relational Operators

Use relational operators to compare the values of two attributes. Attribute values can be compared either with other attribute values or with fixed values.

Relational operators are computed from left to right, except for the ISA operator, which is given a higher precedence. To attach a precedence to a portion of the expression, use parentheses.

Two types of relational operators are supported: scalar and role testing.

Using Scalar Relational Operators

OML supports two types of scalar relational operators:

- Logical
- Equivalence

Table 6–1 describes the logical scalar relational operators supported by OML. As shown in the table, you can express each operator, using either a mnemonic or a symbol.

Table 6–1. Logical Scalar Relational Operators

Operator Mnemonic	Operator Symbol	Example	Explanation
LSS	<	a LSS b	Returns a true value if a is less than b
LEQ	<=	a LEQ b	Returns a true value if a is less than or equal to b
EQL	=	a EQL b	Returns a true value if a is equal to b
NEQ	<>	a NEQ b	Returns a true value if a is not equal to b
GTR	>	a GTR b	Returns a true value if a is greater than b
GEQ	>=	a GEQ b	Returns a true value if a is greater than or equal to b

Table 6–2 describes the equivalence scalar relational operators supported by OML. The equivalence scalar relational operators base their comparisons on the ordering sequence of the data. As shown in the table, you can express each operator by using a mnemonic.

You can use the equivalence scalar relational operators only with string data. In addition, the ccsversions of the data being compared must be the same. A syntax error is returned if you try to use the equivalence scalar relational operators with data of any type other than string or if the ccsversions of the data being compared do not match.

Table 6–2. Equivalence Scalar Relational Operators

Operator Mnemonic	Example	Explanation
EQV-LSS	a EQV-LSS b	Returns a true value if a is less than b
EQV-LEQ	a EQV-LEQ b	Returns a true value if a is less than or equal to b
EQV-EQL	a EQV-EQL b	Returns a true value if a is equal to b
EQV-NEQ	a EQV-NEQ b	Returns a true value if a is not equal to b
EQV-GTR	a EQV-GTR b	Returns a true value if a is greater than b
EQV-GEQ	a EQV-GEQ b	Returns a true value if a is greater than or equal to b

Relational operators can be used with other kinds of operators—for example, arithmetic operators.

If an attribute in the expression is multivalued, use a quantifier to select whether all, some, or none of the values must meet the criterion. Quantifiers are discussed under "Using Quantifiers" later in this section.

Example 1

In the following example, the query returns a true value if a project employee is assigned to the Engineering department:

```
DEPT_TITLE OF DEPT_IN OF PROJECT_EMPLOYEE = "Engineering"
```

Example 2

In the following example, the query returns a true value if the salary of an employee is greater than or equal to 1.25 times the salary of the employee's manager:

```
EMPLOYEE_SALARY OF EMPLOYEE >=
  (EMPLOYEE_SALARY OF EMPLOYEE_MANAGER OF EMPLOYEE * 1.25)
```

Note: *Refer to Table 6–3 to see the effect of null values on the result of an expression.*

Using Role Testing Relational Operators

The ISA operator enables you to test whether an entity has more than one role in the database. For example, use the ISA operator to verify whether a project employee is also a manager. The syntax for the ISA operator is as follows:

```
—<class1 or EVA>— ISA —<class2>————————————————————————————————┤
```

Example

The following example returns the employee identification number of a project employee if he or she is also a manager:

```
FROM PROJECT_EMPLOYEE
RETRIEVE EMPLOYEE_ID
WHERE PROJECT_EMPLOYEE ISA MANAGER
```

The ISA operator has a higher precedence than does any of the Boolean operators.

You can also test for roles, using attributes of type subrole. Any class that has one or more subclasses has an attribute of type subrole. In the ORGANIZATION database, examples of attributes of type subrole include EMPLOYED in the PERSON class and PROFESSION in the EMPLOYEE class. Following is an example of role testing using an attribute of type subrole:

```
FROM PROJECT_EMPLOYEE
RETRIEVE NAME, EMPLOYEE_SALARY, NAME OF SPOUSE
WHERE EMPLOYED OF PROJECT_EMPLOYEE = EMPLOYED OF SPOUSE
   OF PROJECT_EMPLOYEE
```

Note: *In a retrieval query, you can also use the AS clause to return data about an entity in a role not usually visible from the perspective class. For more information, refer to “Using Subroles” in Section 2, “OML Language Conventions.”*

Using the EXISTS Operator

Use the EXISTS operator to ensure that SIM evaluates only occurrences of expression arguments that have values other than null. Add the keyword EXISTS after the item that you want to test. You can use the EXISTS operator with both DVAs and EVAs.

If you use the EXISTS operator on a compound attribute, the query returns a true value if at least one of the components is assigned a nonnull value.

The EXISTS operator takes a higher precedence than that of the Boolean operators.

Example

The following query returns the personal identification of a manager only if that manager is assigned employees and projects with titles:

```
FROM MANAGER
RETRIEVE PERSON_ID
WHERE EMPLOYEES_MANAGING EXISTS
      AND PROJECT_TITLE OF PROJECTS_MANAGING EXISTS
```

Using Boolean Operators

Three Boolean operators are available: NOT, AND, and OR. Use these operators to link statements in an expression and thereby set multiple conditions in one expression.

All the relational operators take precedence over the Boolean operators. The precedence of the Boolean operators themselves is NOT, AND, OR, where NOT is given the highest precedence and OR is given the lowest precedence. To alter the evaluation order, use parentheses.

Example 1

The following query gives each employee a 10 percent pay raise if the employee’s salary is in the range of $10,000 through $20,000 or if the salary of the employee’s manager is greater than $35,000:

```
MODIFY LIMIT = ALL EMPLOYEE
       (EMPLOYEE_SALARY:= (EMPLOYEE_SALARY * 1.10))
WHERE EMPLOYEE_SALARY >= 10000
      AND EMPLOYEE_SALARY <= 20000
      OR EMPLOYEE_SALARY OF EMPLOYEE_MANAGER > 35000
```

Example 2

The following query returns the salary and personal identification number of an employee if his or her salary fits the following criteria:

- His or her salary is greater than or equal to $10,000.
- His or her salary is less than or equal to $20,000, or the salary of the employee's manager is greater than $35,000.

```
FROM EMPLOYEE
RETRIEVE EMPLOYEE_SALARY, PERSON_ID
WHERE EMPLOYEE_SALARY >= 10000
      AND (EMPLOYEE_SALARY <= 20000
           OR EMPLOYEE_SALARY OF EMPLOYEE_MANAGER > 35000
          )
```

Identifying the Effect of the SIM Logic System

A Boolean statement can produce one of three results: true (T), false (F), or null (?). If an attribute has not been assigned a value for a particular entity, the statement containing the null value is evaluated to a null value. When an expression contains several statements linked by Boolean operators and one or more of the statements are null, the expression is not necessarily null. Table 6–3 explains the results of comparing true, false, and null values.

For example, the first line of the table shows that if *a* and *b* are both true, the *a AND b* operation returns a true value; the *a OR b* operation returns a true value; and the *NOT a* operation returns a false value.

Table 6–3. Logic System for Boolean Operators in a SIM Database

a	b	a AND b	a OR b	NOT a
T	T	T	T	F
T	F	F	T	F
T	?	?	T	F
F	T	F	T	T
F	F	F	F	T
F	?	F	?	T
?	T	?	T	?
?	F	F	?	?
?	?	?	?	?

Use the EXISTS operator to verify that a value exists. For more information, refer to "Using the EXISTS Operator" earlier in this section.

Returning a Numeric Value from a Boolean Expression

You can convert the result of a Boolean expression to a numeric value by using the REAL operator. The results of using the REAL operator on a Boolean expression are as follows:

Boolean Expression Result	REAL Operator Result
True	1
False	0
Null	Null

You must follow the REAL operator with a Boolean expression in parentheses. The Boolean expression can be simple or complex.

Example

If a project has more than 10 subprojects, the following query returns the name of the project and the number of additional subprojects. However, if the project has 10 or fewer subprojects, the query returns the name of the project and the value 0 (zero).

```
FROM PROJECT
RETRIEVE PROJECT_TITLE,
         (REAL((COUNT(TRANSITIVE(SUB_PROJECTS))) > 10) *
         (COUNT(TRANSITIVE(SUB_PROJECTS)) - 10))
```

Using Quantifiers

You use quantifiers with multivalued attributes to designate whether all, some, or no occurrences of an attribute must fit a condition before a true result can be returned.

The three supported quantifiers—ALL, SOME, and NO—are discussed in Table 6–4.

Table 6–4. Quantifier Descriptions

Quantifier	Explanation
ALL	Returns a true value if all occurrences of the attribute meet the condition or if there are no occurrences of the attribute
SOME	Returns a true value if at least one occurrence of the attribute meets the condition
NO	Returns a true value if none of the occurrences of the attribute meet the condition, or if there are no occurrences of the attribute

A multivalued expression in parentheses must immediately follow the quantifier keyword.

When SIM evaluates an entity to see if it fits a condition given by a global selection expression or a local filter expression, the entity is evaluated once for each occurrence of a multivalued attribute. Each time you use a quantifier in an expression, a new reference variable is generated. For example, the following query can never return a true value if the SOME quantifier is not used, because no employee would fit the global selection expression since only one value of the PROJECT_TITLE attribute would be examined each time:

```
FROM MANAGER
RETRIEVE NAME OF EMPLOYEES_MANAGING
WHERE PROJECT_TITLE OF PROJECTS_MANAGING = "Camelot"
      AND SOME(PROJECT_TITLE OF PROJECTS_MANAGING) = "Lancelot"
```

Since a new reference variable is generated for the path expression associated with the SOME operator, the two tests involving PROJECT_TITLE in the WHERE clause need not refer to the same entity.

You can use more than one quantifier in an extended qualification, as the following example shows:

```
FROM MANAGER
RETRIEVE NAME, MANAGER_TITLE
WHERE TITLE OF SOME(PROJECT_TEAM) OF ALL(PROJECTS_MANAGING)= Staff
```

The condition is met by any manager for whom at least one employee of each project has the title of *Staff*.

Understanding Quantifier Use Restrictions

The following restrictions apply to the use of quantifiers:

- You cannot compare quantified expressions.
- You cannot quantify a reference variable that is constructed by using the CALLED keyword.

Illustrating the Effect of Quantifiers on Multivalued Attributes

The effect of using a quantifier with a multivalued attribute is not always obvious. The following example WHERE clauses illustrate the use of quantifiers to ensure that:

- At least one value fits a condition.
- At least one value does not fit a condition.
- No values fit a condition.

Example 1: Returning a True Value If At Least One Value Fits a Condition

Both of the following WHERE clauses return a true result if at least one of a manager's employees is 25 years old:

```
WHERE AGE OF SOME(EMPLOYEES_MANAGING) = 25

WHERE AGE OF EMPLOYEES_MANAGING = 25
```

While both of these WHERE clauses produce a true result, the number of entities returned by the queries using these clauses might differ.

Example 2: Returning a True Value If At Least One Value Does Not Fit a Condition

Both of the following WHERE clauses return a true result if at least one of a manager's employees is not 25 years old:

```
WHERE NOT AGE OF EMPLOYEES_MANAGING = 25

WHERE AGE OF EMPLOYEES_MANAGING <> 25
```

Example 3: Returning a True Value If No Value Fits a Condition

Both of the following WHERE clauses return a true value if none of a manager's employees is 25 years old:

```
WHERE NOT AGE OF SOME(EMPLOYEES_MANAGING) = 25

WHERE AGE OF NO(EMPLOYEES_MANAGING) = 25
```

Using Arithmetic Operations

You can use arithmetic operations in the following places:

- The target list
- A relational expression, as an operand
- An UPDATE or MODIFY statement as the right-hand side of the assignment, that is, following the := symbol

You can use only one multivalued attribute in any single arithmetic operation. You can construct the operation using constants, built-in arithmetic and aggregate functions, and attributes.

Using Arithmetic Constants

As well as the usual representation of integers and real numbers, constants can be represented in hexadecimal notation. The hexadecimal value must be enclosed in single quotation marks and be preceded by the digit 4. For example, the hexadecimal representation of the decimal number 46 is 4'2E'.

Using Arithmetic Operators

Table 6–5 describes the arithmetic operators supported in OML.

Table 6–5. Arithmetic Operator Descriptions

Operator	Example	Explanation
*	a * b	Multiply *a* by *b*.
/	a / b	Divide *a* by *b*.
+	a + b	Add *a* to *b*.
–	a – b	Subtract *b* from *a*.
DIV	a DIV b	Divide *a* by *b* and give only the integer portion of the result. For example, 23 DIV 5 returns the value 4.
MOD	a MOD b	Divide *a* by *b* and give only the remainder portion of the result. For example, 23 MOD 5 returns the value 3.
**	a**b	Raise *a* to the power of *b*. For example, 3**2 returns the value 9.

The order of precedence associated with these operators is the following:

```
**     (*   /   DIV   MOD)   (+   -)
Highest---------------------> Lowest precedence
```

The operators given in parentheses have the same precedence; in an expression, SIM deals with these operators from left to right. The exception to this evaluation order is the process for evaluating exponents. Groups of exponents are evaluated from right to left. The following two statements are equivalent:

```
2**3**4**5

2**(3**(4**5))
```

To alter the order of precedence of operators, use parentheses.

You can also retrieve the negative value of an attribute. For example, RETRIEVE –EMPLOYEE_SALARY returns values such as –30,000 and –23,567. In an expression, if you request the negative value (monadic –) of an attribute, then that request is given the highest priority and is evaluated first. For example, –3**2 evaluates to 9 whereas 10 – 3**2 evaluates to 1.

If either operand of an arithmetic operator is null, the result is null.

Example

To find the salary cost of a project compared to the person hours spent on a project, use the following expression:

```
FROM PROJECT
RETRIEVE ((EMPLOYEE_SALARY OF PROJECT_MANAGER
           + SUM(EMPLOYEE_SALARY OF PROJECT_TEAM)
          )
          / EST_PERSON_HOURS OF ASSIGNMENT_HISTORY
         )
```

Note: *The use of the SUM operator is discussed under “Using Aggregate Functions” later in this section.*

Using Arithmetic Functions

Table 6–6 describes the arithmetic functions supported in OML.

Table 6–6. Arithmetic Function Descriptions

Function	Example	Explanation
ABS(a)	ABS(4) = 4 ABS(–5) = 5	Returns the positive value of *a*.
ROUND(a)	ROUND(3.2) = 3 ROUND(–6.7) = –7 ROUND(2.5) = 3	Rounds *a* to the nearest integer. Values ending in .5 are always rounded up.
SQRT(a)	SQRT(121) = 11	Returns the square root of *a*. This operator is valid only when *a* is a positive number.
TRUNC(a)	TRUNC(2.35) = 2 TRUNC(–4.51) = –4	Returns the integer portion of *a*.

Using Aggregate Functions

Use aggregate functions on collections of values designated by path expressions. You cannot use the DISTINCT keyword with an aggregate function. Refer to “Prevention of Duplicate Output” in Section 4, “Formatting the Output from Retrieval Queries,” for more information on the purpose of the DISTINCT keyword.

Table 6–7 describes the aggregate functions supported by OML.

Table 6–7. Aggregate Function Descriptions

Function	Explanation
AVG (<attribute>)	Returns the average value of the attribute. You need not use the EXISTS operator with this function since null values are automatically ignored.
COUNT (<attribute>)	Returns the number of nonnull values of the attribute. The attribute can be a class name, in which case a count of the entities in the class is made. You can use DVAs and EVAs in the expression.
MAX (<attribute>)	Determines the maximum value of the attribute. Null values are ignored. This function can be applied to any attribute using attributes whose underlying type is ordered.
MIN (<attribute>)	Determines the minimum value of the attribute. Null values are ignored. You can apply this function to any expression that uses attributes whose underlying type is ordered.
SUM (<attribute>)	Returns the total value of the attribute. Null values are ignored.

You must enclose in parentheses the expression to which the aggregate function is applied. The result of the function is treated as a single-valued attribute belonging to the class that owns the rightmost attribute in the expression. When you use an aggregate function, you should ensure that the expression is multivalued with respect to the owning class. If the expression is not multivalued, a value of 1 or 0 (zero) is returned. For example, the following query contains an aggregate function that is applied to a single-valued expression:

```
FROM PROJECT
RETRIEVE PROJECT_NO, COUNT(PROJECT_MANAGER)
```

The following query contains an aggregate function that is applied to a multivalued expression:

```
FROM MANAGER
RETRIEVE NAME, COUNT(PROJECTS_MANAGING)
```

Table 6–8 illustrates the use of the COUNT aggregate function.

Table 6–8. Examples of the Use of the COUNT Aggregate Function

Example	Explanation
RETRIEVE COUNT(PROJECT_TEAM) OF PROJECT	For each project, returns the number of employees assigned to that project. COUNT(PROJECT_TEAM) is treated as an immediate attribute of the PROJECT class.
RETRIEVE COUNT(PROJECT_TEAM OF PROJECTS_MANAGING) OF MANAGER	For each manager, returns the number of project employees assigned to all projects for which that manager is responsible. COUNT(PROJECT_TEAM OF PROJECTS_MANAGING) is treated as an immediate attribute of the MANAGER class.
RETRIEVE COUNT(PROJECT_TEAM OF PROJECTS_MANAGING OF DEPT_MANAGERS) OF DEPARTMENT	For each department, returns the number of project employees assigned to all projects for which a manager in that department is responsible. COUNT(PROJECT_TEAM OF PROJECTS_MANAGING OF DEPT_MANAGERS) is treated as an immediate attribute of the DEPARTMENT class.

If the argument includes the perspective class, the result is a single value for the entire database.

Table 6–9 illustrates the use of aggregate functions with arguments that include the perspective class.

Table 6–9. Examples of the Use of Aggregate Functions with the Perspective Class

Example	Explanation
RETRIEVE COUNT(EMPLOYEE)	Returns the total number of employees.
RETRIEVE AVG(EMPLOYEE_SALARY OF MANAGER)	Returns the average salary of all managers.
RETRIEVE MAX(AGE OF PROJECT_EMPLOYEE)	Returns the age of the oldest project employee.

Accessing EVA Information

Table 6–10 explains the two functions available for accessing EVA information.

Table 6–10. INVERSE and TRANSITIVE Functions Descriptions

Function	Explanation
INVERSE	Accesses information when an EVA exists to link CLASSa to CLASSb, but when no equivalent EVA exists to link CLASSb to CLASSa. You can choose to use the INVERSE function even if an inverse EVA is declared.
TRANSITIVE	Accesses information using reflexive EVAs, that is, attributes that link back to the class to which they themselves belong. For example, a subproject is also a project; therefore, if you select the PROJECT class and the EVA SUB_PROJECTS, you can select attributes from the PROJECT class for a subproject. In some cases, you can link back to the same class several times. For example, you can link back to the MANAGER class repeatedly, using the EVA EMPLOYEES_MANAGER to retrieve information about an employee's supervisor, department manager, and divisional manager. You could also use the TRANSITIVE function to return all project numbers of all subprojects assigned to a manager. To limit the number of times the loop is repeated, use the END operator. If you do not use the END operator, you might create an infinite loop in a query. For example, if you select the PERSON class and the EVA SPOUSE, since a spouse by definition has a spouse, your query would never end unless you use the END operator to limit the number of iterations. Any attribute associated with the TRANSITIVE operation is not bound to any other reference to that attribute.

Using the INVERSE Function

The EVA *DEPT_ASSIGNED* links the PROJECT class to the DEPARTMENT class. To access information about a project when the perspective class is DEPARTMENT, use the INVERSE function, as shown in the following example:

```
FROM DEPARTMENT
RETRIEVE PROJECT_TITLE OF INVERSE(DEPT_ASSIGNED)
WHERE DEPT_TITLE = "Engineering"
```

Using the TRANSITIVE Function

The following statement retrieves name information only for the first two levels of management, for example, the names of supervisors and department managers:

```
FROM MANAGER
RETRIEVE NAME,
         NAME OF TRANSITIVE (EMPLOYEE_MANAGER END LEVEL = 2)
```

If the END clause is omitted from the expression, all the information for all levels of management is retrieved.

The subject of the TRANSITIVE function must be an EVA or a series of EVAs. If the EVA is multivalued, the designated END limit is applied independently to each occurrence of the EVA.

You can use the TRANSITIVE function on path expressions that define a cycle in the database schema, as in the following example:

```
FROM PROJECT
RETRIEVE PROJECT_TITLE,
TRANSITIVE(PROJECTS_MANAGING OF DEPT_MANAGERS OF DEPT_ASSIGNED
           END LEVEL = 2)
```

It is a good practice to include an END LEVEL statement in your query to prevent the occurrence of a never-ending loop. If you are not sure of the number of levels, try setting the END LEVEL to some large value, such as 99.

Using String Operations

You can use string operations to manipulate attributes of type string or character. You can construct string operations, using string constants, a string operator, string functions, and attributes of type string or character.

Using String Constants

You can form a string constant in either of the following two ways:

- By enclosing the string in double quotation marks (")
- By enclosing the hexadecimal representation of the string in double quotation marks (") and preceding the enclosed string with the numeral 4

For example, the following two strings are equivalent:

```
"NORTH FIELD"
4"D5D6D9E3C840C6C9C5D3C4"
```

Table 6–11 details the values for conversion of the A Series EBCDIC character set (ccsversion = ASeriesNative) to a hexadecimal format. The string examples given in this section are based on this mapping. If you are using a different character set, then the results from some of the operations shown in this section might be different.

Table 6–11. EBCDIC-to-Hexadecimal Conversion

Hex	EBCDIC	Hex	EBCDIC	Hex	EBCDIC	Hex	EBCDIC	Hex	EBCDIC
00	NUL	2D	ENQ	6C	%	98	q	D5	N
01	SOH	2E	ACK	6D	_	99	r	D6	O
02	STX	2F	BEL	6E	>	A1	-	D7	P
03	ETX	32	SYN	6F	?	A2	s	D8	Q
05	HT	37	EOT	7A	:	A3	t	D9	R
07	DEL	3C	DC4	7B	#	A4	u	E0	\
0B	VT	3D	NAK	7C	@	A5	v	E2	S
0C	FF	3F	SUB	7D	’	A6	w	E3	T
0D	CR	40	(blank)	7E	=	A7	x	E4	U
0E	SO	4A	[	7F	"	A8	y	E5	V
0F	SI	4B	.	81	a	A9	z	E6	W
10	DLE	4C	<	82	b	C0	{	E7	X
11	DC1	4D	(	83	c	C1	A	E8	Y
12	DC2	4E	+	84	d	C2	B	E9	Z
13	DC3	4F	!	85	e	C3	C	F0	0
15	NL	50	&	86	f	C4	D	F1	1
16	BS	5a	]	87	g	c5	E	F2	2
18	CAN	5B	$	88	h	C6	F	F3	3
19	EM	5C	*	89	i	C7	G	F4	4
1C	FS	5D	)	91	j	C8	H	F5	5
1D	GS	5E	;	92	k	C9	I	F6	6
1E	RS	5F	^	93	l	D0	}	F7	7
1F	US	60-	-	94	m	D1	J	F8	8
25	LF	61	/	95	n	D2	K	F9	9
26	ETB	6A	|	96	o	D3	L		
27	ESC	6B	,	97	p	D4	M		

Using the String Operator

OML supports only one string operator—the concatenation operator—which is represented by an ampersand (&) or the keyword CAT. The result of the operator is that all the characters to the left of the operator are joined with the characters that appear to the right of the operator.

For example, "Car" & "pet" becomes "Carpet".

Using String Functions

The string functions supported by OML are described in Table 6–12.

Table 6–12. String Function Descriptions

Function	Syntax	Explanation

EXTRACT	EXT (string,integer1,integer2)	Returns a substring of the string from the character position indicated by integer1 to the character position indicated by integer2.
		Integer1 can be any positive integer.
		Integer2 must be greater than integer1.
		To denote the end of the string, use an asterisk (*) in place of integer2.
		For example,
		`EXT("Monday",1,3) = "Mon"`
		`EXT("Monday",4,*) = "day"`

LENGTH	LENGTH (string)	Returns the number of characters in the string. For example, `LENGTH("December") = 8` If the string is from a fixed-length string, the value returned is the declared length. For a variable-length string, the value returned is the number of characters assigned to the string.

Table 6–12. String Function Descriptions

Function	Syntax	Explanation

POSITION	POS (string1,string2,integer)	Returns the character position of string1 in string2, at or after the character position indicated by the integer. The value returned is always relative to the position of the first character in string2. If the integer is omitted, the position of the first appearance of string1 is returned.
		For example,
		`POS("e","December") = 2`
		`POS("e","December",3) = 4`
		`POS("e","December",5) = 7`

REPEAT	RPT (string, integer)	Returns the string repeated the number of times indicated by the integer. For example, `RPT("ab",3) = "ababab"`

Applying the Pattern-Matching Operator

OML supports the use of one pattern-matching operator—ISIN. You use the ISIN operator to search for a particular pattern within a string. If SIM finds the pattern it returns a true value; otherwise, it returns a false value. The format for the ISIN operator is

```
<strings to be searched> ISIN <pattern to be looked for>
```

The pattern to be looked for is created with the special characters, or symbols, described in Table 6–13. The special characters and symbols can be used only following the ISIN keyword.

Table 6–13. Symbols Used in Pattern-Matching Operations

Symbol	Example	Explanation

_ (underscore)	The following example returns a true value if the component LAST_NAME includes values such as Lind, Lund, or Lynd:
	`LAST_NAME`
	` ISIN L_nd`
		Matches any single character.

* (asterisk)	The following example returns a true value if the component FIRST_NAME includes values such as Phi, Phil, Phill, Philll, and Phillll:
	`FIRST_NAME`
	` ISIN Phil*`
		Matches zero or more occurrences of the preceding character or group of characters in the statement.

{} (curly braces)	The following example returns a true value if the component FIRST_NAME is, for example, Bill, Jill, or Gill: {xe "{ } (curly braces), using: in pattern-matching operations"}
	`FIRST_NAME`
	` ISIN {BJG}ill`
		Matches any one character given within the braces.

I (vertical bar)	The following example returns a true value if the attribute TITLE has values such as tsar or czar:
	`TITLE`
	` ISIN (ts|cz)ar`
		Matches the pattern on either side of the bar.

( ) (parentheses)	The following example returns a true value if the attribute PROJECT_TITLE includes values such as Lancelot3 and Excalibur2:
	`PROJECT_TITLE`
	` ISIN`
	` (Lancelot|Excalibur)`
	` {123}`
		Groups expressions.

Table 6–13. Symbols Used in Pattern-Matching Operations

Symbol	Example	Explanation

" " (double quotation marks)	The following example returns a true value if the attribute TITLE has a value XX_YY:
	`TITLE`
	` ISIN XX"_"YY`
		Allows pattern-matching symbols in the pattern.

? (question mark)	The following example returns a true value if the component LAST_NAME has values such as Coer, Copper, Corer, Cordoner, or Colder:
	`LAST_NAME`
	` ISIN Co?er`
		Denotes any number (including zero) of characters (shorthand for _*).

... (ellipsis)	The following example returns a true value if the component LAST_NAME has values such as Lund or Lynd, but returns a false value if the component LAST_NAME has the value Lind:
	`LAST_NAME`
	` ISIN L{s..z}nd`
		Denotes an allowable range of characters when used with braces.

Comparing Strings

You can compare string values of ordered attributes by using any of the scalar relational operators. However, the strings must be of the same underlying type. For example, if you attempt to compare strings with different ccsversions, a syntax error is returned.

Two strings are considered equal if the strings are of the same length and match uppercase and lowercase characters. If the first nonmatching character in the first string is preceded by the corresponding character in the second string, the first string has a greater value than the second string does. If the strings are of unequal length, blank characters are added to the end of the shorter string to allow the comparison to be completed.

Table 6–14 illustrates string comparisons that return true values.

Table 6–14. Examples of String Comparisons

Example	Explanation
"abc" < "ABC"	Returns a true value because *a* has a lower value than *A*.
"AZZZZ" < "ZAAAA"	Returns a true value because *A* has a lower value than *Z*.
"One, Two" = "One, Two"	Returns a true value because the strings are of the same length and contain the same uppercase and lowercase letters in the same order.
"1" > "0"	Returns a true value because *1* has a higher value than *0*.
"XYZA" > "XYZ"	Returns a true value because *A* has a higher value than the blank character.

Note: *Strings with a ccsversion that is not ASeriesNative base comparisons on the collating sequence of the data. ASeriesNative strings use the binary sequence as the basis for the comparisons.*

Using Kanji Strings

SIM allows strings to be of type Kanji. All the string operations described in this section can be applied to strings of type Kanji. However, the only relational operators valid when strings of type Kanji are compared are EQL (=), NEQ (<>), EQV-EQL, and EQV-NEQ.

Note: *You cannot mix EBCDIC and Kanji strings in string operations.*

Using Symbolic Expressions

Two symbolic expression operators—PRED and SUCC—are supported by OML. The order of symbolic values is determined by the order in which the values are declared in the database schema. The operators are described in Table 6–15.

Table 6–15. Symbolic Expression Descriptions

Operator	Explanation
PRED	Returns the preceding symbolic value. For example, if the attribute DEGREE has the value BS, PRED(DEGREE) returns the value BA.
SUCC	Returns the succeeding symbolic value. For example, if the attribute DEGREE has the value BS, SUCC(DEGREE) returns the value MA.

Note: *An error is returned if you apply the PRED operator to the minimum value or if you apply the SUCC operator to the maximum value a symbolic can take.*

You can use the symbolic expression operators with attributes that are of type symbolic, for example, DEGREE, LEAVE_STATUS, and FLOOR. When denoting values for symbolic types, do not enclose the value in quotation marks (").

Using Date and Time Expressions

Date and time of day are represented by the special data types date and time, respectively. OML supports a variety of functions that can be used on date and time data types or that can be used on values to produce data of type date or time.

Using Date Expressions

OML supports the following nine date operators:

- CURRENT_DATE
- YEAR
- MONTH
- DAY
- ELAPSED_DAYS
- ADD_DAYS
- DAY_OF_WEEK
- MONTH_NAME
- DATE

Table 6–16 describes the date operators.

Table 6–16. Date Operator Descriptions

Purpose	Operator	Explanation
Determining the current date	CURRENT_DATE	Returns the current date. This operator is a function of type date that has no parameters. If you are using the SIM library interface, you must use the date format MM/DD/YYYY. If you are using IQF to process your queries, you can use the InfoExec SESSION OPTIONS (SESSION) screen to select a format of either MM/DD/YYYY or DD/MM/YYYY. For both of these formats, MM is the month in digits (1 through 12), DD is the day in digits (1 through 31), and YYYY is the year. If the first two digits of the year are 19, you can omit them.
Determining the year	YEAR(<date>)	Returns the year as an attribute of type integer. The following example returns the value 75: `YEAR(12/22/75)`
Determining the month	MONTH(<date>)	Returns the month as an attribute of type integer. The following example returns the value 12: `MONTH(12/22/75)`
Determining the day	DAY(<date>)	Returns the day as an attribute of type integer. The following example returns the value 22: `DAY(12/22/75)`

Determining the number of elapsed days	ELAPSED_DAYS (<date1>, <date2>)	Returns the number of days between date1 and date2.
		
		You can use the ELAPSED_DAYS operator to evaluate both positive and negative day lags; therefore, date1 can be greater than or less than date2.
		
		The following examples return the values 3 and –3 respectively:
		
		`ELAPSED_DAYS(9/29/86, 10/2/86)`
		
		`ELAPSED_DAYS(10/2/86, 9/29/86)`

Table 6–16. Date Operator Descriptions

Purpose	Operator	Explanation

Adding and subtracting days	ADD_DAYS (<date>, <no. of days>)	Adds or subtracts a number of days from a date.
		The following examples return the dates 10/2/86 and 9/29/86 respectively:
		`ADD_DAYS(9/29/86, 3)`
		`ADD_DAYS(10/2/86,-3)`

Determining the day of the week	DAY_OF_WEEK (<date>)	Extracts from an attribute of type date the string literal describing the day. The following example returns the value *Tuesday*: `DAY_OF_WEEK(2/3/87)`
Determining the month name	MONTH_NAME (<date>)	Extracts from an attribute of type date the string literal describing the month. The following example returns the value *February*: `MONTH_NAME(2/3/87)`

Table 6–16. Date Operator Descriptions

Purpose	Operator	Explanation

Converting a numeric value to a date	DATE (<integer>, "<format>")	Returns the integer as a date in the designated format. The integer must be six digits with two digits each for year, month, and day. If you supply less than six digits, the number is padded with leading zeros; for example, the number 21589 is interpreted as 021589. You can designate any of the following formats in which YY and yy denote year, MM and mm denote month, and DD and dd denote day:
		• "YYMMDD"
		• "yymmdd"
		• "MMDDYY"
		• "mmddyy"
		Note that you must enclose the format designator in double quotation marks (").
		
		You can use the result of the DATE operator expression as the argument for any of the other date operators. For example, if HIREDATE is an integer describing the date of hire of an employee in the form YYMMDD, the following expression calculates the number of elapsed days between HIREDATE and October 15th, 1991:
		`ELAPSED_DAYS`
		`(DATE(911015,"YYMMDD"),`
		`DATE(HIREDATE,"YYMMDD"))`

Using Time Expressions

OML supports the following six time operators:

- CURRENT_TIME
- HOUR
- MINUTE
- SECOND
- ELAPSED_TIME
- ADD_TIME

Table 6–17 describes the time operators.

Table 6–17. Time Operator Descriptions

Purpose	Operator	Explanation
Determining the current time	CURRENT_TIME	Returns the current time as a function of type time that has no parameters. The time is returned in the form HH:MM:SS, where HH is the hour on a 24-hour clock, MM is the minutes, and SS is the seconds.
Determining the hours	HOUR(<time>)	Extracts the hour from an attribute of type time. The following example returns the value 10: `HOUR(10:30:59)`
Determining the minutes	MINUTE(<time>)	Extracts the minutes from an attribute of type time. The following example returns the value 30: `MINUTE(10:30:59)`
Determining the seconds	SECOND(<time>)	Extracts the seconds from an attribute of type time. The following example returns the value 59: `SECOND(10:30:59)`

Determining the elapsed time	ELAPSED_TIME (<time1>, <time2>)	Returns the number of seconds between time1 and time2.
		You can use the ELAPSED_TIME operator to evaluate both positive and negative time lags; that is, time1 can be greater than or less than time2.
		The following examples return the values 2 and –2 respectively:
		`ELAPSED_TIME(10:59:59,11:00:01)`
		`ELAPSED_TIME(11:00:01,10:59:59)`

Adding and subtracting seconds from a given time	ADD_TIME (<time>, <seconds>)	Adds or subtracts a number of seconds from a time.
		The following examples return the times 11:00:01 and 10:59:59, respectively:
		`ADD_TIME(10:59:59, 2)`
		`ADD_TIME(11:00:01,-2)`

Section 7
Using the SIM Library Entry Points

This section provides a task-oriented description of the SIM library entry points. This section provides you with the following information:

- A procedure for accessing sample programs on the release media.

  The sample programs illustrate how to use the SIM library entry points in both COBOL74 and ALGOL applications.
- Specific requirements for ALGOL applications that use the SIM library entry points.
- Specific requirements for COBOL and RPG applications that use the SIM library entry points.
- Descriptions of different tasks you might perform using the SIM library entry points.

  These descriptions explain which entry points you use and the order in which to use them to accomplish a particular task.
- An explanation of the structures used by the entry points to pass information between SIM and your application.

  In many cases, you do not need to know the contents of these structures. If you are writing ALGOL applications, the layout of the structures might be important to you. If you are writing COBOL or RPG applications, you might find the information interesting, but the copy deck provided on the release media provides the syntax you need to access the information in the structures.

Section 8, "Describing the SIM Library Entry Points," is a reference section that provides, in alphabetical order, a description of each entry point including the parameters associated with each entry point and program fragments illustrating the use of the entry points.

Accessing Sample Programs

To assist in your understanding of how to use the SIM library entry points in a program, several sample programs are included on the release media. It is recommended that you review these sample programs—they might prove to be a good starting point for your own application.

To copy the programs from the release media, use the Simple Installation (SI) program. The following instructions explain how to copy the sample programs using the SI program in interactive mode. For detailed instructions on using the SI program, refer to the *Simple Installation (SI) Operations Guide.*

1. Select INSTALL on the first screen.

   The INSTALL screen is displayed.

2. Complete the INSTALL screen as follows:

   - Type one of the following in the Software Product Name field:
     - SQ1
     - SQ2
     - IE1
     - IE2
     - IE3
     - DMV
     - LNV
   - Type *X* in the box by the Select Additional File Categories to Copy/Install field.
   - Transmit the screen.

     The FILES screen is displayed.

3. On the FILES screen, perform the following steps:

   - Assuming you have already installed the InfoExec software, remove the X from the All box in the System Files group.
   - Type *X* in the box by the Sample Files field under the heading Generation Data.

4. Complete the installation normally.

Running the SI program in this way copies the following files from the release media:

File Name	Description
EXAMPLE/SIM/ALGOL/API	Illustrates how to set up the parameters and how to call most of the entry points. This program is written in ALGOL.
EXAMPLE/SIM/ALGOL/API/SIMPLE	Reads input from a terminal, accepts a SIM OML string and calls the necessary entry points to parse and execute the query. The retrieved data is displayed on the terminal. This program is written in ALGOL and is simpler than the EXAMPLE/SIM/ALGOL/API program.
EXAMPLE/SIM/ALGOL/API/RECORDATATIME	Describes how to decode the description information so that a record-at-a-time program can execute dynamic queries. This program is the primary documentation for the record-at-a-time retrieval feature. This program is written in ALGOL.
EXAMPLE/SIM/COBOL/API	Provides a COBOL74 version of the EXAMPLE/SIM/ALGOL/API program.
EXAMPLE/SIM/COBOL/COPYDECK	Provides a COBOL74 copy deck that you can include in any COBOL74 program.

Using the SIM Library Entry Points in ALGOL Programs

To use the entry points in programs written in ALGOL and similar languages, you must use an INCLUDE statement to associate the SYMBOL/SIM/PROPERTIES file with your program. For examples illustrating the use of the INCLUDE statement, refer to the sample programs contained on the release media.

Formatting Entry Points

In your program, the format of the statement including the entry point depends upon the programming language you are using. When an entry point is processed, a true or false value is associated with the entry point to indicate whether the attempt to process the entry point was successful.

In ALGOL, FORTRAN, and similar languages, you can extract the true or false value associated with the entry point by making the entry point the right-hand side of a Boolean assignment statement. For example, the following ALGOL program fragment uses the result of an entry point as part of the error checking routine. In the program fragment, the arrays of the form PASS_ARRAY(A) are a shorthand method for denoting the required format for string arrays. For more information on array declarations, refer to “Passing Arrays to SIM from ALGOL Programs” later in this section.

```
IF RSLT:= SIM_PUT_VALUE(PASS_ARRAY(CURSOR_INFO),PASS_ARRAY(ITEM_INFO),
                        PASS_ARRAY(STRING_VALUE),NUMERIC_VALUE)
   THEN EXIT("error putting entity value")
ELSE
   GO GET_PARAMETER;
```

Note: *In the ALGOL program fragment, the entry point has the name SIM_PUT_VALUE, and in the COBOL74 program fragment, the same entry point has the name SCIM_PUT_VALUE.*

For detailed examples illustrating the use of the SIM library entry points, refer to the sample programs contained on the release media.

Passing Arrays to SIM from ALGOL Programs

In ALGOL, all array parameters passed to the SIM library entry points must be followed by an offset and size as follows:

```
<array>, <array offset>, <array size>
```

The array contains the actual data being sent to or being received from SIM. The array offset indicates where the information you want to send to SIM is located in the array. The array size identifies the maximum length of the information that can be stored in the array. Using this format for an array allows you to store more than one piece of information in an array.

In the ALGOL program examples provided on the release media, the following define is used to put array information in the correct format:

```
DEFINE PASS_ARRAY(A) = A,0,SIZE(A) #;
```

If you are sending SIM string information that is not based on data contained in the SYMBOL/SIM/PROPERTIES file, you must ensure that the first word of the <array> construct contains the length of the string.

This section and Section 8 use the following convention to distinguish between information extracted from the SYMBOL/SIM/PROPERTIES file and user-defined strings:

- The term *record* refers to an array containing information based on fields defined in the SYMBOL/SIM/PROPERTIES file.
- The term *string* refers to a REAL array containing a one word size field followed by text. The data contained in the array is entry-point dependent.

Using the SIM Library Entry Points in COBOL and RPG Programs

In this section and in Section 8, “Describing the SIM Library Entry Points,” the entry points all start with SIM_. If you are using COBOL or RPG, replace *SIM_* with *SCIM_*.

The SCIM_ entry points also have an additional parameter not described in this section or Section 8. The additional parameter is inserted at the start of the parameter list and is used to return the result of the entry point. For example, the entry point SIM_OPEN_DATABASE is described as having four associated parameters: open_options, dbname_text, usercode_text, and family_text. In a COBOL or RPG program, the equivalent entry point, SCIM_OPEN_DATABASE, has five associated parameters: result, open_options, dbname_text, usercode_text, and family_text. You use the result parameter to establish whether there are any error messages associated with the processing of the entry point. Define the result parameter as a numeric of the following form:

```
PIC 9(19) COMP.
```

The form for using a SIM library entry point in a COBOL program is as follows:

```
CALL "<entry point> OF SIMDRIVERSUPPORT" USING <parameter list>
```

Including Required Program Elements

To use the entry points in your COBOL or RPG application, you must do either of the following:

- Replicate in your program the required record definitions.
- Put the record definitions in a library program and use the COPY statement to access the library program. An example copy deck, named EXAMPLE/SIM/COBOL/COPYDECK, is provided on the release media. For more information on using the COPY statement, refer to the *COBOL ANSI-74 Reference Manual, Volume 1: Basic Implementation.*

You must include the following line in your COBOL program:

```
CHANGE ATTRIBUTE LIBACCESS OF "SIMDRIVERSUPPORT" TO BYFUNCTION
```

Formatting Entry Points

In your program, the format of the statement including the entry point depends upon the programming language you are using. When an entry point is processed, a true or false value is associated with the entry point to indicate whether the attempt to process the entry point was successful.

In COBOL and similar languages, you extract the true or false value associated with the entry point by inserting an extra parameter at the beginning of the parameter list. The following COBOL74 program fragment illustrates the use of the SIM library entry point, SCIM_PUT_VALUE:

```
CALL "SCIM_PUT_VALUE OF SYSTEM/SIM/DRIVER" USING
      RESULT, CURSOR-INFO, ITEM-INFO, STRING-VALUE, NUMERIC-VALUE
IF NOT RESULT-EXCEPTION = 0 THEN
   MOVE "at put value" TO DIN_REC
   PERFORM ERROR_MSG
ELSE
   GO GET-PARAMETERS-EXIT.
```

Note: *In the ALGOL program fragment, the entry point has the name SIM_PUT_VALUE, and in the COBOL74 program fragment, the same entry point has the name SCIM_PUT_VALUE.*

For detailed examples illustrating the use of the SIM library entry points, refer to the sample programs contained on the release media.

Understanding Record Declarations in COBOL

In COBOL and similar languages, you must use a record declaration to pass and receive information from SIM. The release media provide examples of the record definitions you require. If the data you are passing the SIM is user-defined (for example, a database name) and not a value that is predefined by SIM (for example, the status of the database), you must give SIM the length of the data you are passing. In the sample COBOL74 program on the release media, the TALLYER routine is used to calculate the length of data and to move that information into the record passed to SIM.

If the information you are passing the SIM is a predefined value, then you must use a MOVE statement to associate the information with the record. For example, the following COBOL74 statement declares that the database is being opened for update purposes:

```
MOVE DB-PO-MODE-UPDATE TO DB-PO-MODE
```

This section uses the following convention to distinguish between information defined in the SYMBOL/SIM/PROPERTIES file and arrays that are application dependent that must be sent in record format:

- The term *record* refers to data based on the fields defined in the SYMBOL/SIM/PROPERTIES file.
- The term *string* refers to arrays containing a size field followed by text that are application dependent.

Processing Queries Using the SIM Library Entry Points

The following text provides a description of the tasks you need to accomplish to process a query. The tasks are described in the order in which they should be performed by your application. For each task, you are given the names of the entry points you can use to accomplish the task.

Defining the Database

Depending on whether the database you want to open is stored in an ADDS dictionary, you have to use one or two library entry points to open your database.

Opening a Dictionary

If your database is stored in an ADDS dictionary, the first entry point you must use is SIM_SET_DICTIONARY. If you want to use the system default dictionary, or if your database is not defined in a dictionary, you need not use this entry point.

There is one parameter associated with this entry point, called dictionary_name. You must declare the parameter as a string. The string contains the support library name associated with the data dictionary. The first word (48 bits) in the string defines the length of the text. An error is returned if you supply a blank or empty string.

For more information on using the SIM_SET_DICTIONARY entry point, refer to “SIM_SET_DICTIONARY” in Section 8, “Describing the SIM Library Entry Points.”

Opening a Database

To open a database, you use the SIM_OPEN_DATABASE entry point. The following four parameters are associated with the entry point:

- open_options

  Defines the mode, version, and status of the database. The version of a database is only applicable for databases that are stored in an ADDS dictionary.

- dbname_text

  Provides the name of the database.

- usercode_text

  Provides either the usercode associated with the database, or if the database is stored in ADDS, provides the ADDS directory associated with the database.

- family_text

  Defines the pack on which the database is stored.

For detailed information on using the SIM_OPEN_DATABASE entry point, refer to “SIM_OPEN_DATABASE” in Section 8, “Describing the SIM Library Entry Points.”

Opening a Flat File

To open a flat file, you use the SIM_OPEN_DATABASE entry point. This entry point uses four parameters as described under “Opening a Database” in this section. For a description of how the file appears to SIM, see “Querying Files” in Section 1, “Introduction to OML.”

If you created the flat file in IQF, then IQF automatically creates a description in the dictionary. If you did not create the file through IQF, then you must be aware of the following requirements:

- The file must first be loaded in the dictionary, for example, through the COBOL74LOADER utility. The file name following the header in the File Description (FD) entry becomes the file description name in the dictionary. This name is also known as the internal name.
- The COBOL *FD* entry should contain one of the following VALUE OF clauses:
  - VALUE OF TITLE IS literal
  - VALUE OF FILENAME IS literal

  The literal in these clauses is used as the file title when the flat file is opened for retrieval. If you do not specify either clause, the internal name is used as the file title. If you use both clauses, the VALUE OF TITLE clause takes precedence. The VALUE OF TITLE clause allows a usercode and a family name in the literal.

For more information on the FD entry, or the VALUE OF TITLE and VALUE OF FILENAME clauses, refer to the *COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation.* For more information on the COBOL74LOADER utility, refer to the *InfoExec ADDS Operations Guide.* For more information on IQF, see the *InfoExec IQF Operations Guide.*

Parsing the Query Text

A query is a request to take a snapshot of the database. The query describes what is being accessed, or in the case of an update query what is being altered, but it does not dictate when the access or the update is to occur.

Use the SIM_PARSE_QUERY, the SIM_PARSE_FORMATTED_QUERY, or the SIM_PARSE_SQL entry point to compile the query.

When the query is sent to the SIM library, the query is compiled. If there are no compilation errors, a query identifier is associated with the text.

The query identifier is assigned by the Query Driver, which is a program that acts as an interface between the user program and the database.

At this point, the query is compiled for syntax only. The database has not been accessed and no attempt has been made to retrieve or update data.

If you are processing a query written in SIM OML, you must use either the SIM_PARSE_QUERY or the SIM_PARSE_FORMATTED_QUERY entry point. The SIM_PARSE_FORMATTED_QUERY allows you to designate the format for the data you receive from SIM; for example, you might choose to receive the internal form of a symbolic. If you are processing a query written in SQLDB DML, you must use the SIM_PARSE_SQL or SIM_PARSE_FORMATTED_QUERY entry point.

The SIM_PARSE_QUERY and SIM_PARSE_SQL entry points both have the following two parameters:

- query_info

  Defines how you want to format data types and returns a query identifier.

- qtext

  Provides the text of the query.

For more detailed information on using the SIM_PARSE_QUERY and SIM_PARSE_SQL entry points, refer to “SIM_PARSE_QUERY” and “SIM_PARSE_SQL” in Section 8, “Describing the SIM Library Entry Points.”

The SIM_PARSE_FORMATTED_QUERY entry point has the following three parameters:

- query_info

  Defines how you want to format data types and returns a query identifier.

- qtext

  Provides the text of the query.

- qdesc

  Defines the data formatting options you want to use and enables SIM to return the description of the query.

For detailed information on using the QDESC record to set formatting options, refer to “Using the QDESC Record” later in this section. For more information on the SIM_PARSE_FORMATTED_QUERY entry point, refer to “SIM_PARSE_FORMATTED_QUERY” in Section 8, “Describing the SIM Library Entry Points.”

If there are errors when you parse your query, use the SIM_EXCEPTION_INFO, SIM_EXCEPTION_MESSAGE, SIM_NEXT_EXCEPTION, and SIM_GET_EXCEPTION_INFO entry points to retrieve the error messages. For more information on these three entry points, refer to “Returning Error Messages” later in this section. You should refer to Appendix A, “OML Error Messages,” and to Appendix B, “Exception Fields and Categories,” for a list of the error messages and exception messages that might be returned when you process an OML query. Refer to the *InfoExec SQLDB Programming Guide* for an explanation of the syntax error messages that might be returned when you process an SQLDB query.

Performing Queries While in Transaction State

In order to perform an update query, your program must first enter transaction state. It is also good programming practice, though not required, to perform retrieval queries in transaction state.

Note: *You cannot enter transaction state against a database or a file that only allows inquiries. These types of databases or files include the following:*

- *Logic and Information Network Compiler II (LINC II) databases that have been processed by LINC.View.*
- *Data Management System II (DMSII) databases that have been processed by DMS.View and that do not allow update queries.*
- *Any database that uses the Remote Database Backup (RDB) system and is operating in the secondary role. For more information on RDB databases, see the* Remote Database Backup (RDB) Planning and Operations Guide.
- *Files defined in an Advanced Data Dictionary System (ADDS) data dictionary.*

The following discussion explains transactions and the entry points you can use to begin and end transactions, to mark reference points (known as savepoints) while you are in transaction state, and to cancel updates.

Understanding Transactions

An update to a database system is called a transaction. A single transaction can consist of multiple update statements but the transaction itself is an all-or-nothing unit of work; all updates specified by a transaction must complete satisfactorily before any of the requested database changes are made permanent.

In IQF each update to a database is, by default, a separate transaction. However, there are times when it is desirable to group a series of individual update statements into a single transaction; this enables you to ensure that all the updates complete satisfactorily before the database is changed permanently.

It is also possible, through the use of savepoints, to break a multistatement transaction into subunits. Savepoints enable you to undo the changes made since a specific savepoint without undoing all the changes made during the transaction. In other words, savepoints enable you to undo the changes made by the last *n* subunits of the transaction instead of forcing you to reprocess the entire transaction from the start.

Between the time a program begins and ends a transaction, it is said to be in transaction state. There are three types of transaction states, as follows:

- Record-level

  This level of locking secures only the records affected by your query. This is the normal locking strategy. Record-level locking provides shared locks for retrieval queries and exclusive locks for update queries.

- Database-level

  This level of locking provides you with exclusive use of the database. That is, once your program enters transaction state, no other process can enter transaction state. While you are in transaction state any other process suspends when it attempts to enter transaction state. However, *dirty reads* are allowed. The term *dirty reads* refers to performing retrieval queries outside of transaction state.

- Structure-level

  This level of locking secures all structures touched by your query. If you use structure-level locking, there is a high probability of you having deadlock problems because all locks are obtained when you process the cursor. Structure-level locking provides shared locks for retrieval queries and exclusive locks for update queries.

Shared locks allow concurrent access to the same record or structure for retrieval queries. Exclusive locks do not allow any access to the same record or structure. Until the program that has the exclusive lock on the record or structure exits transaction state, the record or structure remains locked. All other programs wishing to access that record or structure must wait until the record or structure is unlocked. Therefore, you should make transactions as short as possible. The longer that a program is in transaction state, the greater the chance that another program will be forced to wait for a locked record or structure.

Additionally, the system can make programs wait to begin a transaction or wait to close a database if a syncpoint is required. Each syncpoint is marked by the system as a recovery checkpoint. The DMSII option SYNCPOINT can be used to regulate the occurrences of the syncpoints.

The system of safeguards that ensures that all completed transactions are recovered is called concurrency control. Concurrency control is handled directly by the DMSII physical layer underlying your database. For an overview of DMSII database recovery including concurrency control and syncpoints, refer to the *DMSII Technical Overview*.

Signaling the Beginning of a Transaction

You indicate to SIM that you wish to begin transaction state by calling either of the following entry points:

- SIM_BEGIN_TRANSACTION
- SIM_COMS_BEGIN_TRANSACTION

  You use this entry point only if you want to start a Communications Management System (COMS) transaction.

Starting a Transaction Outside of COMS

The SIM_BEGIN_TRANSACTION entry point has one associated parameter—locking_level. You can assign locking_level any of the following values:

Number	Name	Description
0 (zero)	RECORD_LEVEL_LOCKING_V	Requests record-level locking.
1	DATABASE_LEVEL_LOCKING_V	Requests database-level locking.
2	STRUCTURE_LEVEL_LOCKING_V	Requests structure-level locking.

Refer to “Example Using the Transaction-Related Entry Points” later in this section for an illustration of how to use the SIM_BEGIN_TRANSACTION entry point.

Starting a COMS Transaction

The SIM_COMS_BEGIN_TRANSACTION entry point has the following two parameters:

- locking_level

  You can assign the locking_level parameter any of the following values:

Number	Name	Description
0 (zero)	RECORD_LEVEL_LOCKING_V	Requests record-level locking.
1	DATABASE_LEVEL_LOCKING_V	Requests database-level locking.
2	STRUCTURE_LEVEL_LOCKING_V	Requests structure-level locking.

- coms_in

  The coms_in parameter is a string parameter that you use to send and receive information from COMS.

To use the SIM library interface with COMS, you must have the following INCLUDE statement in your program:

```
$INCLUDE UNIVERSALS = "SYMBOL/SIM/PROPERTIES" 3010000-3056000
```

You must place this INCLUDE statement after the message declarations as it assumes the existence of the COMS_IN message area.

You access the COMS library by function using the standard support library names as follows:

```
COMSSUPPORT.LIBACCESS:= VALUE(BYFUNCTION);
COMSSUPPORT.FUNCTIONNAME:= "COMSSUPPORT";
```

Refer to “Example Using the COMS Transaction Entry Points” later in this section for an example illustrating the use of this entry point. For detailed information on COMS transactions, refer to the *COMS Programming Guide.*

Signaling the End of a Transaction

Once you have finished querying the database, you need to issue an end-of-transaction message to signal that you have finished altering or retrieving information from the database.

Once your program is out of transaction state, other users can access and alter the entities affected by your query.

Ending a Transaction Outside of COMS

Use the SIM_ETR entry point to end your transaction if you started your transaction with the SIM_BEGIN_TRANSACTION entry point.

There are no parameters associated with the SIM_ETR entry point.

Refer to “Example Using the Transaction-Related Entry Points” later in this section for an illustration of how to use the SIM_ETR entry point.

Note: *Use the SIM_BEGIN_TRANSACTION and SIM_ETR entry points with retrieval queries to ensure that no one alters database information while you are accessing it.*

Ending a COMS Transaction

Use the SIM_COMS_ETR entry point if you started your transaction with the SIM_COMS_BEGIN_TRANSACTION entry point. There is one string parameter associated with this entry point—coms_out.

Refer to “Example Using the COMS Transaction Entry Points” later in this section for an illustration of how to use this entry point.

Marking Savepoints

During a transaction, you can mark points known as savepoints. Using savepoints enables you to undo a subset of the changes you made to the database during the transaction. You use the SIM_SAVEPOINT entry point to mark savepoints. There is one parameter—savepoint—associated with the SIM_SAVEPOINT entry point. The savepoint parameter can take any positive integer value. If you place more than one savepoint during a transaction, you should assign a unique integer to each savepoint. Doing so allows you more flexibility if you need to undo some of the changes you have made.

The savepoint markers are local to your transaction and disappear when your transaction is complete. To use the savepoint markers fully, you must add the appropriate code to your program.

Refer to "Example Using the Transaction-Related Entry Points" later in this section for an example of how to use the SIM_SAVEPOINT entry point.

Canceling Transactions

In the SIM library interface, entry points are provided that enable you to undo all changes you made to the database since your last begin-transaction marker or your last savepoint marker.

Undoing All Updates Since the Last Begin-Transaction Marker

You indicate to SIM that you wish to end a transaction and discard all the requested updates by calling either of the following two entry points:

- SIM_ABORT
- SIM_COMS_ABORT

  You use this entry point only if you wish to end a COMS transaction.

Undoing an Update Outside of COMS

Use the SIM_ABORT entry point to undo all changes you made to the database since the last begin-transaction marker. The SIM_ABORT entry point also ends the transaction. There are no parameters associated with the SIM_ABORT entry point.

Refer to "Example Using the Transaction-Related Entry Points" later in this section for an illustration of how to use the SIM_ABORT entry point.

Undoing a COMS Update

Use the SIM_COMS_ABORT entry point to undo all changes you made to the database since the last COMS begin-transaction marker. There is one string parameter associated with the SIM_COMS_ABORT entry point—coms_out.

Refer to "Example Using the COMS Transaction Entry Points" later in this section for an illustration of how to use this entry point.

Undoing All Updates Since the Last Savepoint Marker

To cancel all updates you made to the database since a designated savepoint, use the entry point SIM_CANCEL. There is one parameter—savepoint—associated with the entry point. You use the savepoint parameter to designate the point after which all changes are to be canceled.

Note: *When you use the SIM_CANCEL entry point your program is not taken out of transaction state. You must explicitly end the transaction.*

Refer to “Example Using the Transaction-Related Entry Points” later in this section for an illustration of how to use the SIM_CANCEL entry point.

Example Using the Transaction-Related Entry Points

Example 7–1 demonstrates how to begin a transaction, specify a savepoint, and end a transaction. In the example, the code for the actual updates is not provided, but you are shown where the update code should be placed.

In the example, the following occurs:

- Three different update queries are parsed.
- The program enters exclusive transaction state.
- The first update query is processed.
- A savepoint is issued.
- The second update query is processed.
- A second savepoint is issued.
- The third update query is processed.
- The third update is undone by canceling all transactions after the second savepoint.
- The first two updates are saved by ending the transaction.

Error handling code is included in the example. For information on the PROCESS_ERROR procedure, refer to “Returning Error Messages” later in this section.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” earlier in this section for an explanation of the PASS_ARRAY define.

```
PROCEDURE TRANSACTION_EXAMPLE;
     BOOLEAN
         RESULT,           % Holds the SIM result word.
         IN_TR_STATE;      % TRUE if the program is in
                           %    transaction state.
     INTEGER
         LEVEL_LOCKING     % Takes value 0 for record-level locking,
                           %       value 1 for database-level locking,
                           %       value 2 for structure-level locking.
     ARRAY
         QUERY_TEXT      [0:30],
         QUERY_INFO_1,
         QUERY_INFO_2,
         QUERY_INFO_3    [0:PO_SIZE],
         CURSOR_INFO_1,
         CURSOR_INFO_2,
         CURSOR_INFO_3   [0:IC_CURSOR_SIZE];
     LABEL
         END_OF_PROCEDURE;

     % Parse the first update query.
     REPLACE QUERY_TEXT[1] BY

      "Modify PERSON (AGE:= AGE + 1) where PERSON-ID = 10";
     QUERY_TEXT[0]:= 51;
     IF RESULT:= SIM_PARSE_QUERY (PASS_ARRAY (QUERY_INFO_1)
                                ,PASS_ARRAY (QUERY_TEXT)
                                ) THEN
         BEGIN
         %  There was an error when trying to parse the query.
         %  For an explanation of PROCESS_ERROR, see the example
         %  program for SIM_EXCEPTION_MESSAGE.
         PROCESS_ERROR;
         GO END_OF_PROCEDURE;          % Exits the procedure
         END;

     %  Parse the second update query. Note that the QUERY_TEXT
     %  array can be reused when parsing queries but that the
     %  QUERY_INFO_n arrays cannot be reused because you do not
     %  want to lose the information for the previously parsed
     %  queries.
```

Example 7–1. Transaction-Related Entry Points (cont.)

```
REPLACE QUERY_TEXT[1] BY
   "Modify PERSON (AGE:= AGE + 1) where PERSON-ID = 20";
QUERY_TEXT[0]:= 51;
IF RESULT:= SIM_PARSE_QUERY (PASS_ARRAY (QUERY_INFO_2)
                            ,PASS_ARRAY (QUERY_TEXT)
                           ) THEN
    BEGIN
    %  There was an error when trying to parse the query.
    PROCESS_ERROR;
    GO END_OF_PROCEDURE;
    END;

%  Parse the third update query.
REPLACE QUERY_TEXT[1] BY
   "Modify PERSON (AGE:= AGE + 1) where PERSON-ID = 30";
QUERY_TEXT[0]:= 51;
IF RESULT:=SIM_PARSE_QUERY (PASS_ARRAY (QUERY_INFO_3)
                            ,PASS_ARRAY (QUERY_TEXT)
                            ) THEN
    BEGIN
    %  There was an error when trying to parse the query.
    PROCESS_ERROR;
    GO END_OF_PROCEDURE;
    END;

%  Now that the queries have been parsed, the program can enter
%  transaction state.

LEVEL_LOCKING:=DATABASE_LEVEL_LOCKING_V;  %  Exclusive use of the
                                          %   database requested.
IF RESULT:=  SIM_BEGIN_TRANSACTION (LEVEL_LOCKING) THEN
    BEGIN
    %  There was an error when trying to enter transaction
    %  state.  If there is an error calling
    %  SIM_BEGIN_TRANSACTION, the
    %  program does not go into transaction state.
    PROCESS_ERROR;
    GO END_OF_PROCEDURE;
    END;

%  The program is now in transaction state.
IN_TR_STATE:= TRUE;

%  Process the first query.
IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_1)
                            ,PASS_ARRAY (CURSOR_INFO_1)
                            ) THEN
```

Example 7–1. Transaction-Related Entry Points (cont.)

```
      BEGIN
      %  Some kind of error occurred.
      PROCESS_ERROR;
      GO END_OF_PROCEDURE;
      END
   ELSE
      BEGIN
      % Now update the database.
      IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_1))
         THEN
         BEGIN
         % Some kind of error occurred.
         PROCESS_ERROR;
         GO END_OF_PROCEDURE;
         END;
      END;

  %  Mark the first savepoint.
  IF RESULT:= SIM_SAVEPOINT (1) THEN
      BEGIN
      %  Some kind of error occurred while trying to mark the
      %  savepoint.  If an error occurs while using the SIM_SAVEPOINT
      %  entry point, the savepoint is not marked but the
      %  program remains in transaction state.
      PROCESS_ERROR;
      GO END_OF_PROCEDURE;
      END;

  %  Process the second query.
  IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_2)
                              ,PASS_ARRAY (CURSOR_INFO_2)
                              ) THEN
      BEGIN
      %  Some kind of error occurred.
      PROCESS_ERROR;
      GO END_OF_PROCEDURE;
      END
   ELSE
      BEGIN
      % Now update the database.
      IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_2))
         THEN
         BEGIN
         % Some kind of error occurred.
         PROCESS_ERROR;
         GO END_OF_PROCEDURE;
         END;
      END;
```

Example 7–1. Transaction-Related Entry Points (cont.)

```
%  Mark the second savepoint.
IF RESULT:= SIM_SAVEPOINT (2) THEN
    BEGIN
    %  Some kind of error occurred while trying to mark the
    %  savepoint.
    PROCESS_ERROR;
    GO END_OF_PROCEDURE;
    END;

%  Process the third query.
IF RESULT:=SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_3)
                            ,PASS_ARRAY (CURSOR_INFO_3)
                           ) THEN
    BEGIN
    %  Some kind of error occurred.
    PROCESS_ERROR;
    GO END_OF_PROCEDURE;
    END
 ELSE
    BEGIN
    % Now update the database.
    IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_3))
       THEN
       BEGIN
       % Some kind of error occurred.
       PROCESS_ERROR;
       GO END_OF_PROCEDURE;
       END;
    END;

%  Undo the third update by canceling all updates since
%  the second savepoint.
IF RESULT:= SIM_CANCEL (2) THEN
    BEGIN
    %  Some kind of error occurred.  An error when calling
    %  SIM_CANCEL indicates that the updates are not canceled
    %  and that the database is still in  transaction state.
    PROCESS_ERROR;
    GO END_OF_PROCEDURE;
    END;
```

Example 7–1. Transaction-Related Entry Points (cont.)

```
    %  If you get to this part of the program, then only the first
    %  two queries have been processed. In order to make the
    %  the changes permanent, you must end the transaction.
    IF RESULT:= SIM_ETR THEN
        BEGIN
        %  Some kind of error occurred.  An error during a call
        %  to SIM_ETR means the transaction did not complete
        %  successfully, the changes are not made to the
        %  database, and the program is no longer in
        %  transaction state.
        PROCESS_ERROR;
        IN_TR_STATE:= FALSE;
        DISPLAY("Transaction aborted during ETR");
        END
    ELSE
        BEGIN
        DISPLAY ("Transaction completed successfully");
        IN_TR_STATE:= FALSE;
        END;

END_OF_PROCEDURE:
    IF IN_TR_STATE THEN
        %  If the program is still in transaction state,
        %  one of the calls between SIM_BEGIN_TRANSACTION and SIM_ETR
        %  returned an error.  You must use the SIM_ABORT entry point
        %  to take the program out of transaction state. If you do
        %  not take the program out of transaction state, any
        %  subsequent attempts to enter transaction state fail with
        %  the error "Already in transaction state."
        BEGIN
        SIM_ABORT;
        DISPLAY ("Transaction aborted");
        END;
    END;
```

Example 7–1. Transaction-Related Entry Points

Example Using the COMS Transaction Entry Points

Example 7–2 illustrates the use of the COMS transaction entry points. The program uses the predeclared types INPUTHEADER and OUTPUTHEADER, which are provided for you by the SIM compiler, and the user-declared type COMS_IN_TYPE. These three types are used to declare the two message areas, COMS_IN and COMS_OUT, that are used in communicating with COMS. The ALGOL declaration for these types is as follows:

```
TYPE INPUTHEADER
        COMS_IN_TYPE(ARRAY CONVERSATION [0:59]);
COMS_IN_TYPE
        COMS_IN;
OUTPUTHEADER
        COMS_OUT;
```

The COMS_IN message area is initialized and COMS is informed that the program is ready to receive input by using the following code:

```
ENABLE(COMS_IN,"ONLINE");
```

Once the database is open, the program requests a message from COMS and continues processing messages until COMS_STATUS has the value EOF_NOTICE (99). (COMS_STATUS is given a data type of real in ALGOL and is used to capture the result of calls on COMS.) The request is made using the following code:

```
DO
        PROCESS_COMS_INPUT
        UNTIL COMS_STATUS = EOF_NOTICE;
```

In the PROCESS_COMS_INPUT procedure, the program receives messages by using the COMS *RECEIVE* statement:

```
COMS_STATUS:= RECEIVE(COMS_IN,EMSG_TEXT);
```

If the status is acceptable and the COMS_IN FUNCTIONINDEX field indicates that a request has been received to process a message, a transaction is processed using the following code:

```
IF (COMS_STATUS = 0 OR COMS_STATUS = 92) AND
            COMS_IN.FUNCTIONINDEX >= 0 THEN
            PROCESS_TRANSACTION;
```

A transaction is initiated by calling the SIM_COMS_BEGIN_TRANSACTION entry point, which passes to SIM the message area used in the COMS *RECEIVE* statement.

```
IF  RSLT:= SIM_COMS_BEGIN_TRANSACTION(0,COMS_IN) THEN
        EXIT("BTR");
```

The transaction is processed using the standard SIM library entry points. At the completion of the transaction the SIM library entry point SIM_COMS_ETR is invoked. The SIM_COMS_ETR entry point uses the message area that COMS returns to the user:

```
IF  RSLT:= SIM_COMS_ETR(COMS_OUT) THEN
        EXIT("ETR");
```

The message returned by COMS has the following format:

```
COMS_OUT.DESTCOUNT:= 1;
COMS_OUT.DESTINATIONDESG:= COMS_IN.STATION;
COMS_OUT.STATUSVALUE:= 0;
COMS_STATUS:=
        SEND(COMS_OUT, OFFSET(POBUFF), EOBUFF);
IF NOT (COMS_STATUS = 0 OR COMS_STATUS = 92) THEN
        DISPLAY("Send error: "!! STRING8(COMS_STATUS,*));
```

The complete code for performing a COMS transaction looks as follows:

```
$RESET LIST STACK
 $SET LINEINFO
 $SET TADS
BEGIN
 $SET ALGOLCOMPILE COMSPROGRAM
TYPE INPUTHEADER
        COMS_IN_TYPE(ARRAY CONVERSATION [0:59]);
COMS_IN_TYPE
        COMS_IN;
OUTPUTHEADER
        COMS_OUT;

 $INCLUDE UNIVERSALS = "SYMBOL/SIM/PROPERTIES" 2000000-2099999

DEFINE
    DEBUG      = FALSE #,
    PQ(Q,QI)   = BEGIN                    % Parse a query.
                   REPLACE PQT:POINTER(QTEXT[1]) BY Q FOR STR_SIZE
                       WHILE NEQ ";";
                   QTEXT[0]:= OFFSET(PQT)-6;
                   IF  DEBUG THEN
                       SHOW("Parsing: "!! STRING(Q,QTEXT[0]));
                   IF  RSLT:= SIM_PARSE_QUERY(P(QI),P(QTEXT)) THEN
                       EXIT("PARSE QUERY");
                   IF  RSLT:= SIM_DESCRIBE_QUERY(
                       P(QI),
                       P(QUERY_INFO)) THEN
                       EXIT("DESCRIBE_QUERY");
                   END #,
```

Example 7–2. COMS Transaction Entry Points (cont.)

```
    OC(Q,C)    = BEGIN                    % Open a cursor.
                   IF  DEBUG THEN
                       SHOW("Open cursor");
                   IF  RSLT:= SIM_OPEN_CURSOR(P(Q),P(C)) THEN
                       EXIT("OPEN CURSOR");
                   END #,
    PC(C)      = BEGIN                    % Process a cursor.
                   IF  DEBUG THEN
                       SHOW("Process cursor");
                   IF  RSLT:= SIM_PROCESS_CURSOR(P(C)) THEN
                       IF  REAL(RSLT).EX_CATEGORYF NEQ EX_COMPLETE THEN
                           EXIT("PROCESS CURSOR");
                   END #,
    P(X)       = X, 0, SIZE(X) #,         % For passing array
                                          %   parameters.
    STR_SIZE   = 200 #;                   % Maximum used by
                                          %   this program.
ARRAY
    ITEM_ID,                              % String for TGL element name.
    ITEM_VALUE,                           % String for SIM_GET_VALUE.
    QTEXT,                                % Text for SIM_PARSE_QUERY.
    NTEXT,                                % Name of database.
    UTEXT,                                % User/directory of database.
    FTEXT,                                % Family of database.
    LANG,                                 % Language for error message.
    MSG_TEXT,                             % Message text for error
                                          %    messages.
    OBUFF     [0:STR_SIZE],               % Output buffer.
    IQ,                                   % Insert query.
    DQ        [0:PO_SIZE],                % Delete query.
    DC,                                   % Delete cursor.
    IC        [0:IC_USER_CURSOR_SIZE],    % Insert cursor.
    OPEN_PO   [0:SIM_OPEN_DATABASE_SIZE], % Database open options
                                          %   record.
    QUERY_INFO[0:SIM_Q_SIZE],             % Information
                                          %   for SIM QUERY RECORD.
    ITEM_INFO [0:SIM_ITEM_RECORD_SIZE];   % Information for
                                          %   SIM_PUT_VALUE.
DOUBLE
    NV;                                   % Numeric value.
BOOLEAN
    RSLT;                                 % Returned by SIM_...
FILE DOUT(KIND = DISK,                    % Output file.
    MAXRECSIZE = STR_SIZE,
    BLOCKSIZE = 420,
    PROTECTION = SAVE);
DEFINE
        EOF_NOTICE  = 99 #;
```

Example 7–2. COMS Transaction Entry Points (cont.)

```
REAL    COMS_STATUS;
EBCDIC ARRAY
        EOBUFF[0] = OBUFF,
        EMSG_TEXT[0] = MSG_TEXT,
        SCRATCH[0:255];
POINTER
        PQT,
        POBUFF;
PROCEDURE SEND_MSG;
    BEGIN
    COMS_OUT.DESTCOUNT:= 1;
    COMS_OUT.DESTINATIONDESG:= COMS_IN.STATION;
    COMS_OUT.STATUSVALUE:= 0;
    COMS_STATUS:=
            SEND(COMS_OUT, OFFSET(POBUFF), EOBUFF);
    IF NOT (COMS_STATUS = 0 OR COMS_STATUS = 92) THEN
            DISPLAY("Send error: "!! STRING8(COMS_STATUS,*));
    POBUFF:= POINTER(OBUFF[0]);
    END SEND_MSG;
PROCEDURE SHOW(WS);
    % Write a string to the output file.
    VALUE WS;
    STRING WS;
    BEGIN
    REPLACE POBUFF:POBUFF BY WS;
    END;
DEFINE EXIT(X) =
    % Write a string to the output file, if an error has occurred,
    % get the error message and write it to the output file.
    BEGIN
    SHOW(X);
    IF  RSLT THEN
        IF  REAL(RSLT).EX_CATEGORYF NEQ EX_COMPLETE AND
           REAL(RSLT).EX_CATEGORYF NEQ EX_NONE THEN
           DO  IF SIM_EXCEPTION_MESSAGE(P(LANG),P(MSG_TEXT)) NEQ 0 THEN
                   BEGIN
                   SHOW("Error getting exception text");
                   SHOW("INFO = "!!STRING(REAL(RSLT).EX_INFOF,*)!!
                        ",CAT="!!STRING(REAL(RSLT).EX_CATEGORYF,*)!!
                        ",SUBCAT="!!STRING(REAL(RSLT).EX_SUBCATEGORYF,*)
                      !!", AT "!!STRING(LINENUMBER,*)
                       );
                   END
               ELSE
                   SHOW(STRING(POINTER(MSG_TEXT[1]),MSG_TEXT[0]))
               UNTIL NOT RSLT:= SIM_NEXT_EXCEPTION;
    GO EXIT_HERE;
    END #;
```

Example 7–2. COMS Transaction Entry Points (cont.)

```
PROCEDURE QDC(S);
    VALUE S;
    STRING S;
    BEGIN
    ARRAY QDCA[0:100];
    REPLACE POINTER(QDCA[1]) BY S;
    QDCA[0]:= LENGTH(S);
    IF  SIM_QD_CONTROLLER(P(QDCA)) THEN
        DISPLAY(POINTER(QDCA[1]));
    END;
PROCEDURE PROCESS_TRANSACTION;
    BEGIN
    LABEL
        EXIT_HERE;
    IF  RSLT:= SIM_COMS_BEGIN_TRANSACTION(0,COMS_IN) THEN
        EXIT("BTR");
    % Parse the query
    PQ(POINTER(MSG_TEXT[0]),DQ);
    % Open the cursor
    OC(DQ,DC);
    % Process the query
    PC(DC);
    IF  SIM_Q_STATEMENT(QUERY_INFO) GTR ST_RETRIEVE_V THEN
        SHOW(STRING(IC_TUPLE_COUNT(DC),*)!!
            " processed")
    ELSE
        DO  BEGIN
            SIM_ITEM_NUMBER(ITEM_INFO):= 1;
            WHILE NOT RSLT:=
                SIM_GET_VALUE(
                    P(DC),
                    P(ITEM_INFO),
                    P(ITEM_VALUE),
                    NV) DO
                    BEGIN
                    RSLT:= SIM_DESCRIBE_ITEM(
                        P(DQ), P(ITEM_INFO), P(ITEM_ID));
                    SHOW(STRING(POINTER(ITEM_ID[1]),ITEM_ID[0]));
                    SHOW("<");
                    SHOW(STRING(POINTER(ITEM_VALUE[1]),ITEM_VALUE[0]));
                    SHOW(">");
                    SIM_ITEM_NUMBER(ITEM_INFO):= *+1;
                    END;
            PC(DC);
            END UNTIL RSLT;
EXIT_HERE:
```

Example 7–2. COMS Transaction Entry Points (cont.)

```
    % Get out of transaction state.
    IF  RSLT:= SIM_COMS_ETR(COMS_OUT) THEN
        EXIT("ETR");
    SEND_MSG;
    END;

PROCEDURE PROCESS_COMS_INPUT;
    BEGIN
    REPLACE MSG_TEXT BY " " FOR STR_SIZE;
    SHOW(48"0A0D");
    COMS_STATUS:=                  RECEIVE(COMS_IN,EMSG_TEXT);
    IF (COMS_STATUS = 0 OR COMS_STATUS = 92) AND
            COMS_IN.FUNCTIONINDEX >= 0 THEN
            PROCESS_TRANSACTION;
    END;

SIM_LIBRARY.LIBACCESS:=VALUE(BYTITLE);
SIM_LIBRARY.TITLE:= "(JPT)SYSTEM/SIM/DRIVER/SQL ON ISYS";
%SIM_LIBRARY.FUNCTIONNAME:= "SIMDRIVER39";

COMSSUPPORT.LIBACCESS:= VALUE(BYFUNCTION);
COMSSUPPORT.FUNCTIONNAME:= "COMSSUPPORT";
ENABLE(COMS_IN,"ONLINE");

POBUFF:= POINTER(OBUFF[0]);
% Prepare to open the database.
REPLACE POINTER(NTEXT[1]) BY "BLOGGSDB";
NTEXT[0]:= LENGTH("BLOGGSDB");
IF  DEBUG THEN
    SHOW("Database is "!!STRING(POINTER(NTEXT[1]),NTEXT[0]));
SIM_OPEN_DATABASE_MODE (OPEN_PO):= 0;
IF  RSLT:= SIM_OPEN_DATABASE(P(OPEN_PO),P(NTEXT),P(UTEXT),P(FTEXT)) THEN
    BEGIN
    SHOW("Open Database Error");
    SEND_MSG;
    END;
DO
        PROCESS_COMS_INPUT
        UNTIL COMS_STATUS = EOF_NOTICE;
END.
```

Example 7–2. COMS Transaction Entry Points

Associating a Cursor with a Compiled Query

Once a query has been compiled without error, and if required, you have put your program in transaction state, you must use the SIM_OPEN_CURSOR entry point to associate a cursor with the query. The SIM library uses the query identifier to identify compiled queries; cursors are used to distinguish between instances of a query.

Once you have associated a cursor with a query, you then do either of the following:

- For a retrieval query
    1. Associate values with any query parameters.
    2. Process the query.
    3. Extract values.
- For an update query
    1. Associate values with any query parameters.
    2. Process the query.

Having both a cursor and a query identifier reduces overhead because recompiling the query each time it is processed is unnecessary. Since a cursor represents one instance of a query, you can associate more than one cursor with a single query. If you do this, the same action is occurring but at different points in time, or with different input. For example, you could use different values for query parameters and rerun a retrieval query at hourly intervals.

You assign a cursor to a query by using the SIM_OPEN_CURSOR entry point. There are two read-only parameters associated with the SIM_OPEN_CURSOR entry point—query_info and cursor_info. For detailed information on how to use the SIM_OPEN_CURSOR entry point, refer to “SIM_OPEN_CURSOR” in Section 8, “Describing the SIM Library Entry Points.”

Using Parameters in a Query

Queries can include elements with the format #<*token*>. (SQLDB DML queries can also use elements with the format :<*token*>.) These elements indicate that the query text contains parameters to which values are assigned after opening a cursor but before processing the cursor. A query whose text includes elements of this type is referred to as a *parameterized query.*

Parameterized queries are useful when a basic query is performed multiple times with the only change to the query text being constant values.

For example, if you are updating employee salaries in the ORGANIZATION database, you might need to perform the following queries:

- MODIFY EMPLOYEE (SALARY:= SALARY + 100) WHERE EMPLOYEE_ID = 10
- MODIFY EMPLOYEE (SALARY:= SALARY + 1500) WHERE EMPLOYEE_ID = 30
- MODIFY EMPLOYEE (SALARY:= SALARY + 300) WHERE EMPLOYEE_ID = 32
- MODIFY EMPLOYEE (SALARY:= SALARY + 2000) WHERE EMPLOYEE_ID = 76

Usually having these types of query requires that you parse each query separately. However, by using parameterized queries, you can define the following query. Then you need parse the text of the query only once and assign values to the parameters #RAISE and #ID using the cursor for each update you want to perform.

```
MODIFY EMPLOYEE (SALARY:= SALARY + #RAISE)
     WHERE EMPLOYEE_ID = #ID
```

You assign parameter values using the SIM_PUT_VALUE entry point. There is no limit to the number of times a parameterized query can be processed.

Gathering Parameter Information

The data type of a query parameter is determined by the attribute associated with the query parameter and on whether you designate the default or alternative data format when you parse the query. For example, if your query includes the following element and the attribute EMPLOYEE_ID is declared as an integer, then the data type associated with the parameter #*1* is integer:

```
EMPLOYEE_ID:= #ID,
```

By default, attributes with a data type of date, time, symbolic, character, Kanji, or string are processed as strings. However, by setting the appropriate parser options before you call the SIM_PARSE_QUERY entry point, you can choose to process attributes with a data type of date, time, or symbolic as integers.

Table 7–1 explains the correlation between the numeric value returned by the item type parameter, the name given to the item type in the SYMBOL/SIM/PROPERTIES file, and the data type associated with the item type.

Table 7–1. Item Values Associated with Data Types

Item Numeric Value	Item Name	Data Type
0	ITEM_IS_REAL_V	Real
2	ITEM_IS_INTEGER_V	Integer
3	ITEM_IS_BCD_V	Binary-coded decimal (BCD)
4	ITEM_IS_STRING_V	String
5	ITEM_IS_VLSTRING_V	Variable-length string
6	ITEM_IS_BOOLEAN_V	Boolean
7	ITEM_IS_DOUBLE_V	Double
14	ITEM_IS_CURSOR_NO_V	The cursor number for the current query
15	ITEM_IS_SURROGATE_V	Entity reference variable

You can use the SIM_DESCRIBE_PARAMETER entry point to determine the data type of any query parameter. For information on the SIM_DESCRIBE_PARAMETER, refer to "SIM_DESCRIBE_PARAMETER" in Section 8, "Describing the SIM Library Entry Points."

Passing Parameter Values to a Query

To pass a value to a parameter when the query is being processed, use the SIM_PUT_VALUE entry point. The following four parameters are associated with the SIM_PUT_VALUE entry point:

- cursor_info

  Identifies the query.

- item_info

  Identifies the parameter to which you want to assign a value.

- string_value

  Contains the string you want to assign to the parameter.

- numeric_value

  Contains the numeric value you want to assign to the parameter.

For more information on the SIM_PUT_VALUE entry point, refer to "SIM_PUT_VALUE" in Section 8, "Describing the SIM Library Entry Points."

Processing a Retrieval Query and Extracting the Output

Once you have compiled your query and provided values for any query parameters, you must perform the following two steps:

1. Process the query.

   You perform this step once for each entity you want to retrieve.

2. Extract the output.

   For each entity you retrieve, you do either of the following:

   - Perform this step once for each item in the query target list that you want to retrieve.
   - Retrieve the information as a record and then decode the record in your program.

These steps are explained in the following text.

Processing the Query

Once a query has been compiled and a cursor has been associated with it, the next step is to process the query. Use the SIM_PROCESS_CURSOR entry point to initiate query processing. For a retrieval query, you must use this entry point once for each entity affected by the query. Because most retrieval queries affect more than one entity, this entry point is usually located in a loop. Conversely, a single processing of an update cursor alters all the affected entities.

There is one parameter, cursor_info, associated with the SIM_PROCESS_CURSOR entry point. For more information on the SIM_PROCESS_CURSOR entry point, refer to "SIM_PROCESS_CURSOR" in Section 8, "Describing the SIM Library Entry Points."

Retrieving Query Output

A query is composed of three layers:

- Query

  Contains global information.

- Template

  Contains entity information.

- Item

  Contains attribute information.

You can choose to retrieve the output from your query in either of the following ways:

- One item at a time by using the SIM_GET_VALUE entry point
- One template at a time by using the SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA entry point

These two methods are described in the following text. For more information about the structure of a query, refer to “Gathering Information about a Query” later in this section.

Retrieving Query Output One Item at a Time

To output the data associated with each entity identified by your retrieval query, you must follow each call you make to SIM using the SIM_PROCESS_CURSOR entry point with one or more calls using the SIM_GET_VALUE entry point.

The following four parameters are associated with the SIM_GET_VALUE entry point:

- cursor_info

  Identifies the query.
- item_info

  Identifies the item in the query target list about which you want information.
- item_value

  Returns the value of the designated item as a string.
- numeric_value

  Returns the value of the designated item as a number.

There is only one parameter—cursor_info—associated with the SIM_PROCESS_CURSOR entry point.

For more information on the SIM_GET_VALUE and the SIM_PROCESS_CURSOR entry points, refer to Section 8, “Describing the SIM Library Entry Points.”

Retrieving Query Output One Template at a Time

To extract your query output one template at a time, use the SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA entry point. The following two parameters are associated with this entry point:

- cursor_info

  Identifies the query.
- query_buffer

  Provides the information for the template.

The information associated with the query_buffer parameter is returned as a block of data which your program is responsible for interpreting. You can determine the layout of the information from the QDUMP file produced by using the QD_CONTROLLER entry point.

For information on the QD_CONTROLLER entry point and the QDUMP file, refer to "Returning Statistical Information" later in this section.

Using the SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA entry point is useful under either of the following circumstances:

- When you know the query ahead of time
- If you are processing a large number of records, and each record contains several items

If you do not know the query ahead of time, your program must be able to decode the template record layout dynamically. To accomplish this you must understand the layout of the query description QUERY_DESC record. This record contains information about item offsets and types and their associated templates. You can obtain a copy of the QUERY_DESC record for a query by using the entry point SIM_PARSE_FORMATTED_QUERY instead of SIM_PARSE_QUERY. SIM_PARSE_FORMATTED_QUERY has the parameter, query_desc, which is filled with the QUERY_DESC record for the parsed query. Refer to the following for more information on the QUERY_DESC record:

- The example program EXAMPLE/SIM/ALGOL/API/RECORDATATIME on the System Software Release (SSR) media for the procedures you require to unpack a template record using the information in the QUERY_DESC record
- "Using the QUERY_DESC Record" later in this section

Example

Assume you are processing the following query:

```
FROM EMPLOYEE
RETRIEVE STRUCTURE NAME, BIRTH_DATE, (NAME, AGE) OF PARENT
WHERE TRUE
```

This query contains the following two templates:

- A template including the compound attribute NAME and its components FIRST_NAME, MID_INITIAL, and LAST_NAME
- A template including the compound attribute NAME, the components of NAME, and the attribute BIRTHDATE

The information provided by the QDUMP file for the first template is represented by Table 7–2. The information for the second template is represented by Table 7–3.

Table 7–2. Layout of First Template

Type	Offset: D: b	Length	Nulls	Target	Name
String	6:0:0	15	0	1	FIRST_NAME OF NAME String type = EBCDIC
String	24:0:0	1	1	1	MID_INITIAL OF NAME String type = EBCDIC
String	30:0:0	20	2	1	LAST_NAME OF NAME String type = EBCDIC
String	54:0:0	10	3	2	BIRTH_DATE String type = EBCDIC

Table 7–3. Layout of Second Template

Type	Offset: D: b	Length	Nulls	Target	Name
String	72:0:0	15	132	3	FIRST_NAME OF NAME OF PARENT String type = EBCDIC
String	90:0:0	1	133	3	MID_INITIAL OF NAME OF PARENT String type = EBCDIC
String	96:0:0	20	134	3	LAST_NAME OF NAME OF PARENT String type = EBCDIC
Integer	120:0:0	48,00	135	4	AGE OF PARENT

The item offsets in the QDUMP file are given as a triple in the form Offset: D: b. The byte offset within the digit is represented by Offset, the digit offset within the byte is represented by D, and the bit offset within the digit is represented by B.

Each template record also contains null Boolean values for each item in the record. The null Boolean values are packed into the beginning of each record, with each Boolean value occupying one digit. The null Boolean values occur in the same order as the corresponding template items and act as presence bits; that is, a true value indicates that the corresponding item has a value.

You can calculate the total record length by adding the length of the last item to the offset of the last item. You must ensure that the size of the query_buffer record you pass to the SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA entry point is large enough to accommodate the maximum record size.

When you are retrieving data from a query that involves multiple templates, you can use the field IC_TEMPLATE_NO in the CURSOR_INFO record to determine which template number was returned by the last cursor call.

The following entry points process an instance of a query, retrieve a record of information, and return the information as a block of data:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PROCESS_CURSOR_AND RETRIEVE_DATA
- SIM_PROCESS_CURSOR_AND_UPDATE_DATA

The returned data is manipulated at the bit level, which is extremely efficient in ALGOL, but not in COBOL. Therefore, the preceding three entry points are not available in COBOL. If you are using COBOL to process an instance of a query and retrieve a record of information, use one of the following COBOL entry points:

- SCIM_PARSE_QUERY
- SCIM_PARSE_SQL
- SCIM_PROCESS_CURSOR
- SCIM_PUT_VALUE
- SCIM_GET_VALUE

Processing an Update Query

Once you have compiled your update query and assigned values to any query parameters, you are ready to process the query. You use the SIM_PROCESS_CURSOR entry point to process the query. There is one parameter—cursor_info—associated with the SIM_PROCESS_CURSOR entry point. The cursor_info parameter identifies the instance of the query you want to process. For more information on using the SIM_PROCESS_CURSOR entry point, refer to “SIM_PROCESS_CURSOR” in Section 8, “Describing the SIM Library Entry Points.”

Deallocating a Cursor from a Query

Use the SIM_CLOSE_CURSOR entry point to deallocate a cursor from a query. The one parameter associated with this entry point is cursor_info. The cursor_info parameter identifies the query cursor and it should be the same record you use when you use the SIM_OPEN_CURSOR and SIM_PROCESS_CURSOR entry points.

For retrieval queries, the SIM_CLOSE_CURSOR entry point should be used after you have obtained all the desired information for that instance of the query. Typically, this is after you have located all the tuples for the query by using the SIM_PROCESS_CURSOR entry point.

Once you have closed a cursor, you can use the CURSOR_INFO records associated with the cursor_info parameter for a different query or for another instance of the same query.

At this point, if you performed your query in transaction state, you should end the transaction by using either the SIM_ETR or the SIM_COMS_ETR entry point.

For more detailed information on the SIM_CLOSE_CURSOR, refer to Section 8, “Describing the SIM Library Entry Points.”

Closing a Query

Use the SIM_CLOSE_QUERY entry point to close the query. The query_info parameter is the only parameter associated with this entry point. The query_info parameter should be the same QUERY_INFO record you use when you parse the query using any of the query parsing entry points.

Closing a query automatically closes all open cursors for that query.

For more information on the SIM_CLOSE_QUERY entry point, refer to “SIM_CLOSE_QUERY” in Section 8, “Describing the SIM Library Entry Points.”

Closing a Database

Once you have finished with a database, use the SIM_CLOSE_DATABASE entry point to close the database. You can improve system performance by keeping the number of open databases to a minimum. The only parameter associated with this entry point is dbname_text, which is defined as a string. The dbname_text parameter contains the database name.

For more information on the SIM_CLOSE_DATABASE entry point, refer to Section 8, “Describing the SIM Library Entry Points.”

Returning Error Messages

The errors returned by SIM can occur while you are compiling a query or they can occur while you are processing a query. For example, incorrectly qualifying an attribute is a compile-time error, while issuing a begin-transaction request when you are already in transaction state is a run-time error.

Appendix A, "OML Error Messages," lists and explains the OML errors you might receive. Appendix B, "Exception Fields and Categories," lists and explains the exceptions you might receive when you compile and run your program. There are three types of exceptions: warnings, completions, and errors. Associated with each exception is a category, subcategory, and text.

Use the following four entry points to check for exception message information:

- SIM_NEXT_EXCEPTION
- SIM_GET_EXCEPTION_INFO
- SIM_EXCEPTION_INFO
- SIM_EXCEPTION_MESSAGE

The SIM_NEXT_EXCEPTION entry point returns a Boolean value that indicates whether more exception messages are available. If more exception messages are available, the next exception message becomes the current exception message. The information you can extract using the SIM_GET_EXCEPTION_INFO, the SIM_EXCEPTION_INFO, and the SIM_EXCEPTION_MESSAGE entry points always relates to the current exception message.

The Boolean value returned by the SIM_NEXT_EXCEPTION entry point is a 48-bit word that contains the category and subcategory numbers associated with the exception. To extract the category and subcategory information, you can break down the 48-bit word by using the partial word fields EX_CATEGORYF and EX_SUBCATEGORYF. Appendix B, "Exception Fields and Categories," lists and explains the category and subcategory information that can be returned. There are no parameters associated with the SIM_NEXT_EXCEPTION entry point.

As an alternative to extracting the partial fields associated with the SIM_NEXT_EXCEPTION entry point, you can use the SIM_GET_EXCEPTION_INFO entry point. The two parameters associated with the SIM_GET_EXCEPTION_INFO entry point provide the category and subcategory of the error. If there are additional exception messages to be returned, there is a true value associated with the SIM_GET_EXCEPTION_INFO entry point. However, before you can use the SIM_GET_EXCEPTION_INFO entry point to extract the next exception message information, you must use the SIM_NEXT_EXCEPTION entry point to make the next exception message the current exception message.

Once you have the category and subcategory associated with the exception, you can choose to do either or both of the following:

- Extract the text associated with the exception using the SIM_EXCEPTION_MESSAGE entry point.

  There are two parameters—lang and message_array—associated with this entry point. The parameter lang is a string that enables you to designate the language in which you want the error message returned. The message_array parameter contains the text of the exception.

  In addition, there is an integer result associated with the SIM_EXCEPTION_MESSAGE entry point. Under normal conditions, this result is the value 0 (zero). However, if an error occurs while you are retrieving the text for the current error, the result of the SIM_EXCEPTION_MESSAGE entry point is nonzero. To determine the meaning of the result, refer to the section in the appropriate language manual that explains calls to the MESSAGESEARCHER procedure.

- Extract more details about the exception using the SIM_EXCEPTION_INFO entry point.

  Using the SIM_EXCEPTION_INFO entry point enables you to retrieve information about such things as CENTRALSUPPORT error messages.

  There is one parameter—exception_info—associated with the SIM_EXCEPTION_INFO entry point. The exception_info parameter is a record and contains information such as the name of the database involved in the exception. You can also obtain other information, such as DMSII DMSTATUS information and CENTRALSUPPORT error codes.

The following examples illustrate the use of the error message entry points. For detailed information on each of the error message entry points, refer to Section 8, “Describing the SIM Library Entry Points.”

PASS_ARRAY is a define used when passing array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” earlier in this section for an explanation of this define.

Example 1

The following procedure checks for CENTRALSUPPORT exceptions. If a CENTRALSUPPORT exception occurs, the program extracts the text associated with this exception instead of extracting the text associated with the SIM exception.

```
PROCEDURE PROCESS_CENTRALSUPPORT_ERROR;
BEGIN
    ARRAY EXCEPT_INFO [0:EI_SIZE];
    BOOLEAN RESULT;
    INTEGER
        ERROR_CATEGORY
      ,ERROR_SUBCATEGORY
     ;
    DEFINE
        BAIL_OUT =
            IF EX_CATEGORY (RESULT) NEQ EX_NONE AND
               EX_CATEGORY (RESULT) NEQ EX_COMPLETE THEN
                BEGIN
                DISPLAY ("CATEGORY = "!!
                         STRING (EX_CATEGORY (RESULT), *)!!
                         "SUBCATEGORY = "!!
                         STRING (EX_SUBCATEGORY (RESULT), *));
                END #;

    IF RESULT:= SIM_GET_EXCEPTION_INFO (PASS_ARRAY (ERROR_CATEGORY)
                                 ,PASS_ARRAY (ERROR_SUBCATEGORY)
                                ) THEN
        BEGIN
        DO BEGIN
           IF RESULT:= SIM_EXCEPTION_INFO
                          (PASS_ARRAY (EXCEPT_INFO)) THEN
               BEGIN
               % Some kind of error occurred.
               BAIL_OUT;
               END
           ELSE
               BEGIN
               IF EI_INTL_PROCRSLT (EXCEPT_INFO) THEN
                  BEGIN
                   % At this point call the CENTRALSUPPORT library
                   % entry point to obtain the error message text.
                  END;
               END;
           END
        UNTIL NOT RESULT:= SIM_NEXT_EXCEPTION;

        END;
END;
```

Example 2

In this example, the message supplied by SIM is extracted. In this instance the first display by the procedure is a general SIM error “An error occurred in calling CENTRALSUPPORT.” The SIM_NEXT_EXCEPTION entry point call moves the CENTRALSUPPORT error information to the top of the error stack. On the second call to the SIM_EXCEPTION_MESSAGE entry point, the CENTRALSUPPORT error text that SIM obtains is displayed.

PASS_ARRAY is a define used when passing array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” earlier in this section for an explanation of this define.

```
PROCEDURE PROCESS_CENTRALSUPPORT_ERROR;
BEGIN
    ARRAY
        EXCEPT_INFO [0:EI_SIZE]
      ,LANG
      ,MESSAGE_ARRAY [0:12]
      ;
    BOOLEAN RESULT;
    INTEGER
        ERROR_CATEGORY
      ,ERROR_SUBCATEGORY
      ,MLS_RESULT
      ;
    DEFINE
        BAIL_OUT =
              IF EX_CATEGORY (RESULT) NEQ EX_NONE AND
                 EX_CATEGORY (RESULT) NEQ EX_COMPLETE THEN
                  BEGIN
                  DISPLAY ("CATEGORY = "!!
                           STRING (EX_CATEGORY (RESULT), *)!!
                           "SUBCATEGORY = "!!
                           STRING (EX_SUBCATEGORY (RESULT), *));
                  END #;

    IF RESULT:=SIM_GET_EXCEPTION_INFO (PASS_ARRAY (ERROR_CATEGORY)
                                   ,PASS_ARRAY (ERROR_SUBCATEGORY)
                                  ) THEN
    ELSE
    IF ERROR_CATEGORY = EX_FAILED AND
       ERROR_SUBCATEGORY = EX_CENTRALSUPPORT THEN
        BEGIN
        % We just want to look at CENTRALSUPPORT errors.

        % Set up language.
        REPLACE POINTER (LANG [1]) BY "ENGLISH";
        LANG [0]:= 7;

        DO BEGIN
           MLS_RESULT:= SIM_EXCEPTION_MESSAGE
                                 (PASS_ARRAY (LANG)
                           ,PASS_ARRAY (MESSAGE_ARRAY)
                          );
           IF MLS_RESULT LSS 0 THEN
               BEGIN
               % Some kind of error occurred.
               DISPLAY ("MLS RESULT = "!! STRING (MLS_RESULT, *));
               END
           ELSE
               BEGIN
               % Here I have the error message text supplied by SIM.
               DISPLAY (STRING (POINTER (MESSAGE_ARRAY [1]),
                        MESSAGE_ARRAY [0]));
               END;
           END
        UNTIL NOT RESULT:= SIM_NEXT_EXCEPTION;

END;
```

Example 3

The following example demonstrates the use of the SIM_NEXT_EXCEPTION, the SIM_GET_EXCEPTION_INFO, and the SIM_EXCEPTION_MESSAGE entry points.

The example is intended to be called after an error has occurred. However, if no errors occur, SIM returns a code indicating that there are no errors.

PASS_ARRAY is a define used when passing array parameters. Refer to "Passing Arrays to SIM from ALGOL Programs" earlier in this section for an explanation of this define.

```
PROCEDURE PROCESS_ERROR;
    BEGIN
    BOOLEAN
        RESULT;
    INTEGER
        MLS_RSLT,
        ERROR_CATEGORY,
        ERROR_SUBCATEGORY;
    ARRAY
        LANGUAGE  [0:5],
        MESSAGE   [0:30];
    LABEL
        END_OF_PROCEDURE;

    %  Set up the LANGUAGE parameter so that the error messages
    %  are returned in the language of your choice, in this case,
    %  ENGLISH.

    REPLACE LANGUAGE[1] BY "ENGLISH";
    LANGUAGE[0]:= 7;

    RESULT:= SIM_GET_EXCEPTION_INFO (ERROR_CATEGORY,
                                     ,ERROR_SUBCATEGORY
                                    );
    IF ERROR_CATEGORY = EX_NONE THEN
        BEGIN

             %  This is the result that signals that no errors exist.
             %  SIM can return an appropriate error message if
             %  requested; however, this example issues its own
             %  message.

        DISPLAY("No errors are present.");
        END
    ELSE
    IF ERROR_CATEGORY = EX_COMPLETE AND
       ERROR_SUBCATEGORY = EX_CMPL_NO_MORE THEN
        BEGIN

        %  This is the result that signals that a query has
        %  completed and there is no more data to be returned.
    %  SIM can return an appropriate error message if
    %  requested; however, this example issues its own
    %  message.

    DISPLAY("Query completed");
    END
ELSE
    BEGIN
    DO
        BEGIN

        %  The next two statements show an alternative to
        %  calling the SIM_GET_EXCEPTION_INFO entry point to
        %  get the exception category and subcategory
        %  information.

        ERROR_CATEGORY:= EX_CATEGORY (RESULT);
        ERROR_SUBCATEGORY:= EX_SUBCATEGORY (RESULT);

        MLS_RSLT:= SIM_EXCEPTION_MESSAGE (PASS_ARRAY(LANGUAGE)
                                         ,PASS_ARRAY(MESSAGE)
                                         );
        IF MLS_RSLT LSS 0 THEN
            BEGIN
            DISPLAY ("An error occurred while retrieving "!!
                      "the message text from SIM; "!!
                      "MLS result = "!! STRING (MLS_RSLT, *)
                     );
            END
        ELSE
            BEGIN

            %  The message is in the MESSAGE array.  The first
            %  word contains the length of the message. The message
            %  begins in the second word.
            %
            %  If the SIM_EXCEPTION_MESSAGE entry point returns the
            %  value 1, the exception message could not be
            %  translated because the requested language could not
            %  be found. Instead, the message is displayed
            %  in the user's default language.

            DISPLAY (STRING(POINTER(MESSAGE[1]), MESSAGE[0]));
            END;
        END
    UNTIL NOT RESULT:= SIM_NEXT_EXCEPTION;
    END;
END;
```

Saving and Restoring Queries

You can use the entry points SIM_SAVE_QUERY, SIM_LOAD_QUERY, and SIM_ADD_QUERY to save and reuse query information between runs of a program. Using these entry points, you can use one run of a program to parse the queries and then request that SIM save the query information in a file. Subsequently, your program or another program can then have the option of either parsing the queries itself or requesting that SIM load the query information from the saved file. Loading queries from a saved file requires less time and resources than reparsing the queries. For programs with a large number of queries, the savings can be significant.

Saving Queries

Use the SIM_SAVE_QUERY entry point to save queries in a file. There is one parameter associated with this entry point—file_title. The file_title parameter is a string which designates the file into which you want to save the query information. The file title can be any valid A Series file title.

When you call the SIM_SAVE_QUERY entry point, all queries that are currently open are saved. A query is considered to be open if the text has been parsed without errors and the query has never been closed (by calling the SIM_CLOSE_QUERY entry point). The existence of open cursors for a query is not used to determine if a query is open or closed.

Restoring Queries

You use the SIM_ADD_QUERY and the SIM_LOAD_QUERY entry points to restore query information from a saved file. The entry point that you need to use depends on whether a database is open when you call the entry point.

Both entry points use the following two parameters:

- file_title

  The file_title parameter is a string that designates the file containing the saved query information. The first word of the parameter must contain the length of the file title. The file title can be any valid A Series file title.

- query_info

  The query_info parameter is a record that contains the query information that was in the designated file. Word 0 (zero) of the QUERY_INFO record contains the number of QUERY_INFO records.

Note: *You must insure that the array you declare for the query_info parameter is large enough to contain all information for all queries in the saved file. If the array is too small, an error is returned.*

In your program, you must examine the PO_USER_QUERY_NO, the PO_ID_LENGTH, and the PO_ID fields of each query identifier record to determine the significance of each query. For more information, refer to “Using the QUERY_INFO Record” later in this section.

You can use the SIM_LOAD_QUERY entry point either before you open any database, or after you have closed all the databases you had opened. If you use the SIM_LOAD_QUERY entry point while you still have a database open, an error is returned.

Note: *Using the SIM_LOAD_QUERY entry point invalidates all the query information that might have existed before you used the SIM_LOAD_QUERY entry point.*

If you already have a database open, then use the SIM _ADD_QUERY entry point to load a query. Using the SIM_ADD_QUERY entry point appends all query identifier records to the end of the list of existing open queries.

For more information on the SIM_ADD_QUERY and SIM_LOAD_QUERY entry points, refer to “SIM_ADD_QUERY” and “SIM_LOAD_QUERY” in Section 8, “Describing the SIM Library Entry Points.”

Examples Illustrating Saving and Restoring Queries from Files

The following ALGOL procedures illustrate how you might use the SIM_LOAD_QUERY, SIM_SAVE_QUERY, and SIM_ADD_QUERY entry points.

Example 1

The following ALGOL procedure receives three parameters—qtext, query_id, and query_no. The qtext and query_id parameters are defined as strings. Word 0 (zero) of both strings contains the length of the string associated with the parameter. The query_no parameter is defined as an integer. The purpose of each parameter is described in the following text:

- Qtext contains the user-supplied query text.
- Query_id contains the user-supplied query identification text.
- Query_no contains the user-supplied query identification number.

The procedure parses the query text and then marks the query information record with an identifier for later use. The procedure returns a true result if any problems occur when the query is parsed.

```
BOOLEAN PROCEDURE PARSE_AND_ID (QTEXT, QUERY_NO, QUERY_ID);
      VALUE QUERY_NO, QUERY_ID;
      INTEGER QUERY_NO;
      ARRAY QTEXT, QUERY_ID [0];
  BEGIN
      ARRAY QUERY_INFO [0:PO_SIZE];
      BOOLEAN RESULT;              % Holds result from SIM library
                                   % call.
      % Initialize query_info.
      REPLACE POINTER (QUERY_INFO) BY
          48"000000" FOR SIZE (QUERY_INFO) WORDS;

      % Mark the query with a user-specified identifier.
      PO_USER_QUERY_NO (QUERY_INFO):= QUERY_NO;
      REPLACE PO_ID (QUERY_INFO) BY
          POINTER (QUERY_ID [1]) FOR QUERY_ID [0];
      PO_ID_LENGTH (QUERY_INFO):= QUERY_IND [0];
      IF RESULT:= SIM_PARSE_QUERY (PASS_ARRAY (QUERY_INFO)
                                   PASS_ARRAY (QTEXT)) THEN
          BEGIN
          % Some kind of error occurred.
          PROCESS_ERROR;
          END;

      % Return procedure result.
      PARSE_AND_ID:= RESULT;

  END OF PARSE_AND_ID;
```

Example 2

The following ALGOL procedure receives three parameters—query_info, qstack, and qcount. The parameter query_info is a record that is returned when either the SIM_ADD_QUERY or the SIM_LOAD_QUERY entry point is called. The qstack parameter is a two-dimensional array that contains the parsed queries obtained from the QUERY_INFO record. The qcount parameter is an integer that maintains a count of the queries that are currently associated with the qstack parameter.

The procedure traverses the QUERY_INFO record, placing specific parsed queries into the qstack array for later use. Only those queries with a value for the PO_USER_QUERY_NO item greater than 0 (zero) are restored to the qstack array.

```
PROCEDURE DECOMPOSE_QUERY_INFO (QUERY_INFO, QSTACK, QCOUNT);
      INTEGER QCOUNT;
      ARRAY QUERY_INFO [0],
            QSTACK [0,0];
  BEGIN
      INTEGER
          NUM_OF_QUERIES          % Count of queries in query_info.
         ,STACK_SIZE              % Number of rows in qstack.
         ,RESTORE_I               % Row index to start putting QID.
         ,QUERY_COUNT             % Count of restored queries.
         ,QUERY_NO                % Query number in query info.
        ;

      % Get number of saved queries in query info.
      NUM_OF_QUERIES:= QUERY_INFO [0];

      % Check to see if query_stack is large enough to hold
      % queries to be restored.
      STACK_SIZE:= SIZE (QSTACK);
      IF STACK_SIZE LSS (STACK_SIZE + NUM_OF_QUERIES) THEN
          RESIZE (QSTACK,STACK_SIZE+NUM_OF_QUERIES,RETAIN);

      % Row index to start restoring queries into qstack.
      RESTORE_INX:= QCOUNT + 1;
      QUERY_COUNT:= QUERY_NO:= 0;
      FOR QUERY_NO:= 1 STEP 1 WHILE QUERY_COUNT < NUM_OF_QUERIES DO
          BEGIN
          ARRAY QID [-(QUERY_NO*PO_SIZE)] = QUERY_INFO;
          IF PO_QUERY_NO (QID) > 0 THEN
              BEGIN
              REPLACE POINTER (QSTACK [RESTORE_INX,0]) BY
                  POINTER (QID [0]) FOR PO_SIZE WORDS;
              RESTORE_INX:= * + 1;
              QUERY_COUNT:= * + 1;
              END;
          END;
      % Return number of restored queries.
      QCOUNT:= * + QUERY_COUNT;
  END OF DECOMPOSE_QUERY_INFO;
```

Example 3

In Example 1, the procedure parsed a query and then associated an identifier with the query. In this example, all currently open, parsed queries are saved. The procedure receives one parameter—file_title. The parameter is a string with word 0 (zero) containing the length of the file title.

The procedure uses the SIM_SAVE_QUERY entry point to save the query information. The procedure returns a true result if any problems occur while the queries are being saved.

```
BOOLEAN PROCEDURE SAVE_QUERIES (FILE_TITLE);
     ARRAY FILE_TITLE [0];
  BEGIN
     BOOLEAN RESULT;          % Holds result from SIM library call.

  IF RESULT:= SIM_SAVE_QUERY (PASS_ARRAY (FILE_TITLE)) THEN
         BEGIN
         % Some kind of error occurred.
         PROCESS_ERROR;
         END;

     % Return procedure result.
     SAVE_QUERIES:= RESULT;

  END OF SAVE_QUERIES;
```

Example 4

Example 3 illustrates how to save queries. This example illustrates how to add and load queries from a file. The procedure receives three parameters—add_queries, file_title, and query_info.

The parameter add_queries is a Boolean value. If the add_queries parameter has a true value, the SIM_ADD_QUERY entry point is used. If the add_queries parameter has a false value, the SIM_LOAD_QUERY entry point is used.

The parameter file_title is a string with word 0 (zero) containing the length of the file title.

The parameter query_info is a record that holds all the stored query information in the file.

The procedure returns a true result if there are any problems while trying to restore the saved queries.

```
PROCEDURE ADD_OR_LOAD_QUERIES
      (ADD_OR_LOAD, FILE_TITLE, QUERY_INFO);
      VALUE ADD_OR_LOAD;
      BOOLEAN ADD_OR_LOAD;
      ARRAY FILE_TITLE, QUERY_INFO [0];
  BEGIN
      BOOLEAN
          RESULT            % Holds result from SIM library call.
        ,DONE               % For loop control.
        ,ADD_LOAD_ERROR     % Error restoring queries.
       ;

      % Initialize loop control and query_info.
      DONE:= FALSE;
      REPLACE POINTER(QUERY_INFO) BY
          48"000000" FOR SIZE (QUERY_INFO);

      WHILE NOT DONE DO
          BEGIN
          IF ADD_QUERIES THEN       % Use SIM_ADD_QUERY entry point.
              RESULT:= SIM_ADD_QUERY (PASS_ARRAY (FILE_TITLE)
                                     ,PASS_ARRAY (QUERY_INFO)
                                    )
          ELSE
              RESULT:= SIM_LOAD_QUERY (PASS_ARRAY (FILE_TITLE)
                                      ,PASS_ARRAY (QUERY_INFO)
                                     );

          % Check SIM library entry point call result.
          IF RESULT THEN
              BEGIN
              IF EX_CATEGORY (RESULT) = EX_FAILED AND
                 EX_SUBCATEGORY (RESULT) = EX_FAIL_DB_OPEN THEN
                  BEGIN
                  % Unable to restore queries when a database
```

```
                % is open using the SIM_LOAD_QUERY entry point.
                PROCESS_ERROR;
                ADD_LOAD_ERROR:= DONE:= TRUE;
                END
            ELSE
            IF EX_CATEGORY (RESULT) = EX_FAILED AND
               EX_SUBCATEGORY (RESULT) =
               EX_FAIL_QID_INFO_TOO_SMALL THEN
                BEGIN
                % Make query_info record larger. Please note
                % that in this example the incremental size is
                % for one more query (PO_SIZE). In your
                % application it might be advantageous to multiply
                % PO_SIZE by some amount to lower the amount
                % of resizes needed, if any.
                RESIZE (QUERY_INFO, SIZE(QUERY_INFO)+PO_SIZE,
                   DISCARD);
                END
            ELSE
               BEGIN
               % Some kind of error occurred.
               PROCESS_ERROR;
               ADD_LOAD_ERROR:= DONE:= TRUE;
               END;
            END
        ELSE
            DONE:= TRUE;
        END;

    IF ADD_LOAD_ERROR THEN
        ADD_OR_LOAD_QUERIES:= RESULT;

END OF ADD_OR_LOAD_QUERIES;
```

Gathering Information about a Query

A query is composed of three layers:

- Query
- Template
- Item

The *query layer* contains information at a global level, for example, whether tabular or structured output is required and how many template levels are involved in the query.

The *template layer* contains information at an entity level, for example, the number of hierarchical levels of the database involved in the query and the attributes required.

The *item layer* returns information at the attribute level, for example, the type, length, and precision of data values.

Example

Figure 7–1 illustrates the template and item layers for the following query. The query has three template levels. The first template level has one template with two items. The second template level has two templates, both with one item. The third template level has one template with two items.

```
FROM PROJECT_EMPLOYEE
RETRIEVE STRUCTURE
         EMPLOYEE_ID, AVG(EMPLOYEE_SALARY),
         EMPLOYEE_SALARY OF EMPLOYEE_MANAGER,
         DEPT_TITLE OF DEPT_IN,
         PROJECT_TITLE OF PROJECTS_MANAGING OF EMPLOYEE_MANAGER,
         PROJECT_NO OF PROJECTS_MANAGING OF EMPLOYEE_MANAGER
```

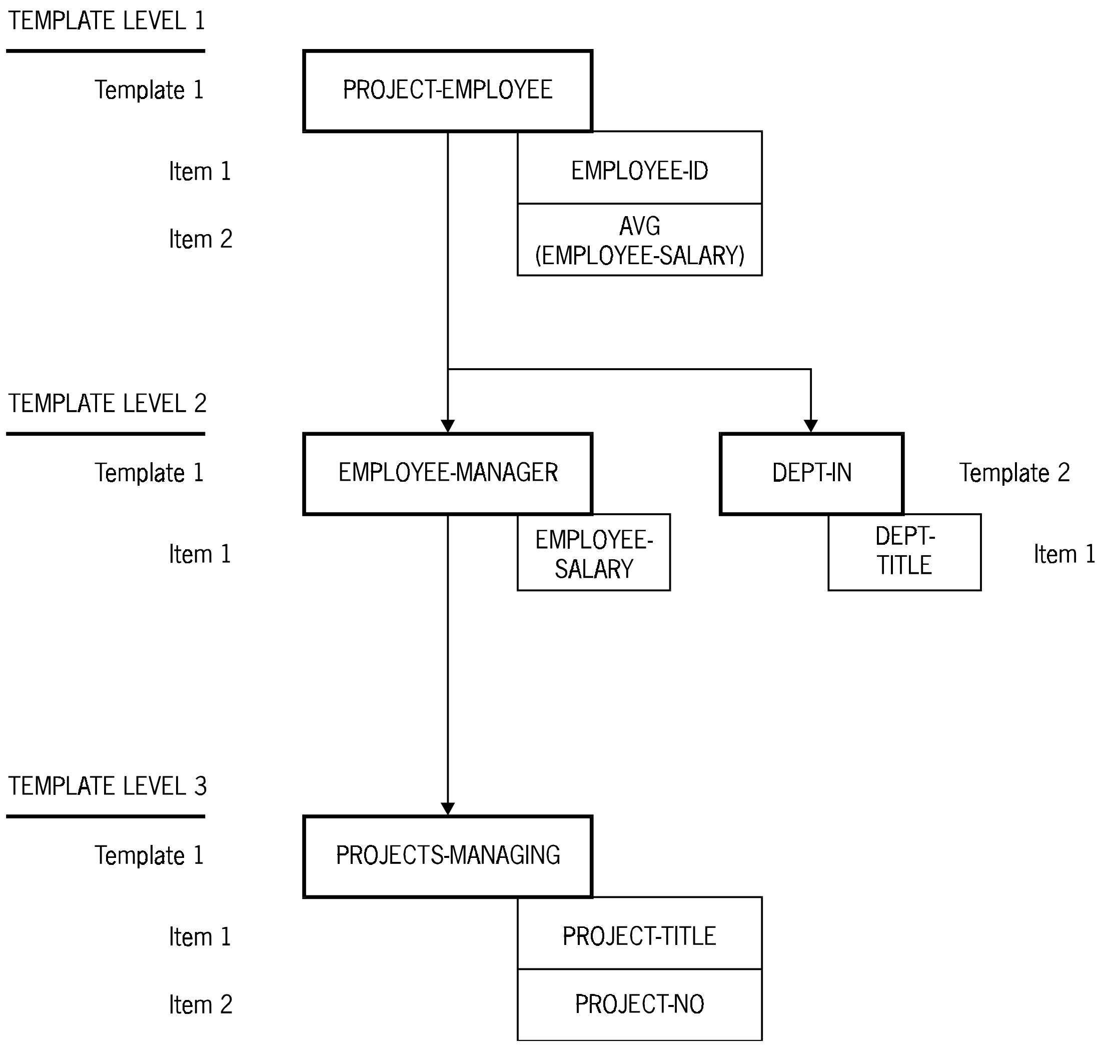


Figure 7–1. Template and Item Layers of a Structured Query

Structured queries might have more than one template level. Also, the numbering of the items need not match the order in which the items are listed in your query because the item numbering for each template level starts at one.

Tabular queries can have only one template level. Also, the numbering of the items matches the order in which the items are listed in your query.

To gather information about a query at the query, template, and item levels, use the following entry points:

- SIM_DESCRIBE_QUERY
- SIM_DESCRIBE_TEMPLATE
- SIM_DESCRIBE_ITEM

If your program includes the query text, then you are not likely to use the SIM_DESCRIBE_QUERY entry point. However, if your program accepts query text at run time, then you need to use the SIM_DESCRIBE_QUERY entry point, along with the SIM_DESCRIBE_TEMPLATE and SIM_DESCRIBE_ITEM entry points, to provide your program with enough information so that it can perform the requested action.

For example, some queries might include parameters. In these cases, your program must supply information for the parameters at some point between the call to the SIM_OPEN_CURSOR entry point and the first call to the SIM_PROCESS_CURSOR entry point.

The following information describes how to access the text of a query, and it also describes the SIM_DESCRIBE_QUERY, SIM_DESCRIBE_TEMPLATE, and SIM_DESCRIBE_ITEM entry points.

An example illustrating the use of the SIM_DESCRIBE_QUERY, SIM_DESCRIBE_TEMPLATE, and SIM_DESCRIBE_ITEM entry points is provided under “Example Illustrating How to Gather Query Information” later in this section.

Writing Out the Query Text

You can use the SIM_DESCRIBE_DML_TEXT entry point to place the text of a query into a string.

The following two parameters are associated with this entry point:

- query_info

  The query_info parameter is a record that identifies the query.

- qtext

  The qtext parameter is the string into which the query text is to be placed.

Retrieving Query Level Information

You use the SIM_DESCRIBE_QUERY entry point to retrieve basic information about a parsed query. This information includes such things as the query type (insert, modify, retrieval, or delete) and output format (tabular or structured).

The following two parameters are associated with the SIM_DESCRIBE_QUERY entry point:

- query_info

  The query_info parameter identifies the query about which information is being requested. The query_info parameter is the same record area as that passed to any of the query parsing entry points.

- query_record

  The query_record parameter is a record into which SIM places information about the query. The information returned includes such things as the type of query (insert, modify, retrieval, or delete), whether the output is tabular or structured, the number of templates (for structured output only), and if parameter values are required.

For more information on using SIM_DESCRIBE_QUERY, refer to "SIM_DESCRIBE_QUERY" in Section 8, "Describing the SIM Library Entry Points."

Retrieving Template Level Information

The data retrieved by a query is held internally by SIM in records. These records are described by templates. Each template describes exactly one record.

The following two parameters are associated with the SIM_DESCRIBE_TEMPLATE entry point:

- query_info

  Identifies the query about which you want information. The query_info parameter is the same record area as the one you pass to any of the query parsing entry points.

- template_info

  Returns information about a particular template produced by your query.

For tabular queries, only one record is used. In this case, the most useful piece of information returned by the SIM_DESCRIBE_TEMPLATE entry point is the number of attributes this record contains. The number of attributes in the record is the number of attributes retrieved by the query.

Structured queries can produce different records for the information retrieved at different levels of the query. The records at each level are described by a different template. Furthermore, a single record might have data returned in multiple templates. The total number of attributes retrieved by a structured query is determined by totaling the number of attributes described in each template.

Note: *When retrieving data for a query with multiple templates, it is essential that the program examine the TEMPLATE_NO field in the CURSOR_INFO record associated with the cursor_info parameter. You use the cursor_info parameter when making calls to the SIM_PROCESS_CURSOR entry point. The value of the TEMPLATE_NO field indicates which template describes the data being returned by SIM for the current cursor.*

For more information on using the SIM_DESCRIBE_TEMPLATE entry point, refer to “SIM_DESCRIBE_TEMPLATE” in Section 8, “Describing the SIM Library Entry Points.”

Retrieving Item Level Information

Each data template produced by a query can contain values for many attributes. You use the SIM_DESCRIBE_ITEM entry point to return information about designated attributes.

The following parameters are associated with the SIM_DESCRIBE_ITEM entry point:

- query_info

  Identifies the query about which you want information. The query_info parameter is the same record area as the one you pass to any of the query parsing entry points.
- item_info

  Describes one of the attributes retrieved by a query. This parameter is a record.
- item_name

  Contains the name of the item described by the information in the item_info parameter. The item_name parameter is a string. The first word of the string contains the length of the item name.

Assuming that the query is as described in Figure 7–1, the SIM_DESCRIBE_ITEM entry point returns the following information if you request information for the first item in template 3:

Information	Value
Item type	String
Item sign	False
Item length	20 characters
Item precision	0
Item position in the target list	4
Item name	DEPT_TITLE OF DEPT_IN OF PROJECT_EMPLOYEE
Item ccsversion	0
Item database number	An integer that identifies which currently open database the item is a part of
Item internal identification	An integer that uniquely identifies the entity in the schema. If the item is not an entity in the schema, the value of the item internal identification is 0 (zero)

Note: *The ccsversion is assumed to be ASeriesNative, which is identified by the ccsversion number 0.*

For more information on using the SIM_DESCRIBE_ITEM entry point, refer to “SIM_DESCRIBE_ITEM” in Section 8, “Describing the SIM Library Entry Points.”

Example Illustrating How to Gather Query Information

Example 7–3 demonstrates the use of the SIM_DESCRIBE_QUERY, SIM_DESCRIBE_TEMPLATE, and SIM_DESCRIBE_ITEM entry points. The example program can be used with any query text.

The example assumes the query text has already been parsed. If an error occurs, the procedure PROCESS_ERROR is called. For more information on error processing, refer to “Returning Error Messages” earlier in this section. PASS_ARRAY is a define used when passing array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” earlier in this section for an explanation of this define.

```
ARRAY
    QUERY_INFO     [0:PO_SIZE];

PROCEDURE SIM_DESCRIBE_EXAMPLE;
    BEGIN
    INTEGER
        TEMPLATE_NO,
        ITEM_INDEX,
        NUMBER_OF_ITEMS,
        TOTAL_ITEMS;
    ARRAY
        TEMPLATE_INFO  [0: SIM_TEMPLATE_SIZE],
        ITEM_INFO      [0: SIM_ITEM_RECORD_SIZE],
        ITEM_NAME      [0:100],    %  Must allow for long strings
                                   %  for qualified attributes.
        QUERY_RECORD   [0: SIM_Q_SIZE];
    LABEL
        END_OF_PROCEDURE;

    IF SIM_DESCRIBE_QUERY (PASS_ARRAY (QUERY_INFO)
                          ,PASS_ARRAY (QUERY_RECORD)
                         ) THEN
        BEGIN
        DISPLAY ("An error occurred while calling "!!
                 "SIM_DESCRIBE_QUERY"
                );
        PROCESS_ERROR;
        GO END_OF_PROCEDURE;
        END;
```

Example 7–3. Gathering Query Information (cont.)

```
CASE SIM_Q_STATEMENT (QUERY_RECORD) OF
    BEGIN
    (ST_RETRIEVE_V):
        IF SIM_Q_TABULAR (QUERY_RECORD) THEN
           DISPLAY("Query is a tabular RETRIEVE statement")
        ELSE
           DISPLAY("Query is a structured RETRIEVE statement");
    (ST_INSERT_V):
        DISPLAY("Query is an INSERT statement");
    (ST_MODIFY_V):
        DISPLAY("Query is a MODIFY statement");
    (ST_DELETE_V):
        DISPLAY("Query is a DELETE statement");
    END;  % CASE

IF SIM_Q_INPUT_REQUIRED (QUERY_RECORD) THEN
    DISPLAY ("This query has parameters");

IF SIM_Q_STATEMENT (QUERY_RECORD) = ST_RETRIEVE_V THEN
    BEGIN

    %  The code to look at templates and items is only useful
    %  for RETRIEVE queries.  Queries that update the database
    %  do not return data.

    DISPLAY ("This query returns data at "!!
             STRING(SIM_Q_LEVELS (QUERY_RECORD), *)!! " levels"
            );

    TEMPLATE_NO:= 1;
    SIM_QT_TEMPLATE_NUMBER (TEMPLATE_INFO):= TEMPLATE_NO;
    WHILE NOT SIM_DESCRIBE_TEMPLATE (PASS_ARRAY (QUERY_INFO)
                                    ,PASS_ARRAY (TEMPLATE_INFO)
                                   ) DO
        BEGIN
        DISPLAY("    Template Number: "!!
                STRING(TEMPLATE_NO, *)
               );
        DISPLAY("  Level: "!!
                STRING(SIM_QT_LEVEL_NUMBER (TEMPLATE_INFO), *)
                ":"
             );
        NUMBER_OF_ITEMS:= SIM_QT_ITEM_COUNT (TEMPLATE_INFO);
        TOTAL_ITEMS:= TOTAL_ITEMS + NUMBER_OF_ITEMS;
        DISPLAY("  Item count: "!!
                STRING(NUMBER_OF_ITEMS, *)
               );
```

Example 7–3. Gathering Query Information (cont.)

```
FOR ITEM_INDEX:= 1 STEP 1 UNTIL NUMBER_OF_ITEMS DO
    BEGIN

    %  The item for which information is desired is
    %  identified by item number and template number.

    SIM_ITEM_NUMBER (ITEM_INFO):= ITEM_INDEX;
    SIM_ITEM_TEMPLATE (ITEM_INFO):= TEMPLATE_NO;

    IF SIM_DESCRIBE_ITEM (PASS_ARRAY (QUERY_INFO)
                        ,PASS_ARRAY (ITEM_INFO)
                        ,PASS_ARRAY (ITEM_NAME)
                         ) THEN
        BEGIN
        DISPLAY ("An error occurred while calling "!!
                 "SIM_DESCRIBE_ITEM"
                );
        PROCESS_ERROR;
        GO END_OF_PROCEDURE;
        END;

    %  Display the item name first.

    DISPLAY ("    "!! STRING(POINTER(ITEM_NAME[1]
                            ,ITEM_NAME[0]
                           )
            );

    CASE SIM_ITEM_TYPE (ITEM_INFO) OF
        BEGIN
    (ITEM_IS_REAL_V):
        DISPLAY ("    Real");
    (ITEM_IS_INTEGER_V):
        BEGIN
        DISPLAY ("    Integer"):
        DISPLAY ("    "!!
                 STRING (SIM_ITEM_PRECISION (ITEM_INFO)
                       ,*
                      )!!
                 " Digits"
              );
        END;
    (ITEM_IS_BCD_V):
```

Example 7–3. Gathering Query Information (cont.)

```
    BEGIN
    DISPLAY ("    BCD Number"):
    DISPLAY ("    Precision: "!!
             STRING (SIM_ITEM_PRECISION (ITEM_INFO)
                   ,*
                    )
          );
    DISPLAY ("    Scale: "!!
             STRING (SIM_ITEM_SCALE (ITEM_INFO), *)
          );
    IF SIM_ITEM_SIGNED (ITEM_INFO) THEN
       DISPLAY ("    Signed");
    END;
(ITEM_IS_STRING_V):
    BEGIN
    CASE SIM_ITEM_SUBTYPE (ITEM_INFO) OF
       BEGIN
    (ITEM_IS_EBCDIC_V):
       DISPLAY ("    Fixed length EBCDIC string"):
    (ITEM_IS_KANJI_V):
       DISPLAY ("    Fixed length KANJI string"):
    ELSE:
       DISPLAY ("    Fixed length string of "!!
                "unknown type"
               );
       END;  % CASE on subtype
    DISPLAY ("    Length: "!!
             STRING (SIM_ITEM_LENGTH (ITEM_INFO),*)
            );
    DISPLAY ("    CCS Version: "!!
             STRING (SIM_ITEM_CCSVERSION(ITEM_INFO)
                   ,*
                    )
            );
    END;
(ITEM_IS_VLSTRING_V):
    BEGIN
    CASE SIM_ITEM_SUBTYPE (ITEM_INFO) OF
       BEGIN
    (ITEM_IS_EBCDIC_V):
       DISPLAY ("    Variable length EBCDIC "!!
                "string"
               );
    (ITEM_IS_KANJI_V):
       DISPLAY ("    Variable length KANJI "!!
                "string"
             ):
       END;  % CASE on subtype
```

Example 7–3. Gathering Query Information (cont.)

```
                    DISPLAY ("    Maximum length: "!!
                            STRING (SIM_ITEM_LENGTH (ITEM_INFO), *)
                          );
                    DISPLAY ("    CCS Version: "!!
                             STRING (SIM_ITEM_CCSVERSION(ITEM_INFO)
                                   ,*
                                  )
                          );
                    END;
                (ITEM_IS_Boolean_V):
                    BEGIN
                    DISPLAY ("    Boolean"):
                    END;
                END;

            TEMPLATE_NO:= TEMPLATE_NO + 1;
            SIM_QT_TEMPLATE_NUMBER (TEMPLATE_INFO):= TEMPLATE_NO;
            END;

        DISPLAY ("Total number of attributes retrieved by "!!
                 "this query = "!! STRING (TOTAL_ITEMS, *)
              );
        END;
END_OF_PROCEDURE:
    END;
```

Example 7–3. Gathering Query Information

Returning Statistical Information

Use the SIM_QD_CONTROLLER entry point to interrogate the Query Driver for statistical and processing information about a query.

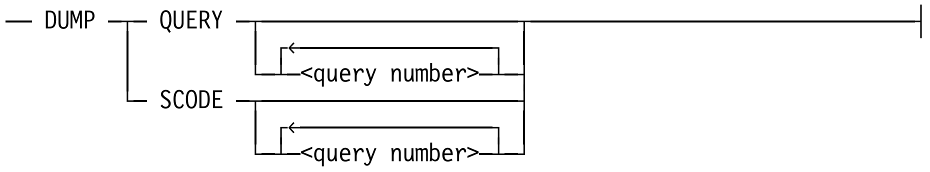
Use the SIM_QD_CONTROLLER entry point to request a dump containing the following information:

- A listing of the internal SIM instructions (the symbolic form of the query).
- The semicompiled code (s-code) produced during query processing. S-code is a miniprogram produced by SIM to produce efficient query processing.
- A query graph showing the strategy tree used in the optimization of a query.

There is one parameter—qdc_text—associated with the SIM_QD_CONTROLLER entry point. The qdc_text parameter is a string that enables you to designate the type of information you want and, using the <query number> construct, the queries for which you want information. The value for the <query number> construct is the IC_QUERY_NO item of the CURSOR_INFO record associated with the query. To use the CURSOR_INFO record, refer to “Using the CURSOR_INFO Record” later in this section.

Use the following command to generate a dump of the query symbolic or the s-code:

Generating Dumps



If you do not designate a query number, all the queries currently known to the Query Driver are dumped.

The information from this command is stored in a file called

```
QDUMP/<mix number>/<serial number>
```

The mix number is that of the user stack, and the serial number is an integer. A value of 1 is assigned as the serial number for the first query processed. The serial number is then incremented by 1 for each subsequent query processed.

To obtain a query graph, you must set the PO_PRINT_QGRAPH parser option for the query. Refer to “Using the QUERY_INFO Record” later in this section for more information on the PO_PRINT_QGRAPH parser option.

The query graph information is dumped to a file called

```
QGRAPH/<mix number>/<serial number>
```

The mix number is that of the user stack, and the serial number is an integer. A value of 1 is assigned as serial number for the first query processed. The serial number is then incremented by 1 for each subsequent query processed.

Use the following command to end the dump process and to close the output file:

Ending Dumps

```
── CLOSE ─┬─ QUERIY ─┬──────────────────────────────────────────────────┤
          └─ GRAPH ──┘
```

For more information on using the SIM_QD_CONTROLLER entry point, refer to “SIM_QD_CONTROLLER” in Section 8, “Describing the SIM Library Entry Points.”

Understanding Data Structures

Several of the SIM library entry points use records that are based on information contained in the SYMBOL/SIM/PROPERTIES file. When you write an ALGOL application you must use an INCLUDE statement that associates part of the SYMBOL/SIM/PROPERTIES file with your application. When you write a COBOL application you should include the copy deck provided on the release media called EXAMPLE/SIM/COBOL/COPYDECK.

The following text explains the data structures that you might use in your application and it identifies the SIM library entry points that use the data structures.

Using the Fields Associated with the Result of Any Entry Point

Each entry point has an associated Boolean result that indicates whether the call to the SIM library processed correctly. Table 7–4 lists the fields associated with this Boolean result.

To interrogate the value associated with any field in the Boolean result, use a statement with the following format:

```
IF BOOLEAN = SIM_PARSE_QUERY
   THEN
   IF EX_CATEGORY(BOOLEAN) = 3
   DO.....
```

Table 7–4. Entry Point Result Word Layout

Field Name	Description
EX_UPDATECOUNT	Duplicates value produced by IC_TUPLE_COUNT field in CURSOR_INFO record.
EX_CATEGORY	Contains the error category. Refer to Appendix B, “Exception Fields and Categories,” for an explanation of the error categories that might be returned.
EX_SUBCATEGORY	Contains the error subcategory. Refer to Appendix B, “Exception Fields and Categories,” for an explanation of the error subcategories associated with each error category.
EX_MORE_EXCEPTIONS	Returns a true value if there are more exceptions.
EX_EXCEPTION	Returns a true value if there is an exception.

Using the CURSOR_INFO Record

You use the CURSOR_INFO record with the following entry points:

- SIM_CLOSE_CURSOR
- SIM_GET_VALUE
- SIM_OPEN_CURSOR
- SIM_PROCESS_CURSOR
- SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA
- SIM_PUT_VALUE

The size of the CURSOR_INFO record is defined by the field IC_USER_CURSOR_SIZE. Therefore, in your ALGOL application, you must include the following declaration:

```
ARRAY CURSOR_INFO[0:IC_USER_CURSOR_SIZE];
```

SIM uses the CURSOR_INFO record as a pointer to a particular instance of a query. Whenever a cursor is required by an entry point, you should use the CURSOR_INFO record as the actual parameter. The only exception to this requirement occurs when you use the SIM_PUT_VALUE entry point. If the query you are processing contains a clause in the form CURRENT(#1), you need to use the cursor to supply the value for the parameter—in this instance #1. When you use the SIM_PUT_VALUE to supply a parameter value, you must assign the IC_CURSOR_NO field as the value for the numeric_value parameter of the SIM_PUT_VALUE entry point. You can also set the following fields in the CURSOR_INFO record:

- IC_BREAK_OUT
- IC_CANCEL_OUT
- IC_USER_QUERY_NO
- IC_USER_CURSOR_NO

All the other fields in the CURSOR_INFO record are read-only fields.

To interrogate the value associated with any field in the CURSOR_INFO record, use a statement with the following format:

```
<value>:= <field name>(CURSOR_INFO);
```

To assign a value to a field, use a statement with the following format:

```
<field name>(CURSOR_INFO):= <value>;
```

Table 7–5 describes the fields in the CURSOR_INFO record.

Table 7–5. CURSOR_INFO Record Layout

Field Name	Description
IC_QUERY_NO	Provides the SIM-allocated query number. SIM assigns this value when you open the cursor.
IC_CURSOR_NO	Provides the SIM-allocated cursor number. SIM assigns this value when you open the cursor.
IC_LEVEL_NO	Provides the template level of the last output returned by the cursor.
IC_TEMPLATE_NO	Provides the template number for the last template returned by the cursor.
IC_BREAK_OUT	Allows a query interrupt bit to be set that causes a currently processing query to stop processing. Subsequently, the query processing can be restarted.
IC_CANCEL_OUT	Cancels a currently processing query. The query processing cannot be restarted.
IC_TUPLE_COUNT	Provides the number of perspective entities effected by the update query. This field is not valid for retrieval queries.
IC_USER_CURSOR_NO	Enables you to supply a cursor number.
IC_USER_QUERY_NO	Enables you to supply a query number.

Using the EXCEPTION_INFO Record

You use the EXCEPTION_INFO record with the SIM_EXCEPTION_INFO entry point.

The size of the EXCEPTION_INFO record is defined by the field EI_SIZE. Therefore, in your ALGOL application, you must include the following declaration:

```
ARRAY EXCEPTION_INFO[0: EI_SIZE];
```

To interrogate the fields in the EXCEPTION_INFO record, use a statement with the following format:

```
<value>:= <field name>(EXCEPTION_INFO);
```

To assign a value to a field, use a statement with the following format:

```
<field name>(EXCEPTION_INFO):= <value>;
```

Table 7–6 describes the fields in the EXCEPTION_INFO record.

Table 7–6. EXCEPTION_INFO Record Layout

Field Name	Description
EI_DMSTATUS	Returns the DMSTATUS word. The DMSTATUS word consists of the following: • The number of the structure involved in the error. • The DMSII category. • The DMSII subcategory. • A Boolean value indicating if an error occurred. If an error occurred the Boolean value is set to true.
EI_DMSTR_NUM	Returns the structure number involved in the error.
EI_DMCATEGORY	Returns the DMSII category.
EI_DMSUBCATEGORY	Returns the DMSII subcategory.
EI_DMERROR	Set to true if an error occurred.
EI_LUC_NAME	Names the SIM structure involved in the error.
EI_DB_NAME	Names the SIM database involved.
EI_STR_NAME	Names the DMSII structure involved.
EI_VERIFY_NAME	Names the SIM verify involved. This field is only valid if there are user-declared verifies in the database schema.
EI_DML_OFFSET	Provides the location of the token involved in a syntax error. This field is valid only for queries written in SIM OML.
EI_TOKEN	Names the token involved in the syntax error. This field is valid only for queries written in SIM OML.

Using the ITEM_INFO Record

You use the ITEM_INFO record with the following entry points:

- SIM_DESCRIBE_ITEM
- SIM_GET_VALUE
- SIM_PUT_VALUE

The size of the ITEM_INFO record is defined by SIM_ITEM_RECORD_SIZE. Therefore, in your ALGOL application, you must include the following declaration:

```
ARRAY ITEM_INFO[0: SIM_ITEM_RECORD_SIZE];
```

To interrogate the value associated with any field in the ITEM_INFO record, use a statement with the following format:

```
<value>:= <field name>(ITEM_INFO);
```

To assign a value to a field, use a statement with the following format:

```
<field name>(ITEM_INFO):= <value>;
```

Some of the fields in the ITEM_INFO record start with SIM_ITEM_. These fields are returned when you use the SIM_DESCRIBE_ITEM entry. When you are using the SIM_DESCRIBE_ITEM entry point, use the SIM_ITEM_NUMBER field to indicate which item you want information on. The value can indicate either the position of the item in the query target list, or the number of the query parameter.

Table 7–7 describes the fields in the ITEM_INFO record.

Table 7–7. ITEM_INFO Record Layout

Field Name	Description
SIM_ITEM_NUMBER	Identifies the item being described.
SIM_ITEM_TEMPLATE	Identifies the template that contains the item.
SIM_ITEM_TYPE	Describes the item type. Valid item types are • ITEM_IS_REAL_V Identifies a real item. • ITEM_IS_INTEGER_V Identifies an integer item. • ITEM_IS_BCD_V Identifies a binary-coded decimal item. • ITEM_IS_STRING_V Identifies a string item. • ITEM_IS_VLSTRING_V Identifies a variable-length string item. • ITEM_IS_BOOLEAN_V Identifies a Boolean item. • ITEM_IS_DOUBLE_V Identifies a double item. • ITEM_IS_DOUBLE_REAL_V Identifies a double real item. • ITEM_IS_DOUBLE_INTEGER_V Identifies a double integer item. • ITEM_IS_CURSOR_NO_V Provides a cursor for the CURRENT(Query-No) construct. • ITEM_IS_SURROGATE_V Provides the surrogate for an entity-reference variable.
SIM_ITEM_SIGNED	Evaluates to true if the item is signed.
SIM_ITEM_NULL	Evaluates to true if the item is null.

Continued

Table 7–7. ITEM_INFO Record Layout (cont.)

Field Name	Description
SIM_ITEM_TARGET_LIST_INX	Provides the target list index for the item.
SIM_ITEM_LENGTH	Provides the length of the item. The value depends on the data type of the item as follows: • Returns a bit value for real and Boolean values. • Returns a byte value for 8-bit strings. • Returns a double byte (16 bits) value for Kanji strings. • Returns a digit value for surrogates and cursors. This field is not valid for any other item types.
SIM_ITEM_SCALE	Returns the scale of items with a data type of integer or number.
SIM_ITEM_PRECISION	Returns the length of items with a data type of integer or number.
SIM_ITEM_CCSVERSION	Returns the ccsversion number.
SIM_ITEM_DATABASE_NO	Returns the internal database number.
SIM_ITEM_INTERNAL_ID	Returns the internal identifier for the item.
SIM_ITEM_BITSPERCHAR	Returns a value of 8 or 16 to indicate the number of bits in a character.

Using the OPEN_OPTIONS Record

You use the OPEN_OPTIONS record with the SIM_OPEN_DATABASE entry point.

The size of the OPEN_OPTIONS record is defined by the field SIM_OPEN_DATABASE_SIZE. Therefore, in your ALGOL application, you must include the following declaration:

```
ARRAY OPEN_OPTIONS[0: SIM_OPEN_DATABASE_SIZE];
```

To interrogate the values associated with any field in the OPEN_OPTIONS record, use a statement with the following format:

```
<value>:= <field name>(OPEN_OPTIONS);
```

To assign a value to a field, use a statement with the following format:

```
<field name>(OPEN_OPTIONS):= <value>;
```

Table 7–8 describes the fields in the OPEN_OPTIONS record.

Table 7–8. OPEN_OPTIONS Record Layout

Field Name	Description
SIM_OPEN_DATABASE_VERSION	Provides the version of the database you want to open. You can designate this value.
SIM_OPEN_DATABASE_STATUS	Provides the database status. A value of 2 requests a test database, while a value of 3 requests a production database. You can designate this value.
SIM_OPEN_DATABASE_NUMBER	Provides the internal number of the database. SIM designates this value.
SIM_OPEN_DATABASE_MODE	Provides the database open mode. A value of 0 (zero) requests update access, while a value of 5 requests inquiry access. You can designate this value.

Using the QUERY_DESC Record

You use the QUERY_DESC record with the SIM_PARSE_FORMATTED_QUERY and SIM_DESCRIBE_FORMAT entry points.

The size of the QUERY_DESC record is dependent upon the number of items to be returned in the query. A size of 1024 words is recommended as a starting size for this record. In your ALGOL application this size is declared as follows:

```
ARRAY QUERY_DESC [0:1023];
```

The examples for the SIM_DESCRIBE_QUERY and the SIM_PARSE_FORMATTED_QUERY entry points in Section 8, “Describing the SIM Library Entry Points,” illustrate the ALGOL code for handling a situation where the correct size of the QUERY_DESC record cannot be determined at the start of the program.

It is recommended that you **do not** build this record. If you want to format the input or the output templates, use the following procedure:

1. Parse the query using the SIM_PARSE_FORMATTED_QUERY entry point.
2. Set the QUERY_INFO record format option as follows:

   ```
   PO_USER_FORMATTED(QUERY_INFO):= 1
   ```

3. Go through the QUERY_DESC record returned by the SIM_PARSE_FORMATTED_QUERY entry point setting the desired user field offsets.

   (Refer to the example program EXAMPLE/SIM/ALGOL/API/RECORDATATIME on the release media to see how to accomplish this step.)
4. Reparse the query using the SIM_PARSE_FORMATTED_QUERY entry point with the modified QUERY_DESC record.

In the QUERY_DESC record there are three basic groups of fields as follows:

- Field names that start with DESC_ are part of the description header information. To interrogate values in the DESC_fields, you must use a statement with the following format:

  ```
  <value>:= <DESC_ field>(QUERY_DESC);
  ```

- Field names that start with TH_ are part of the template header. To interrogate the TH_ fields, you must use a statement with the following format:

  ```
  <value>:= <TH_ field>(QUERY_DESC,<DESC_ field value>);
  ```

- Field names that start with ITEM_ are part of the item directory. To interrogate the ITEM_ fields, you must use a statement with the following format:

  ```
  <value>:= <ITEM_ field>(QUERY_DESC, <item offset>);
  ```

  To calculate the item offset, refer to the program EXAMPLE/SIM/ALGOL/API/RECORDATATIME on the release media.

Table 7–9 describes the fields in the QUERY_DESC record. Note that only the offset and size fields can be set; all the other fields are effectively read-only fields.

Table 7–9. QUERY_DESC Record Layout

Field Name	Description
DESC_VERSION	Provides the version level. This field is always set a value of DESC_VERSION_LEVEL.
DESC_STATEMENT	Provides the query type. This field can take any of the following values: • ST_RETRIEVE_V for a retrieval query • ST_INSERT_V for an insert query • ST_MODIFY_V for a modify query • ST_DELETE_V for a delete query
DESC_TABULAR	Provides a Boolean value indicating whether the query has a tabular or a structured format. A value of true identifies a tabular format.
DESC_INPUT_REQUIRED	Provides a Boolean value indicating whether there are query parameters included in the query text. A value of true indicates that there are query parameters.
DESC_DML_LANGUAGE	Returns the query language, as follows: • A value of DESC_LANG_SIMV identifies a SIM OML query. • A value of DESC_LANG_SQLV identifies an SQLDB DML query.
DESC_DATA_LENGTH	Provides the maximum data size that the query can return.
DESC_LEVEL_COUNT	Provides the maximum number of template levels used by the query.
DESC_TEMPLATE_COUNT	Provides the number of templates used by the query.
DESC_OUTPUT_TEMPLATE	Provides the offset of the output template description.
DESC_INPUT_TEMPLATE	Provides the offset of the input template description.
TH_SIZE	Provides the length of the current template. The start of the next template is located TH_SIZE more than the start of the current template.
TH_LEVEL_NUMBER	Provides the level number of the current template.
TH_TEMPLATE_NUMBER	Provides the template number of the current template.
TH_ITEM_COUNT	Provides a count of the items in the current template.

Continued

Table 7–9. QUERY_DESC Record Layout (cont.)

Field Name	Description
ITEM_TRUE_TYPE	Identifies items of type date and time which are being returned as integers. To use this field, you must identify the offset of the item you are interested in.
ITEM_DATA_TYPE	Describes the item type. Valid item types are • ITEM_IS_REAL_V Identifies a real item. • ITEM_IS_INTEGER_V Identifies an integer item. • ITEM_IS_BCD_V Identifies a binary-coded decimal item. • ITEM_IS_STRING_V Identifies a string item. • ITEM_IS_VLSTRING_V Identifies a variable-length string item. • ITEM_IS_BOOLEAN_V Identifies a Boolean item. • ITEM_IS_DOUBLE_V Identifies a double item. • ITEM_IS_DOUBLE_REAL_V Identifies a double real item. • ITEM_IS_DOUBLE_INTEGER_V Identifies a double integer item. • ITEM_IS_CURSOR_NO_V Provides a cursor for the CURRENT(Query-No) construct. • ITEM_IS_SURROGATE_V Provides the surrogate for an entity-reference variable.
ITEM_BYTE_OFFSET	Provides the buffer item offset in bytes.
ITEM_DIGIT_OFFSET	Provides the buffer item offset in digits.

Continued

Table 7–9. QUERY_DESC Record Layout (cont.)

Field Name	Description
ITEM_BIT_IN_DIGIT	Provides the buffer item offset within the digit. This field can take the values 0 through 3.
ITEM_SIGNED	Identifies a signed item with a true value.
ITEM_LENGTH	Provides the length of an item. The value returned depends on the data type of the item as follows: • The number of bits are returned for Boolean and Real items. • The number of bytes are returned for 8-bit strings. • The number of double bytes (16 bits) are returned for 16-bit strings. • The number of digits are returned for surrogates and cursor numbers. This field is not valid for any other data types.
ITEM_SCALE	Provides the scale for items with a data type of number or integer.
ITEM_PRECISION	Provides the length for items with a data type of number or integer.
ITEM_NAME	Provides the offset in the QDESC record of the name of the item.
ITEM_NAME_LEN	Provides the length of the name of the item.
ITEM_NAME_PTR	Provides an ALGOL pointer to the name of the item.
ITEM_NULL_OFFSET	Provides the offset in the buffer for the null bit for the item. The null bit field is always 1 digit long.
ITEM_TARGET_LIST_INX	Provides the ordinal position of the item in the target list.
ITEM_STRING_CCSVERSION	Identifies the ccsversion number.
ITEM_STRING_BITSPERCHAR	Identifies the number of bits needed for a single character. This field can take the values 8 and 16.
ITEM_STRING_SOURCE_LEN	Provides the indicator parameter for SQLDB DML queries.
ITEM_SURROGATE_CLASS	Provides the internal identifier for the class of an entity-reference variable.
ITEM_LUC_NUMBER	Provides the internal identifier for the item. A value of 0 (zero) means that this field is not valid for the item.
ITEM_DB_NUMBER	Provides the database number of the item. This field has no meaning for saved queries.

Using the QUERY_INFO Record

You use the QUERY_INFO record with the following entry points:

- SIM_ADD_QUERY
- SIM_CLOSE_QUERY
- SIM_DESCRIBE_DML_TEXT
- SIM_DESCRIBE_ITEM
- SIM_DESCRIBE_PARAMETER
- SIM_DESCRIBE_QUERY
- SIM_DESCRIBE_TEMPLATE
- SIM_LOAD_QUERY
- SIM_OPEN_CURSOR
- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

The size of the QUERY_INFO record is defined by the field PO_SIZE. Therefore, in your ALGOL application, you must include the following declaration:

```
ARRAY QUERY_INFO[0:PO_SIZE];
```

To interrogate the value associated with any field in the QUERY_INFO record, use a statement with the following format:

```
<value>:= <field name>(QUERY_INFO);
```

To assign a value to a field, use a statement with the following format:

```
<field name>(QUERY_INFO):= <value>;
```

Table 7–10 describes the fields in the QUERY_INFO record.

Table 7–10. QUERY_INFO Record Layout

Field Name	Description
PO_QUERY_NO	Contains the internal query number assigned by SIM.
PO_PACK_ALL	Byte and digit aligns the target list items where possible. Assign a value of 1 to this field to set this option.
PO_USER_FORMATTED	Provides the layout of the input and output templates. This field is set if you use the SIM_PARSE_FORMATTED_QUERY or the SIM_DESCRIBE_QUERY entry point.
PO_LINCDELETED	Returns the *MAINT="D"* records for a LINC II database. Assign a value of 1 to this field to set this option.
PO_PRINT_QGRAPH	Returns a query tree and a strategy tree for the query. The trees are written to a file called QGRAPH/<mix number>/<serial number>. These trees are useful when you are trying to improve the efficiency of your query. Refer to "Writing Efficient Queries" in Section 1, "Introduction to OML," for more details on query and strategy trees. Assign a value of 1 to this field to set this option.
PO_FORMAT	Sets options for different data types. Refer to Table 7–11 for a list of the assignments you can make.
PO_USER_QUERY_NO	A 16-bit value that you can assign to track your query. This field is not used by SIM.
PO_ID_LENGTH	Provides the length of the PO_ID field. You designate this value if you want to use the PO_ID field to help track your query. This field is not used by SIM.
PO_ID	Acts as a pointer to your query. The string you assign must be less than 30 characters. You define the length of this field by using the PO_ID_LENGTH field. This field is not used by SIM.
PO_INTL_CCSVERSION	A 16-bit value that you can assign to the ccsversion to be used in parsing the query.
PO_INTL_BITSPERCHAR	An 8-bit value that you can assign as the number of bits required to define a character.

You must use the following format to assign values to the PO_FORMAT field:

```
PO_FORMAT (QUERY_INFO,<initial data type>):= <new data type>;
```

For example, to obtain symbolics in their internal format, that is, as integers instead of strings, you make the following assignment:

```
PO_FORMAT (QUERY_INFO,PO_SYMBOLIC):= PO_FMT_INTERNAL;
```

Table 7–11 gives the values you can assign in place of the <new data type> construct. You need to make these assignments before parsing your query.

Table 7–11. Designating Data Formats

Initial Data Type	Allowed Assignments	Explanation
Date, symbolic	PO_FMT_DEFAULT	Returns values as identifiers.
	PO_FMT_INTERNAL	Returns values as numbers.
Number	PO_FMT_REAL	Formats numbers as real values.
	PO_FMT_INTEGER	Formats numbers as integers. If you choose this option, you are responsible for ensuring that the scale of the number is correct.
	PO_FMT_COMP_NUMERIC	Formats COBOL display numeric pictures as computational numeric pictures, and formats trailing sign numbers as leading sign numbers.
Character	PO_FMT_LEFT_JUST	Left-justifies the character in the word.
Variable-length string	PO_FMT_FIX_STRING	Formats variable-length strings as strings.
	PO_FMT_PAD_STRING	Pads variable-length strings with blanks, but retains the length word.

Table 7–12 gives the values you can assign in place of the <initial data type> construct.

Table 7–12. Data Type Constructs

Construct	Explanation
PO_INTEGER	Integer
PO_REAL	Real
PO_NUMBER	Number
PO_CHARACTER	Characters of any type
PO_STRING	Strings of any type
PO_VLSTRING	Strings of any type
PO_DATE	Date
PO_TIME	Time
PO_BOOLEAN	Boolean
PO_SYMBOLIC	Symbolic
PO_POPULATION	Population
PO_CURSOR_REF	Cursor number for the query
PO_ERV	Entity-reference variable

Using the QUERY_RECORD Record

You use the QUERY_RECORD record with the SIM_DESCRIBE_QUERY entry point.

The size of the QUERY_RECORD record is defined by SIM_Q_SIZE. Therefore, in your ALGOL application, you must include the following declaration:

```
ARRAY QUERY_RECORD[0: SIM_Q_SIZE];
```

To interrogate the values associated with any field in the QUERY_RECORD record, use a statement with the following format:

```
<value>:= <field name>(QUERY_RECORD);
```

To assign a value to a field, use a statement with the following format:

```
<field name>(QUERY_RECORD):= <value>;
```

Table 7–13 describes the fields in the QUERY_RECORD record.

Table 7–13. QUERY_RECORD Record Layout

Field Name	Description
SIM_Q_STATEMENT	Identifies the query type. The values that can be associated with this field are • ST_RETRIEVE_V for a retrieval query • ST_INSERT_V for an insert query • ST_MODIFY_V for a modify query • ST_DELETE_V for a delete query
SIM_Q_TABULAR	Returns a true value if the query is tabular.
SIM_Q_INPUT_REQUIRED	Returns a true value if the query has parameters.
SIM_Q_LEVELS	Returns the number of template levels in the query.
SIM_Q_RECORD_LENGTH	Returns the maximum buffer size for the query.

Using the TEMPLATE_INFO Record

You use the TEMPLATE_INFO record with the SIM_DESCRIBE_TEMPLATE entry point.

The size of the TEMPLATE_INFO record is defined by the field SIM_QT_SIZE. Therefore, in your ALGOL application, you must include the following declaration:

```
ARRAY TEMPLATE_INFO[0: SIM_QT_SIZE];
```

To interrogate the value associated with any field in the TEMPLATE_INFO record, use a statement with the following format:

```
<value>:= <field name>(TEMPLATE_INFO);
```

To assign a value to a field, use a statement with the following format:

```
<field name>(TEMPLATE_INFO):= <value>;
```

Table 7–14 describes the fields in the TEMPLATE_INFO record.

Table 7–14. TEMPLATE_INFO Record Layout

Field Name	Description
SIM_QT_TEMPLATE_NUMBER	Provides the template number for the current template.
SIM_QT_LEVEL_NUMBER	Provides the level number for the current template.
SIM_QT_ITEM_COUNT	Provides the number of items in the current template.
SIM_QT_RECORD_LENGTH	Provides the record length of the current template.

Section 8
Describing the SIM Library Entry Points

This section provides the detailed information you need to use each SIM library entry point. The entry points are described in alphabetical order. For task-oriented information about the entry points, refer to Section 7, “Using the SIM Library Entry Points.”

In this section and in Section 7, “Using the SIM Library Entry Points,” the entry points all start with SIM_. If you are using COBOL or RPG, replace *SIM_* with *SCIM_*.

The SCIM_ entry points also have an additional parameter that is inserted at the start of the parameter list and is used to return the result of the entry point. For example, the entry point SIM_OPEN_DATABASE is described as having four associated parameters, as follows:

- open_options
- dbname_text
- usercode_text
- family_text

In a COBOL or RPG program, the equivalent entry point, SCIM_OPEN_DATABASE, has five associated parameters, as follows:

- result
- open_options
- dbname_text
- usercode_text
- family_text

You use the result parameter to establish whether there are any error messages associated with the processing of the entry point. Define the result parameter as a numeric of the following form:

```
PIC 9(19) COMP.
```

The form for using a SIM library entry point in a COBOL program is as follows:

```
CALL "<entry point> OF SIMDRIVERSUPPORT" USING <parameter list>
```

This section and Section 7, “Using the SIM Library Entry Points,” use the following convention to distinguish between information extracted from the SYMBOL/SIM/PROPERTIES file and user-defined strings:

- For ALGOL
  - The term *record* refers to an array containing information based on fields defined in the SYMBOL/SIM/PROPERTIES file.
  - The term *string* refers to a REAL array containing a one word size field followed by text. The data contained in the array is entry-point dependent.
- For COBOL
  - The term *record* refers to data based on the fields defined in the SYMBOL/SIM/PROPERTIES file.
  - The term *string* refers to arrays containing a size field followed by text that are application dependent.

Note: *The following entry points are not defined for COBOL:*

- *SIM_PARSE_FORMATTED_QUERY*
- *SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA*
- *SIM_PROCESS_CURSOR_AND_UPDATE_DATA*

SIM_ABORT

Purpose

Use the SIM_ABORT entry point to undo all changes made to the database by an application since the last begin-transaction marker. The SIM_ABORT entry point also ends the transaction. Refer to “Performing Queries While in Transaction State” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

There are no parameters associated with the SIM_ABORT entry point.

Related Entry Points

The following entry points are also used with transactions:

- SIM_BEGIN_TRANSACTION
- SIM_CANCEL
- SIM_ETR
- SIM_SAVEPOINT

Example

The following ALGOL program fragment illustrates the use of the SIM_ABORT entry point. In the example, both SIM_OPEN_CURSOR calls must complete successfully in order for the requested changes to be made to the database. If either update fails, all the requested database changes are canceled by using the SIM_ABORT entry point.

It is assumed that the appropriate QUERY_INFO_x and CURSOR_INFO_x arrays have already been declared and that the QUERY_INFO arrays now contain information from calls to the SIM library using the SIM_PARSE_QUERY entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Enter transaction state.
  IF RESULT:= SIM_BEGIN_TRANSACTION (LEVEL_LOCKING) THEN
      BEGIN
      % There was an error trying to enter transaction
      % state.
      PROCESS_ERROR;
      GO END_OF_EXAMPLE;
      END;

  % The program is now in transaction state - process the compiled
  % queries.
  IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_1)
                             ,PASS_ARRAY (CURSOR_INFO_1)
                            ) THEN
      BEGIN
      % Some kind of error occurred.
      PROCESS_ERROR;
      ERROR_OCCURRED:= TRUE;
      GO UPDATE_CHECK;
      END
  ELSE
      BEGIN
      % Now update the database.
      IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_1)) THEN
          BEGIN
          % Some kind of error occurred.
          PROCESS_ERROR;
          ERROR_OCCURRED:= TRUE;
          GO UPDATE_CHECK;
          END;
      END;

  IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_2)
                             ,PASS_ARRAY (CURSOR_INFO_2)
                            ) THEN
      BEGIN
      % Some kind of error occurred.
      PROCESS_ERROR;
      ERROR_OCCURRED:= TRUE;
      GO UPDATE_CHECK;
      END
  ELSE
```

```
    BEGIN
    % Now update the database.
    IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_2)) THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        ERROR_OCCURRED:= TRUE;
        GO UPDATE_CHECK;
        END;
    END;

% End transaction state.
IF RESULT:= SIM_ETR THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    GO END_OF_EXAMPLE;
    END;

UPDATE_CHECK:
    IF ERROR_OCCURRED THEN
        BEGIN
        % Undo all updates made by the previous
        % SIM_OPEN_CURSOR calls.
        IF RESULT:= SIM_ABORT THEN
            BEGIN
            % An error occurred while undoing the transactions.
            PROCESS_ERROR;
            END;
        GO END_OF_EXAMPLE;
        END;

END_OF_EXAMPLE:
```

SIM_ADD_QUERY

Purpose

Use the SIM_ADD_QUERY entry point to restore query information from a saved file. Refer to “Saving and Restoring Queries” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

file_title

Type: String

Allowable Values	Description
Any valid A Series file title	Designates the title of the file containing the saved query information. The file_title parameter is an input parameter. The first word of the array must contain the length of the string.

query_info

Type: Record

Allowable Values	Description
System-supplied	Provides an array of query information (QUERY_INFO) records. Any valid query information that was contained in the query_info parameter before you used the SIM_ADD_QUERY entry point is retained, and the information for the restored queries is appended to the end of the existing information. In your program, you must examine the PO_USER_QUERY_NO, the PO_ID_LENGTH, and the PO_ID fields of each query identifier record. For more information, refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points.”

Note: *You must ensure that the array you declare for the query_info parameter is large enough to contain all information for those queries that were open before you used the SIM_LOAD_QUERY entry point and the information for all queries in the saved query file.*

Related Entry Points

The following entry points are also used with saving and restoring queries:

- SIM_LOAD_QUERY
- SIM_SAVE_QUERY

Example

The following ALGOL example demonstrates the use of the SIM_ADD_QUERY entry point. In this example, it is assumed that the database accessed by the queries in the saved file is already open.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to "Returning Error Messages" in Section 7, "Using the SIM Library Entry Points," for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to "Passing Arrays to SIM from ALGOL Programs" in Section 7, "Using the SIM Library Entry Points," for an explanation of the PASS_ARRAY define.

```
% Declare the file_title and query_info arrays.
  ARRAY
      FILE_TITLE [0:255]
    ,QUERY_INFO [0:PO_SZ]
   ;
  BOOLEAN
      RESULT
    ,QUERY_FILE_ADDED
   ;
  POINTER PTR;
  % Set up file title.
  REPLACE PTR:POINTER (FILE_TITLE [1]) BY "SAVED/QUERIES";
  FILE_TITLE [0]:= DELTA (POINTER (FILE_TITLE [1]), PTR);
  QUERY_FILE_ADDED:= FALSE;
  DO BEGIN
     IF RESULT:= SIM_ADD_QUERY (PASS_ARRAY (FILE_TITLE)
                              ,PASS_ARRAY (QUERY_INFO)
                             ) THEN
         BEGIN
         IF EX_CATEGORY (RESULT) = EX_FAILED AND
            EX_SUBCATEGORY (RESULT) = EX_FAIL_QID_INFO_TOO_SMALL THEN
             BEGIN
             % Query info array too small. Resize array larger.
             RESIZE (QUERY_INFO, SIZE(QUERY_INFO)+PO_SZ, DISCARD);
             END
         ELSE
             BEGIN
             % Some kind of error occurred.
             PROCESS_ERROR;
             GO END_OF_EXAMPLE;
             END
         END
     ELSE
         QUERY_FILE_ADDED:= TRUE;
     END
  UNTIL QUERY_FILE_ADDED;
  END_OF_EXAMPLE:
```

SIM_BEGIN_TRANSACTION

Purpose

Use the SIM_BEGIN_TRANSACTION entry point to indicate to SIM that you wish to begin transaction state. Refer to “Performing Queries While in Transaction State” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

locking_level

Type: Integer

Allowable Values	Description
0 (zero) or RECORD_LEVEL_LOCKING_V	Requests record-level locking.
1 or DATABASE_LEVEL_LOCKING_V	Requests database-level locking.
2 or STRUCTURE_LEVEL_LOCKING_V	Requests structure-level locking.

Related Entry Points

The following entry points are also used with transactions:

- SIM_ABORT
- SIM_CANCEL
- SIM_COMS_BEGIN_TRANSACTION
- SIM_COMS_ETR
- SIM_ETR
- SIM_SAVEPOINT

Example

The following ALGOL example demonstrates the use of the SIM_BEGIN_TRANSACTION entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

```
% Declare locking_level.
INTEGER LOCKING_LEVEL;

% Exclusive use of the structure is requested.
LOCKING_LEVEL:= STRUCTURE_LEVEL_LOCKING_V;

% Enter transaction state.
IF RESULT:= SIM_BEGIN_TRANSACTION (LOCKING_LEVEL) THEN
   BEGIN
   % There was an error trying to enter transaction
   % state.
    PROCESS_ERROR;
    GO END_OF_EXAMPLE;
    END;

% The program is now in transaction state - process the compiled
% queries. For the purposes of brevity, this code has been omitted.

END_OF_EXAMPLE;
```

SIM_CANCEL

Purpose

Use the SIM_CANCEL entry point to cancel all changes made to the database by an application since a designated savepoint. For more information, refer to “Undoing All Updates Since the Last Savepoint Marker” in Section 7, “Using the SIM Library Entry Points.”

Parameters

savepoint

Type: Integer

Allowable Values	Description
Any value designated by using the SIM_SAVEPOINT entry point	Designates the point after which all changes made by the application are to be canceled. To place a savepoint, use the SIM_SAVEPOINT entry point.

Related Entry Points

The following entry points are also used with transactions:

- SIM_ABORT
- SIM_BEGIN_TRANSACTION
- SIM_COMS_BEGIN_TRANSACTION
- SIM_COMS_ETR
- SIM_ETR
- SIM_SAVEPOINT

Example

The following ALGOL example demonstrates the use of the SIM_CANCEL entry point.

In this example, a transaction to the database requires a lengthy process between two different SIM_OPEN_CURSOR calls. If the results of the intermediate process are incorrect, the SIM_CANCEL entry point is used to undo any changes requested by this example since the designated savepoint.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO_x arrays are already declared and are assumed to contain information from a previous call using the SIM_PARSE_QUERY entry point. Also, it is assumed that the CURSOR_INFO_x arrays are already declared.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Enter transaction state.
  IF RESULT:= SIM_BEGIN_TRANSACTION (LOCKING_LEVEL) THEN
      BEGIN
      % There was an error trying to enter transaction
      % state.
      PROCESS_ERROR;
      GO END_OF_EXAMPLE;
      END;

  % The program is now in transaction state - process the compiled
  % query.
  IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_1)
                              ,PASS_ARRAY (CURSOR_INFO_1)
                             ) THEN
      BEGIN
      % Some kind of error occurred.
      PROCESS_ERROR;
      GO END_OF_TRANSACTION;
      END
  ELSE
      BEGIN
      % Now update the database.
      IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_1)) THEN
          BEGIN
          % Some kind of error occurred.
          PROCESS_ERROR;
          GO END_OF_TRANSACTION;
          END;
      END;
```

```
% Mark a savepoint.
IF RESULT:= SIM_SAVEPOINT (1) THEN
    BEGIN
    % Some kind of error occurred while trying to mark the
    % savepoint.
    PROCESS_ERROR;
    GO END_OF_TRANSACTION;
    END;

IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_2)
                            ,PASS_ARRAY (CURSOR_INFO_2)
                           ) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    GO END_OF_TRANSACTION;
    END

ELSE
    BEGIN
    % Now update the database.
    IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_2)) THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        GO END_OF_TRANSACTION;
        END;
    END;

 % Mark a savepoint.
IF RESULT:= SIM_SAVEPOINT (2) THEN
    BEGIN
    % Some kind of error occurred while trying to mark the
    % savepoint.
    PROCESS_ERROR;
    GO END_OF_TRANSACTION;
    END;

% Here is a lengthy update process.
PROCESS_MRP (PROCESS_RESULT);
```

```
 IF PROCESS_RESULT THEN
    BEGIN
    % The results of PROCESS_MRP are incorrect.
    % Undo the updates made from the second savepoint.
    IF RESULT:= SIM_CANCEL (2) THEN
        BEGIN
        % Some kind of error occurred. An error when calling
        % SIM_CANCEL indicates that the updates are not canceled
        % and that the database is still in transaction state.
        PROCESS_ERROR;
        END;
    GO END_OF_TRANSACTION;
    END;

IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_3)
                            ,PASS_ARRAY (CURSOR_INFO_3)
                           ) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    GO END_OF_TRANSACTION;
    END
ELSE

    BEGIN
    % Now update the database.
    IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_3)) THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        GO END_OF_TRANSACTION;
        END;
    END;

END_OF_TRANSACTION:

    % End transaction state.
    IF RESULT:= SIM_ETR THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        GO END_OF_EXAMPLE;
        END;

END_OF_EXAMPLE:
```

SIM_CLOSE_CURSOR

Purpose

Use the SIM_CLOSE_CURSOR entry point to deallocate a cursor for a query. For more information, refer to “Deallocating a Cursor from a Query” in Section 7, “Using the SIM Library Entry Points.”

Parameters

cursor_info

Type: Record

Allowable Values	Description
Same record area as that passed to the SIM_OPEN_CURSOR entry point	Identifies the instance of the query that you want to close. Refer to “Using the CURSOR_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

Related Entry Points

The following entry points also process cursor information:

- SIM_OPEN_CURSOR
- SIM_PROCESS_CURSOR
- SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA
- SIM_PROCESS_CURSOR_AND_UPDATE_DATA

Example

The following ALGOL procedure demonstrates the use of the SIM_CLOSE_CURSOR entry point. If the cursor is successfully closed, the message “Cursor closed” is displayed.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is declared outside of the procedure. The structure is assumed to contain information from a previous call to SIM_PARSE_QUERY.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% The defines for PO_SIZE and IC_CURSOR_SIZE are declared in
 % the file SYMBOL/SIM/PROPERTIES.
 ARRAY
     QUERY_INFO  [0:PO_SIZE],
     CURSOR_INFO [0:IC_CURSOR_SIZE];

 PROCEDURE CLOSE_CURSOR_EXAMPLE;
     BEGIN
     BOOLEAN
         RESULT;                  % Holds the SIM result word.
     LABEL
         END_OF_PROCEDURE;

     % First, initialize the CURSOR_INFO record.
     REPLACE CURSOR_INFO BY 0 FOR SIZE (CURSOR_INFO) WORDS;

     IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO)
                                ,PASS_ARRAY (CURSOR_INFO)
                               ) THEN
         BEGIN
         %  Some kind of error occurred.
         PROCESS_ERROR;
         DISPLAY ("Example failed during OPEN_CURSOR call");
         GO END_OF_PROCEDURE;       % Exits the procedure.
         END;

     WHILE NOT RESULT:=
             SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO)) DO
         BEGIN
         %
         %  Typically, code is added here to retrieve
         %  information about the current tuple.
         %
         END;

   %  The last tuple has been retrieved, so now the cursor needs
     %  to be closed.
     IF RESULT:=
             SIM_CLOSE_CURSOR (PASS_ARRAY (CURSOR_INFO)) THEN
         BEGIN
         %  An error occurred.
         PROCESS_ERROR;
         DISPLAY ("Example failed on CLOSE_CURSOR call");
         END
     ELSE
         % No errors occurred.
         DISPLAY ("Cursor closed");
 END_OF_PROCEDURE:
     END;
```

SIM_CLOSE_DATABASE

Purpose

Use the SIM_CLOSE_DATABASE entry point to close a database. You can improve system performance by keeping the number of open database to a minimum.

Parameters

dbname_text

Type: String

Allowable Values	Description
Any database name	Identifies the database you want to close. The first word of the string must provide the length of the string.

Related Entry Points

The following entry points are used to open a database:

- SIM_OPEN_DATABASE
- SIM_SET_DICTIONARY

Example

The following ALGOL procedure demonstrates the use of the SIM_CLOSE_DATABASE entry point. If the database is successfully closed, the message “Database closed” is displayed.

If the query is not closed, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

It is assumed that you have already provided a database name and that the name is currently stored in the string variable DATABASE_NAME.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
ARRAY
     DBNAME_TEXT [0:30];

 PROCEDURE CLOSE_DATABASE_EXAMPLE;
     BEGIN
     BOOLEAN
         RESULT;                   % Holds the SIM result word.

     %  First, move the database name from the string
     %  DATABASE_NAME to DBNAME_TEXT. DBNAME_TEXT is the
     %  parameter for the SIM_CLOSE_DATABASE entry point.
     REPLACE POINTER(DBNAME_TEXT[1]) BY DATABASE_NAME FOR
         LENGTH (DATABASE_NAME);
     DBNAME_TEXT[0]:= LENGTH (DATABASE_NAME);

     %  Now the database can be closed.
     IF RESULT:=SIM_CLOSE_DATABASE (PASS_ARRAY (DBNAME_TEXT))
     THEN
         BEGIN
         % An error occurred while closing the database.
         PROCESS_ERROR;
         DISPLAY ("Example failed while closing the database");
         END
     ELSE
         DISPLAY ("Database closed");
     END;
```

SIM_CLOSE_QUERY

Purpose

Use the SIM_CLOSE_QUERY entry point to close a query and release all cursors related to that query.

Parameters

query_info

Type: Record

Allowable Values	Description
Same record area as that passed to the query parsing entry points	Identifies the query you want to close. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

Related Entry Points

The following entry points are used to assign a query identifier:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL procedure demonstrates the use of the SIM_CLOSE_QUERY entry point. If the query is successfully closed, the message “Query closed” is displayed.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the processes for opening, processing, and closing the cursors for the query have been omitted.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% The defines for PO_SIZE and IC_CURSOR_SIZE are declared in
 % the file SYMBOL/SIM/PROPERTIES.
 ARRAY
     QUERY_TEXT  [0:30],
     QUERY_INFO  [0:PO_SIZE],
     CURSOR_INFO [0:IC_CURSOR_SIZE];

 PROCEDURE CLOSE_QUERY_EXAMPLE;
     BEGIN
     BOOLEAN
         RESULT;                   % Holds the SIM result word.
     LABEL
         END_OF_PROCEDURE;

     % First, parse a query.
     REPLACE QUERY_TEXT [1] BY
         "From PERSON Retrieve NAME";
     QUERY_TEXT [0]:= 25;
     IF RESULT:= SIM_PARSE_QUERY (PASS_ARRAY (QUERY_INFO)
                                 ,PASS_ARRAY (QUERY_TEXT)
                                ) THEN
         BEGIN
         % An error occurred while parsing the query.
         PROCESS_ERROR;
         DISPLAY ("Example failed while parsing the query");
         GO END_OF_PROCEDURE;        % Exit the procedure.
         END;

     %  The code for opening and processing the cursors
     %  should be placed here. For the purposes of brevity, the code
     %  has been omitted from this example.

     %  Now close the query.
     IF RESULT:= SIM_CLOSE_QUERY (PASS_ARRAY (QUERY_INFO)) THEN
         BEGIN
         %  An error occurred.
         PROCESS_ERROR;
         DISPLAY ("Example failed on CLOSE_QUERY call");
         END
     ELSE
         % No errors occurred.
         DISPLAY ("Query closed");

     END_OF_PROCEDURE:
     END;
```

SIM_COMS_ABORT

Purpose

Use the SIM_COMS_ABORT entry point to undo all changes made to the database by an application since your last COMS begin-transaction marker. The SIM_COMS_ABORT entry point also ends the COMS transaction. For more information, refer to “Performing Queries While in Transaction State” in Section 7, “Using the SIM Library Entry Points.”

Parameters

coms_out

Type: String

Allowable Values	Description
System-supplied	Returns messages from COMS. Refer to the *COMS Programming Guide* for more information.

Related Parameters

The following entry points are also used with COMS transactions:

- SIM_COMS_BEGIN_TRANSACTION
- SIM_COMS_ETR

Example

The following ALGOL example demonstrates the use of the SIM_COMS_ABORT entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

```
% Declare coms_out array.
OUTPUTHEADER
        COMS_OUT;
BOOLEAN RESULT;

% The code for parsing, entering COMS transaction state, and opening
% and processing cursors should be placed here. For the purposes
% of brevity, the code has been omitted from this example.

IF RESULT:= SIM_COMS_ABORT (COMS_OUT) THEN
   BEGIN
   % Some kind of error occurred.
   PROCESS_ERROR;
   END;
```

SIM_COMS_BEGIN_TRANSACTION

Purpose

Use the SIM_COMS_BEGIN_TRANSACTION entry point to initiate a Communications Management System (COMS) transaction. Refer to “Performing Queries While in Transaction State” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

locking_level

Type: Integer

Allowable Values	Description
0 (zero) or RECORD_LEVEL_LOCKING_V	Requests record-level locking.
1 or DATABASE_LEVEL_LOCKING_V	Requests database-level locking.
2 or STRUCTURE_LEVEL_LOCKING_V	Requests structure-level locking.

coms_in

Type: String

Allowable Values	Description
Refer to the *COMS Programming Guide*.	Sends messages to COMS.

Related Entry Points

The following entry points are also used with COMS transactions:

- SIM_CANCEL
- SIM_COMS_ABORT
- SIM_COMS_ETR
- SIM_SAVEPOINT

Example

The following ALGOL example demonstrates the use of the SIM_COMS_BEGIN_TRANSACTION entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

```
% Declare coms_in array.
TYPE INPUTHEADER
        COMS_IN_TYPE (ARRAY CONVERSATION [0:59]);
COMS_IN_TYPE
        COMS_IN;
BOOLEAN RESULT;
INTEGER LOCKING_LEVEL;

% Initialize coms_in message area.
ENABLE (COMS_IN, "ONLINE");

% The code for parsing a query should be placed
% here. For the purposes of brevity, the code has been omitted
% from this example.

% Setting exclusive use of the database.
LOCKING_LEVEL:= DATABASE_LEVEL_LOCKING_V;

% Enter transaction state.
IF RESULT:= SIM_COMS_BEGIN_TRANSACTION (LOCKING_LEVEL, COMS_IN) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    END;
```

SIM_COMS_ETR

Purpose

Use the SIM_COMS_ETR entry point to end a COMS transaction. Refer to “Performing Queries While in Transaction State” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

coms_out

Type: String

Allowable Values	Description
System-supplied	Returns a message area provided by COMS. Refer to the *COMS Programming Guide* for more information.

Related Entry Points

The following entry points are also used with COMS transactions:

- SIM_CANCEL
- SIM_COMS_ABORT
- SIM_COMS_BEGIN_TRANSACTION
- SIM_COMS_ETR
- SIM_SAVEPOINT

Example

The following ALGOL example demonstrates the use of the SIM_COMS_ETR entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

```
% Declare coms_out array.
OUTPUTHEADER
        COMS_OUT;
BOOLEAN RESULT;

% The code for parsing queries, entering COMS transaction state,
% and opening, processing, and closing cursors should be placed here.
% For the purposes of brevity, the code has been omitted from this
% example.

IF RESULT:= SIM_COMS_ETR (COMS_OUT) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    END;
```

SIM_DESCRIBE_DML_TEXT

Purpose

Use the SIM_DESCRIBE_DML_TEXT entry point to write out the text of a query.

Parameters

query_info

Type: Record

Allowable Values	Description
Same record area as that passed to the query parsing entry points	Identifies the query for which you want the query text. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

qtext

Type: String

Allowable Values	Description
System-supplied	Provides the SIM OML or SQLDB DML associated with the QUERY_INFO record.

Related Entry Points

The following entry points are also used to gather query information:

- SIM_DESCRIBE_ITEM
- SIM_DESCRIBE_QUERY
- SIM_DESCRIBE_PARAMETER
- SIM_DESCRIBE_TEMPLATE

The following entry points are used to parse a query:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL example demonstrates the use of the SIM_DESCRIBE_DML_TEXT entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is already declared and it is assumed to contain information from a previous call to any of the query parsing entry points.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare qtext array.
ARRAY QTEXT [0: TEXT_SZ];
BOOLEAN RESULT;
DEFINE TEXT_SZ        = 30 #;       % Length of text for this example.

IF RESULT:= SIM_DESCRIBE_DML_TEXT (PASS_ARRAY (QUERY_INFO)
                                  ,PASS_ARRAY (QTEXT)
                                 ) THEN
    BEGIN
    % Some kind of error occurred.
    PARSE_ERROR;
    END
ELSE
    BEGIN
    % Having obtained the query text for the parsed query,
    % display the text.
    DISPLAY (STRING (POINTER(QTEXT [1]),QTEXT [0])));
    END;
```

SIM_DESCRIBE_FORMATS

Purpose

Use the SIM_DESCRIBE_FORMATS entry point to obtain the query description record.

Parameters

query_info

Type: Record

Allowable Values	Description
System-supplied	Identifies the query for which you want the query text. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

query_desc

Type: Record

Allowable Values	Description
System-supplied	Provides the record layout for the templates defined by the query. Refer to the EXAMPLE/SIM/ALGOL/ API/RECORDATATIME sample program on the release media for an explanation of how to use and decipher the QUERY_DESC record. Refer to “Using the QUERY_DESC Record” in Section 7, “Using the SIM Library Entry Points,” for additional information on the layout of the QUERY_DESC record.

Related Entry Points

The following entry point is also used to gather query information:

- SIM_PARSE_FORMATTED_QUERY

The following entry points are used to parse a query:

- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL example demonstrates the use of the SIM_DESCRIBE_FORMATS entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is already declared and it is assumed to contain information from a previous call to any of the query parsing entry points.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

Note: *The code for parsing or loading queries has been omitted from this example. Refer to “SIM_LOAD_QUERY” later in this section for the applicable code.*

```
% Declare query_desc and query_info arrays.
ARRAY
   QUERY_DESC [0:QUERY_DESC_SZ]
  ,QUERY_INFO [0:PO_SZ]
  ;
BOOLEAN
   RESULT
  ,FORMAT_QUERY
  ;

%Insert the code for parsing or loading queries here.

WHILE NOT FORMAT_QUERY DO
   BEGIN
   IF RESULT := SIM_DESCRIBE_FORMATS (PASS_ARRAY (QUERY_INFO)
                                     ,PASS_ARRAY (QUERY_DESC)
                                     ) THEN
      BEGIN
      % Some kind of error occurred while describing the format.
      IF EX_CATEGORY (RESULT) = EX+FAILED AND
         EX_SUBCATEGORY (RESULT) = EX_FAIL_FORMATS_TOO_SMALL THEN
```

```
            BEGIN
            %Resize the query_desc array.
            RESIZE
              (QUERY_DESC, SIZE(QUERY_DESC) + QUERY_DESC_SZ, DISCARD);
            END
         ELSE
            BEGIN
            % Some kind of error occurred.
            PROCESS_ERROR;
            GO END_OF_EXAMPLE;
            END;
         END
      ELSE
         FORMAT_QUERY := TRUE;
      END;

   END_OF_EXAMPLE;
```

SIM_DESCRIBE_ITEM

Purpose

Use the SIM_DESCRIBE_ITEM entry point to retrieve item level information for the designated query. Refer to “Gathering Information about a Query” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

query_info

Type: Record

Allowable Values	Description
Same record area as that passed to any of the query parsing entry points	Identifies the query about which you want information. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

item_info

Type: Record

Allowable Values	Description
System-supplied	Describes one of the attributes retrieved by a query. The description of the attribute includes its type, length, scale, and precision (for numbers), and the position in the target list of the query text. Refer to “Using the ITEM_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

item_name

Type: String

Allowable Values	Description
System-supplied	Returns the name of the item described by the information returned in the item_info parameter.

Related Entry Points

The following entry points are also used for gathering query information:

- SIM_DESCRIBE_DML_TEXT
- SIM_DESCRIBE_PARAMETER
- SIM_DESCRIBE_QUERY
- SIM_DESCRIBE_TEMPLATE

The following entry points are used to parse a query:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL example demonstrates the use of the SIM_DESCRIBE_ITEM entry point.

In this example, the QUERY_INFO array is declared already and it is assumed to contain information from a previous call to any of the query parsing entry points.

Also, it is assumed the QUERY_RECORD and TEMPLATE_INFO arrays are declared already.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare item_info and item_name arrays.
  ARRAY
      ITEM_INFO [0: SIM_ITEM_RECORD_SIZE]
    ,ITEM_NAME [0:NAME_SZ]
   ;
  BOOLEAN RESULT;
  INTEGER
      TEMPLATE_NO
    ,NUMBER_OF_ITEMS
   ;
  DEFINE NAME_SZ         = 20 #;      % Size of names for this example

  % A call to SIM_DESCRIBE_QUERY is made at this point. For the
  % purposes of brevity, this code has been omitted.
```

```
IF SIM_Q_STATEMENT (QUERY_RECORD) = ST_RETRIEVE_V THEN
    BEGIN
    % The code to look at templates and items is only
    % useful for retrieval queries. Queries that update the
    % database do not return data.

    TEMPLATE_NO:= 1;
    SIM_QT_TEMPLATE_NUMBER (TEMPLATE_INFO):= TEMPLATE_NO;
    WHILE NOT RESULT:= SIM_DESCRIBE_TEMPLATE
                            (PASS_ARRAY (QUERY_INFO)
                           ,PASS_ARRAY (TEMPLATE_INFO)
                          ) DO
        BEGIN
        NUMBER_OF_ITEMS:=
            SIM_QT_ITEM_COUNT (TEMPLATE_INFO);
        FOR ITEM_INDEX:= 1 STEP 1 UNTIL NUMBER_OF_ITEMS DO
            BEGIN
            % Initialize the item_name array.
            REPLACE POINTER (ITEM_NAME [1]) BY 48"404040404040" FOR
               SIZE (ITEM_NAME) - 1 WORDS;
            ITEM_NAME [0]:= 0;

            SIM_ITEM_NUMBER (ITEM_INFO):=ITEM_INDEX;
            SIM_ITEM_TEMPLATE (ITEM_INFO):= TEMPLATE_NO;
            IF RESULT:= SIM_DESCRIBE_ITEM (PASS_ARRAY (QUERY_INFO)
                                          ,PASS_ARRAY (ITEM_INFO)
                                          ,PASS-ARRAY (ITEM_NAME)
                                         ) THEN
                BEGIN
                % Some kind of error occurred.
                PROCESS_ERROR;
                END
            ELSE
                BEGIN
                % For the purposes of brevity, code for using the
                % item_info record has been omitted.
                END;
            END;

        IF RESULT THEN
           BEGIN
           % Some kind of error occurred.
           PROCESS_ERROR;
           GO END_OF_EXAMPLE;
           END;

        TEMPLATE_NO:= * + 1;
        SIM_QT_TEMPLATE_NUMBER (TEMPLATE_INFO):=
            TEMPLATE_NO;
        END;
    END;
END_OF_EXAMPLE:
```

SIM_DESCRIBE_PARAMETER

Purpose

Use the SIM_DESCRIBE_PARAMETER entry point to identify the data type associated with a query parameter. Refer to “Using Parameters in a Query” in Section 7, “Using the SIM Library Entry Points,” for more information.

Table 7–1 explains the correlation between the numeric value returned by the item type parameter, the name given to the item type in the SYMBOL/SIM/PROPERTIES file, and the name of the item type.

Note: *When you supply a query parameter for a SIM OML query, you must preface the identifier with a number sign (#). However, when you supply a query parameter for an SQLDB DML query, you can preface the identifier either with a number sign (#) or with a colon (:).*

Parameters

query_info

Type: Record

Allowable Values	Description
Same record areas as that passed to any of the query parsing entry points	The query_info parameter is an input parameter. The value you pass to SIM is the value SIM associates with the query_info parameter when you use either the SIM_PARSE_QUERY or the SIM_PARSE_SQL entry point. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

item_info

Type: Record

Allowable Values	Description

System-supplied	Describes the number, type, subtype, sign, length, scale, and precision of the parameter. The item_info parameter is an input/output parameter—you must supply the parameter number; SIM returns the other information.
	
	You determine the parameter number by its position in the query text. For example, the following retrieval query has three different parameters. (Do not count multiple occurrences of the same parameter.) If you want to return information about the parameter #AGE, you must supply a parameter number of 1. The same value is used for #AGE when you compare both AGE and AGE OF SPOUSE.
	
	`FROM EMPLOYEE`
	`RETRIEVE NAME`
	`WHERE AGE > #AGE`
	`AND PERSON_ID > #ID`
	`AND AGE OF SPOUSE > #AGE`
	`AND SALARY > #SALARY`

Related Entry Points

The following entry point is used to associate a value with a query parameter:

- SIM_PUT_VALUE

The following entry points are used to parse a query:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL example demonstrates the use of the SIM_DESCRIBE_PARAMETER entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is already declared and it is assumed to contain information from a previous call to any of the query parsing entry points.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare item_info array.
ARRAY ITEM_INFO [0: SIM_ITEM_RECORD_SIZE];
BOOLEAN
    RESULT
  ,DONE
 ;
INTEGER PARAMETER_NUMBER;

FOR PARAMETER_NUMBER:= 1 STEP 1 WHILE NOT DONE DO
    BEGIN
    SIM_ITEM_NUMBER (ITEM_INFO):= PARAMETER_NUMBER;
    IF RESULT:= SIM_DESCRIBE_PARAMETER (PASS_ARRAY (QUERY_INFO)
                                       ,PASS_ARRAY (ITEM_INFO)
                                      ) THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        DONE:= TRUE;
        END
    ELSE
        BEGIN
        % For the purposes of brevity, the code for using the
        % ITEM_INFO record has been omitted. For more
        % information, refer to the example given under
        % "SIM_PUT_VALUE" later in this section.
        END;
    END;
```

SIM_DESCRIBE_QUERY

Purpose

Use the SIM_DESCRIBE_QUERY entry point to return query-level information for the designated query. For more information, refer to “Gathering Information about a Query” in Section 7, “Using the SIM Library Entry Points.”

Parameters

query_info

Type: Record

Allowable Values	Description
The same record area as that passed to any of the query parsing entry points	Identifies the query about which you want information. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

query_record

Type: Record

Allowable Values	Description
System-supplied	Describes such things as the type of query (insert, modify, retrieval, or delete), whether the output is tabular or structured, the number of templates (for structured output only), and if parameter values are required. Refer to “Using the QUERY_RECORD Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

Related Entry Points

The following entry points are also used to gather information about a query:

- SIM_DESCRIBE_DML_TEXT
- SIM_DESCRIBE_ITEM
- SIM_DESCRIBE_PARAMETER
- SIM_DESCRIBE_TEMPLATE

The following entry points are used to parse a query:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL example demonstrates the use of the SIM_DESCRIBE_QUERY entry point. In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is declared already and it is assumed to contain information from a previous call to any of the query parsing entry points.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare query_desc array.
ARRAY QUERY_RECORD [0: SIM_Q_SIZE];
BOOLEAN
    RESULT
 ;

% For the purposes of brevity, the code for calling any of
% the query parsing entry points has been omitted.

IF RESULT:= SIM_DESCRIBE_QUERY (PASS_ARRAY (QUERY_INFO)
                               ,PASS_ARRAY (QUERY_RECORD)
                              ) THEN
   BEGIN
   % Some kind of error occurred.
     PROCESS_ERROR;
     GO END_OF_EXAMPLE;
   END;

% To obtain query type.
IF SIM_Q_STATEMENT (QUERY_RECORD) = ST_RETRIEVE_V THEN
  BEGIN
  END;

% To check if the output is tabular or structured.
IF SIM_Q_TABULAR (QUERY_RECORD) THEN
  BEGIN
  END;

END_OF_EXAMPLE:
```

SIM_DESCRIBE_TEMPLATE

Purpose

Use the SIM_DESCRIBE_TEMPLATE entry point to return template-level information for the designated query. Refer to “Gathering Information about a Query” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

query_info

Type: Record

Allowable Values	Description
Same record area as that passed to the query parsing entry points	Identifies the query about which you want information. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

template_info

Type: Record

Allowable Values	Description
System-supplied	Provides the information about a particular template produced by the requested query. Information returned includes the template number and level, and the number of attributes whose values are returned at this template level. Refer to “Using the TEMPLATE_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

Related Entry Points

The following entry points are also used to gather information about a query:

- SIM_DESCRIBE_DML_TEXT
- SIM_DESCRIBE_ITEM
- SIM_DESCRIBE_PARAMETER
- SIM_DESCRIBE_QUERY

The following entry points are used to parse a query:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL example demonstrates the use of the SIM_DESCRIBE_TEMPLATE entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is declared already and it is assumed to contain information from a previous call to any of the query parsing entry points.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare query_record and template_info arrays.
  ARRAY
      TEMPLATE_INFO [0: SIM_TEMPLATE_SIZE]
    ,QUERY_RECORD [0: SIM_Q_SIZE]
   ;
  BOOLEAN
      RESULT
   ;

   % For the purposes of brevity, the code for calling any of
   % the query parsing entry points has been omitted.
```

```
IF RESULT:= SIM_DESCRIBE_QUERY (PASS_ARRAY (QUERY_INFO)
                               ,PASS_ARRAY (QUERY_RECORD)
                              ) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    GO END_OF_EXAMPLE;
    END;

IF SIM_Q_STATEMENT (QUERY_RECORD) = ST_RETRIEVE_V THEN
    BEGIN
    % The code for looking at template items is only useful
    % for retrieval queries. Queries that update the database
    % do not return data.

    FOR TEMPLATE_NO:=1 STEP 1 UNTIL SIM_Q_LEVELS (QUERY_RECORD) DO
        BEGIN
        % Set up template_info for entry point call.
        SIM_QT_TEMPLATE_NUMBER (TEMPLATE_INFO):= TEMPLATE_NO;
        IF RESULT:=SIM_DESCRIBE_TEMPLATE
                         (PASS_ARRAY (QUERY_INFO)
                        ,PASS_ARRAY (TEMPLATE_INFO)) THEN
            BEGIN
            % Some kind error occurred calling entry point.
            PROCESS_ERROR;
            END
        ELSE
            BEGIN
            % For the purposes of brevity, the code for using the
            % template information has been omitted. For a
            % more complete example, refer to the example for
            % the SIM_DESCRIBE_ITEM  entry point in this section.
            END;
        END;

END_OF_EXAMPLE:
```

SIM_ETR

Purpose

Use the SIM_ETR entry point to end a transaction. For more information on transactions, refer to "Performing Queries While in Transaction State" in Section 7, "Using the SIM Library Entry Points."

Parameters

There are no parameters associated with the SIM_ETR entry point.

Related Entry Points

The following entry points are also used with transactions:

- SIM_ABORT
- SIM_BEGIN_TRANSACTION
- SIM_CANCEL
- SIM_SAVEPOINT

Example

The following ALGOL example demonstrates the use of the SIM_ETR entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to "Returning Error Messages" in Section 7, "Using the SIM Library Entry Points," for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is declared already and it is assumed to contain information from a previous call to any of the query parsing entry points. Also, it is assumed that the CURSOR_INFO array is declared previously.

PASS_ARRAY is a define used to pass array parameters. Refer to "Passing Arrays to SIM from ALGOL Programs" in Section 7, "Using the SIM Library Entry Points," for an explanation of the PASS_ARRAY define.

```
% Enter transaction state.
IF RESULT:= SIM_BEGIN_TRANSACTION (LEVEL_LOCKING) THEN
   BEGIN
       % There was an error trying to enter transaction
       % state.
    PROCESS_ERROR;
    GO END_OF_EXAMPLE;
    END;
```

```
% The program is now in transaction state. For the purposes of
% brevity, the code for opening, processing, and closing
% a cursor for the compiled query has been omitted.

% End transaction state.
IF RESULT:= SIM_ETR THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    END;

END_OF_EXAMPLE:
```

SIM_EXCEPTION_INFO

Purpose

Use the SIM_EXCEPTION_INFO entry point to return exception message information. Refer to "Returning Error Message Information" in Section 7, "Using the SIM Library Entry Points," for more information. Lists of the exceptions that might be returned are presented in Appendix A, "OML Error Messages," and Appendix B, "Exception Fields and Categories."

Parameters

exception_info

Type: Record

Allowable Values	Description
System-supplied	Provides the following for DMSIISUPPORT errors: • LUC name • Structure name Provides the following for ADDS interface errors: • Dictionary name • Database name input to ADDS Provides the error message text for SIM OML or SQLDB DML syntax errors. Provides the verify name for VERIFY errors. Provides the error code for CENTRALSUPPORT errors. In addition, for all errors the EXCEPTION_INFO record provides the database name and any applicable DMSII DMSTATUS information. Refer to "Using the EXCEPTION_INFO Record" in Section 7, "Using the SIM Library Entry Points," for more information.

Related Entry Points

The following entry points are also used to gather information about exceptions:

- SIM_EXCEPTION_MESSAGE
- SIM_GET_EXCEPTION_INFO
- SIM_NEXT_EXCEPTION

Example

The following ALGOL example demonstrates the use of the SIM_EXCEPTION_INFO entry point.

PASS_ARRAY is a define used to pass array parameters. Refer to "Passing Arrays to SIM from ALGOL Programs" in Section 7, "Using the SIM Library Entry Points," for an explanation of the PASS_ARRAY define.

For more information on error processing, refer to "Returning Error Messages" in Section 7, "Using the SIM Library Entry Points."

```
% Declare except_info array.
ARRAY EXCEPT_INFO [0: EI_SIZE];
BOOLEAN RESULT;

IF RESULT:= SIM_GET_EXCEPTION_INFO (ERROR_CATEGORY
                                    ,ERROR_SUBCATEGORY
                                   ) THEN
    BEGIN
    % See if a DMSIISUPPORT error occurred.
     IF ERROR_CATEGORY = EX_SYSTEM AND
       (ERROR_SUBCATEGORY = EX_SYS_LOGICAL_DB OR
        ERROR_SUBCATEGORY = EX_SYS_PHYSICAL_DB) THEN
        BEGIN
        DO BEGIN
          % I know this information would be displayed in a SIM
          % error message but I want to make my own message.
           RESULT:= SIM_EXCEPTION_INFO (PASS_ARRAY (EXCEPT_INFO));
           DM_STATUS:= EI_DMSTATUS (EXCEPT_INFO);
           REPLACE LUC_NAME BY
              POINTER(EI_LUC_NAME (EXCEPT_INFO)) FOR EI_LUC_NAME_LEN;
           REPLACE DB_NAME BY
              POINTER(EI_DB_NAME (EXCEPT_INFO)) FOR EI_DB_NAME_LEN;
           REPLACE DMSII_STR_NAME BY
              POINTER(EI_STR_NAME (EXCEPT_INFO)) FOR EI_STR_NAME_LEN;
           END
        UNTIL NOT RESULT:= SIM_NEXT_EXCEPTION;
        END;
    END;
END_OF_EXAMPLE:
```

SIM_EXCEPTION_MESSAGE

Purpose

Use the SIM_EXCEPTION_MESSAGE entry point to return exception message information. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for more information. Appendix A, “OML Error Messages,” and Appendix B, “Exception Fields and Categories,” list the exceptions you might receive using this entry point.

Parameters

lang

Type: String

Allowable Values	Description
Language name, for example, French or Spanish	Designates the language in which the message is returned. The first word (48 bits) in the array defines the length of the text. The lang parameter is an input parameter.

message_array

Type: String

Allowable Values	Description
System-supplied	Returns the text of the error message. The first word (48 bits) in the array defines the length of the text. The message_array parameter is an output parameter.

Related Entry Points

The following entry points are also used to gather information about exceptions:

- SIM_EXCEPTION_INFO
- SIM_GET_EXCEPTION_INFO
- SIM_NEXT_EXCEPTION

Example

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

For more information on error processing, refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points.”

```
% Declare lang and message_array arrays.
ARRAY
    LANG
  ,MESSAGE_ARRAY [0: STR_SZ]
 ;
BOOLEAN RESULT;
INTEGER MLS_RESULT;
DEFINE STR_SZ      = 200 #;     % Maximum size used in this example.

% For the purposes of brevity the code using the SIM_GET_EXCEPTION_INFO
% has been omitted. It is assumed the call is successful.

% Set up language used.
REPLACE POINTER (LANG [1]) BY "PORTUGUESE";
LANG [0]:= 10;

DO BEGIN
  % Here we display any error message supplied by SIM.
   MLS_RESULT:= SIM_EXCEPTION_MESSAGE (PASS_ARRAY (LANG)
                                      ,PASS_ARRAY (MESSAGE_ARRAY)
                                     );
   IF MLS_RESULT LSS 0 THEN
       BEGIN
       % Some kind of error occurred obtaining the
       % error message.
       END
   ELSE
       BEGIN
       % Got the error message. If SIM_EXCEPTION_MESSAGE entry
       % point returns a value of 1, the exception message could
       % not be translated because the requested language could
       % not be found. Instead, the message is displayed in the
       % user's default language.
       END;
   END
UNTIL NOT RESULT:= SIM_NEXT_EXCEPTION;
```

SIM_GET_EXCEPTION_INFO

Purpose

Use the SIM_GET_EXCEPTION_INFO entry point to return exception message information. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for more information. Appendix A, “OML Error Messages,” and Appendix B, “Exception Fields and Categories,” list the exceptions you might receive when using this entry point.

Parameters

category

Type: Integer

Description: Returns the category of the current error.

subcategory

Type: Integer

Description: Returns the subcategory of the current error.

Related Entry Points

The following entry points are also used to gather information about exceptions:

- SIM_EXCEPTION_INFO
- SIM_EXCEPTION_MESSAGE
- SIM_NEXT_EXCEPTION

Example

The following ALGOL example demonstrates the use of the SIM_GET_EXCEPTION_INFO entry point.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

For more information on error processing, refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points.”

```
% Declare error category and subcategory
INTEGER
    ERROR_CATEGORY
  ,ERROR_SUBCATEGORY
 ;
BOOLEAN RESULT;

IF RESULT:= SIM_GET_EXCEPTION_INFO (ERROR_CATEGORY
                                   ,ERROR_SUBCATEGORY
                                  ) THEN
    BEGIN
    % Check for a DMSIISUPPORT error.
    IF ERROR_CATEGORY = EX_SYSTEM AND
       (ERROR_SUBCATEGORY = EX_SYS_LOGICAL_DB OR
        ERROR_SUBCATEGORY = EX_SYS_PHYSICAL_DB) THEN
        BEGIN
        % For the purposes of brevity, the code for obtaining more error
        % information has been omitted.
        END;
    END;
```

SIM_GET_VALUE

Purpose

Use the SIM_GET_VALUE entry point to retrieve one piece of data from the database. Refer to “Retrieving Query Output” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

cursor_info

Type: Record

Allowable Values	Description
Same record area as that passed to the SIM_OPEN_CURSOR entry point	Identifies the cursor associated with the query. The cursor_info parameter is an input parameter. The value you pass to SIM is the value SIM associates with the cursor_info parameter when you use the SIM_OPEN_CURSOR entry point. Refer to “Using the CURSOR_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

item_info

Type: Record

Allowable Values	Description
In the ITEM_NUMBER field, 1 to the number of items in the query target list	Identifies the item about which you want information. The item_info parameter is an input parameter. For more information, refer to “Using the ITEM_INFO Record” in Section 7, “Using the SIM Library Entry Points.”

item_value

Type: String

Allowable Values	Description
System-supplied	Returns the item value as a string. The item_value parameter is an output parameter.

numeric_value

Type: Double

Allowable Values	Description
System-supplied	Returns the item value as a number, if applicable. The numeric_value parameter is an output parameter. If the value is normally of type string, a 0 (zero) is returned. If the item is of type Boolean, a 0 (zero) indicates a false value and a 1 indicates a true value. If the item has a null value, an error of category 1, subcategory 14 is returned.

Related Entry Points

The following entry point is also used to extract data from the database:

- SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA

Example 1

The following ALGOL procedure uses the entry point SIM_GET_VALUE to return the values associated with the items about which you request information in your query.

In addition, the procedure uses the field SIM_ITEM_NUMBER to check for the number of the item in the target list and the entry point SIM_DESCRIBE_ITEM to return the names of the items. The entry point SIM_DESCRIBE_ITEM returns the names of the items only for the first tuple at any given template level.

PUTOUT and PUTOUTBUFF are user-written procedures for storing and displaying the data returned by the query.

```
PROCEDURE RETR_VALUE;
    BEGIN
    SIM_ITEM_NUMBER(ITEM_INFO):= 1;
    WHILE NOT RSLT:=
        SIM_GET_VALUE(
            P(CURSOR_INFO),
            P(ITEM_INFO),
            P(ITEM_VALUE),
            NUMERIC_VALUE) DO
            BEGIN
            IF  FIRST_ENTITY.[IC_LEVEL_NO(CURSOR_INFO):1] THEN
```

```
                BEGIN
                RSLT:= SIM_DESCRIBE_ITEM(
                    P(QUERY_INFO), P(ITEM_INFO), P(ITEM_ID));
                END
            ELSE
                ITEM_ID[0]:= 0;
            PUTOUT;
            SIM_ITEM_NUMBER(ITEM_INFO):= *+1;
            END;
    FIRST_ENTITY.[0:1]:= FALSE;
    FIRST_ENTITY.[IC_LEVEL_NO(IC):1]:= FALSE;
    PUTOUTBUFF;
    IF  REAL(RSLT).EX_CATEGORYF NEQ EX_COMPLETE AND
        REAL(RSLT).EX_CATEGORYF NEQ EX_NONE THEN
        EXIT("Get Value");
    END OF RETR_VALUE;
```

Example 2

In the following example, the QUERY_INFO array is declared already and it is assumed to contain information from a previous call to the SIM_PARSE_QUERY entry point. Also, the CURSOR_INFO array has been declared already and contains information from a call to the SIM_OPEN_CURSOR entry point.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare item_info, item_value arrays.
  ARRAY
      ITEM_INFO [0: SIM_ITEM_RECORD_SIZE]
    ,ITEM_VALUE
    ,ITEM_NAME [0: VALUE_SZ]
   ;
  BOOLEAN
      RESULT
    ,DONE
    ,NULL_VALUE
   ;
  DOUBLE NUMERIC_VALUE;
  INTEGER ITEM_NUMBER;
  DEFINE VALUE_SZ        = 30 #;  % Size of strings for this example.
```

```
% For the purposes of brevity, the code for parsing the query and
% processing the query has been omitted.

% Start at first item in query.
ITEM_NUMBER:= 1;
SIM_ITEM_NUMBER (ITEM_INFO):= ITEM_NUMBER;

DONE:= FALSE;
WHILE NOT DONE DO
    BEGIN
    RESULT:= SIM_GET_VALUE (PASS_ARRAY (CURSOR_INFO)
                           ,PASS_ARRAY (ITEM_INFO)
                           ,PASS_ARRAY (ITEM_VALUE)
                           ,NUMERIC_VALUE
                          );

    IF EX_CATEGORY (RESULT) = EX_WARNING AND
       EX_SUBCATEGORY (RESULT) = EX_WARN_NULL_ITEM THEN
        BEGIN
        % The value is null.
        NULL_VALUE:=TRUE;
        END
    ELSE
    IF EX_CATEGORY (RESULT) NEQ EX_COMPLETE AND
       EX_CATEGORY (RESULT) NEQ EX_NONE THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        GO_END_OF_EXAMPLE;
        END
    ELSE
        NULL_VALUE:= FALSE;

    % Get item_info and item_name.
    REPLACE POINTER (ITEM_NAME [1]) BY
       48"404040404040" FOR SIZE (ITEM_NAME) - 1 WORDS;
    ITEM_NAME [0]:= 0;
    IF RESULT:= SIM_DESCRIBE_ITEM (PASS_ARRAY (QUERY_INFO)
                                  ,PASS_ARRAY (ITEM_INFO)
                                  ,PASS_ARRAY (ITEM_NAME)
                                 ) THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        GO END_OF_EXAMPLE;
        END;
    IF NOT NULL_VALUE THEN
        BEGIN
        CASE SIM_ITEM_TYPE (ITEM_INFO) OF
            BEGIN
            (ITEM_IS_BOOLEAN_V):
                IF NUMERIC_VALUE = 0 THEN
                    % Item is FALSE.
                ELSE
                    % Item value is TRUE.

            (ITEM_IS_REAL_V):
                % Value is in numeric_value.

            (ITEM_IS_STRING_V):
                % Value is in item_value.

            ELSE:
              ;
            END;
        END;

    ITEM_NUMBER:= * + 1;
    SIM_ITEM_NUMBER (ITEM_INFO):= ITEM_NUMBER;
    END;

END_OF_EXAMPLE:
```

SIM_LOAD_QUERY

Purpose

Use the SIM_LOAD_QUERY entry point to restore query information from a saved file.

You can use the SIM_LOAD_QUERY entry point only before you open any databases or after you have closed all the databases you had opened. If you use the SIM_LOAD_QUERY entry point while you still have an open database, an error is returned. Refer to “Saving and Restoring Queries” in Section 7, “Using the SIM Library Entry Points,” for more information.

Parameters

file_title

Type: String

Allowable Values	Description
A valid A Series file title	Designates the title of the file containing the saved query information. The file_title parameter is an input parameter. The first word of the string must contain the length of the file title.

query_info

Type: Record

Allowable Values	Description
System-supplied	Provides the query numbers located in the file. The parameter is an array of query information (QUERY_INFO) records. In your program, you must examine either the PO_USER_QUERY_NO or the PO_ID fields to determine the significance of each query. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

Notes:

- *You must ensure that the array you declare for the query_info parameter is large enough to contain all information for all queries in the saved query file.*
- *Using the SIM_LOAD_QUERY entry point invalidates all the information that might have existed in the query_info parameter before you used the SIM_LOAD_QUERY entry point.*

Related Entry Points

The following entry points are also used to save and restore queries:

- SIM_ADD_QUERY
- SIM_SAVE_QUERY

Example

The following ALGOL example demonstrates the use of the SIM_LOAD_QUERY entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare file_title and query_info arrays.
  ARRAY
      FILE_TITLE [0:255]
    ,QUERY_INFO [0:PO_SZ]
   ;
  BOOLEAN
      RESULT
    ,QUERY_FILE_LOADED;
  POINTER PTR;

  % Set up file title.
  REPLACE PTR:POINTER(FILE_TITLE [1]) BY "SAVED/QUERIES";
  FILE_TITLE [0]:= OFFSET (PTR) - 6;
```

```
QUERY_FILE_LOADED:= FALSE;
DO BEGIN
   IF RESULT:= SIM_LOAD_QUERY (PASS_ARRAY (FILE_TITLE)
                              ,PASS_ARRAY (QUERY_INFO)
                             ) THEN
       BEGIN
       IF EX_CATEGORY (RESULT) = EX_FAILED AND
          EX_SUBCATEGORY (RESULT) = EX_FAIL_QID_INFO_TOO_SMALL THEN
           BEGIN
           % Query_info array too small. Make the array larger.
           RESIZE (QUERY_INFO, SIZE(QUERY_INFO)+PO_SZ, DISCARD);
           END
       ELSE
           BEGIN
           % Some kind of error occurred.
           PROCESS_ERROR;
           GO END_OF_EXAMPLE;
           END
       END
   ELSE
       QUERY_FILE_LOADED:= TRUE;
   END
UNTIL QUERY_FILE_LOADED;

END_OF_EXAMPLE:
```

SIM_NEXT_EXCEPTION

Purpose

Use the SIM_NEXT_EXCEPTION entry point to identify whether more exception messages are available. If more exception messages are available, the next exception message becomes the current exception message. The information you can extract using the SIM_GET_EXCEPTION_INFO and SIM_EXCEPTION_MESSAGE entry points always relates to the current exception message.

The Boolean value returned by the SIM_NEXT_EXCEPTION entry point is a 48-bit word that contains the category and subcategory numbers associated with the exception. To extract the category and subcategory information, you can break down the 48-bit word by using the partial word fields EX_CATEGORYF and EX_SUBCATEGORYF. Appendix B, “Exception Fields and Categories,” lists and explains the category and subcategory information that can be returned.

For more information, refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points.”

Parameters

There are no parameters associated with the SIM_NEXT_EXCEPTION entry point.

Related Entry Points

The following entry points are also used to return exception messages:

- SIM_EXCEPTION_INFO
- SIM_EXCEPTION_MESSAGE
- SIM_GET_EXCEPTION_INFO

Example

The following ALGOL example demonstrates the use of the SIM_NEXT_EXCEPTION entry point.

This example illustrates how to obtain additional information about any error you might encounter.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

For more information on error processing, refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points.”

```
% Declare category and subcategory.
INTEGER
    ERROR_CATEGORY
  ,ERROR_SUBCATEGORY
 ;
BOOLEAN RESULT;

IF RESULT:= SIM_GET_EXCEPTION_INFO (ERROR_CATEGORY
                                   ,ERROR_SUBCATEGORY
                                  ) THEN
    BEGIN

% At this point the first group of exception information is
% now available.

DO BEGIN
   % For the purposes of brevity, the obtaining of error
   % messages or specific error information has been omitted.
   END
UNTIL NOT RESULT:= SIM_NEXT_EXCEPTION;

END;

END_OF_EXAMPLE:
```

SIM_OPEN_CURSOR

Purpose

Use the SIM_OPEN_CURSOR to identify an instance of a query. Refer to "Associating a Cursor with a Compiled Query" in Section 7, "Using the SIM Library Entry Points," for more information.

Parameters

query_info

Type: Record

Allowable Values	Description
Same record area as that passed to the query parsing entry points	Denotes the query identifier. The query_info parameter is an input parameter. Refer to "Using the QUERY_INFO Record" in Section 7, "Using the SIM Library Entry Points," for more information.

cursor_info

Type: Record

Allowable Values	Description
System-supplied	Identifies the query cursor. The cursor_info parameter is an output parameter. SIM returns a unique identifier for each cursor you open. When you use the other SIM library entry points, you must use this value to identify your cursor. Refer to "Using the CURSOR_INFO Record" in Section 7, "Using the SIM Library Entry Points," for more information.

Related Entry Points

The following entry points are also used for cursor operations:

- SIM_CLOSE_CURSOR
- SIM_PROCESS_CURSOR
- SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA
- SIM_PROCESS_CURSOR_AND_UPDATE_DATA

The following entry points are used to parse a query:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL procedure calls the SIM_OPEN_CURSOR entry point. If the cursor is successfully opened, the message “Cursor opened” is displayed.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO array is declared outside of the procedure. The structure is assumed to contain information from a previous call using the SIM_PARSE_QUERY entry point.

The CURSOR_INFO array is also declared outside of the procedure. The information which SIM places into the CURSOR_INFO array is needed to complete the query; declaring CURSOR_INFO outside of the procedure ensures that the information is retained after the procedure exits.

PASS_ARRAY is a define used when passing array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% The defines for PO_SIZE and IC_CURSOR_SIZE are declared in
  % the file SYMBOL/SIM/PROPERTIES.
         ARRAY
             QUERY_INFO  [0:PO_SIZE],
             CURSOR_INFO [0:IC_CURSOR_SIZE];

         PROCEDURE OPEN_CURSOR;
             BEGIN
             BOOLEAN
                 RESULT;             % Holds the SIM result word.

             % First, initialize the CURSOR_INFO record.
             REPLACE CURSOR_INFO BY 0 FOR SIZE (CURSOR_INFO) WORDS;
```

```
IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO)
                            ,PASS_ARRAY (CURSOR_INFO)
                           ) THEN
    %  Some kind of error occurred.
    PROCESS_ERROR
ELSE
    % No errors occurred.
    DISPLAY ("Cursor opened");
END;
```

SIM_OPEN_DATABASE

Purpose

Use the SIM_OPEN_DATABASE entry point to open a database or a flat file.

Parameters

open_options

Type: Record

Allowable Values	Description
Refer to “Using the OPEN_OPTIONS Record” in Section 7, “Using the SIM Library Entry Points.”	Identifies one of the three categories of open database or file options, as follows: • Mode The two modes in which you can open a database or flat file are update (0) and inquiry (5). • Version The version is the version number of the database or flat file; a value of 0 (zero) means use the most current version. • Status The values of status for a database or flat file are production (3), test (2), and not specified (0).

dbname_text

Type: String

Allowable Values	Description
Name of any existing database or flat file.	Names the database or flat file you want to use.

usercode_text

Type: String

Allowable Values	Description
Any valid usercode.	Provides the usercode associated with the database, flat file, or the ADDS directory.

family_text

Type: String

Allowable Values	Description
Any valid family name.	Names the family on which your database or flat file is stored.

Related Entry Points

The following entry points are also used to open and close databases and flat files:

- SIM_CLOSE_DATABASE
- SIM_SET_DICTIONARY

Example

The following example ALGOL procedure assumes that a user has entered the name of a database and that the name is currently stored in the string variable DATABASE_NAME. The example also assumes that the arrays USERCODE_TEXT and FAMILY_TEXT have been declared previously.

If the database is successfully opened, the message “Database opened” is displayed. If an error occurs, for instance because the database was already open or because the database could not be located, the procedure PROCESS_ERROR is called. (The procedure PROCESS_ERROR is explained under “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points.”)

PASS_ARRAY is a define used when passing array parameters. Refer to “Passing Arrays to SIM from ALGOL programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
BOOLEAN RSLT;
PROCEDURE OPEN_DATABASE_EXAMPLE;
    BEGIN
    REPLACE POINTER(DBNAME_TEXT[1]) BY DATABASE_NAME FOR LENGTH
        (DATABASE_NAME);
    DBNAME_TEXT[0]:= LENGTH (DATABASE_NAME);
    IF  RSLT:= SIM_OPEN_DATABASE(PASS_ARRAY (OPEN_OPTIONS)
                                ,PASS_ARRAY (DBNAME_TEXT)
                                ,PASS_ARRAY (USERCODE_TEXT)
                                ,PASS_ARRAY (FAMILY_TEXT)
                               )THEN
        PROCESS_ERROR  % Uses SIM_EXCEPTION_MESSAGE entry point.
    ELSE
        DISPLAY ("DATABASE OPENED");
    END;
```

SIM_PARSE_FORMATTED_QUERY

Purpose

Use the SIM_PARSE_FORMATTED_QUERY entry point to parse a SIM OML query and to designate data formats for items. Refer to “Reading In the Query Text” in Section 7, “Using the SIM Library Entry Points,” for more information.

Notes:

- *When you supply a query parameter for a SIM OML query, you must preface the identifier with a number sign (#). However, when you supply a query parameter for an SQLDB DML query, you can preface the identifier either with a number sign (#) or with a colon (:).*
- *This entry point is not defined for COBOL.*

Parameters

query_info

Type: Record

Allowable Values	Description
System-supplied	Tells SIM how to format certain data types. For example, attributes of type date can be returned as real numbers. After calling the SIM_PARSE_FORMATTED_QUERY entry point, the QUERY_INFO record contains additional information which uniquely identifies the query to SIM. This information is required later when you execute the query, so you should not alter this record until after you close the query using the SIM_CLOSE_QUERY entry point. Therefore, your program must allocate one QUERY_INFO record for each query that has been parsed but not closed. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

qtext

Type: String

Allowable Values	Description
Any SIM OML query text	Passes the query text to the SIM library. The first word of the string must define the length of the query text.

query_desc

Type: Record

Allowable Values	Description
System-supplied	Provides the record layout for the templates defined by the query. Refer to the EXAMPLE/SIM/ALGOL/ API/RECORDATATIME sample program on the software release media for an explanation of how to use and decipher the QUERY_DESC record. Refer to “Using the QUERY_DESC Record” in Section 7, “Using the SIM Library Entry Points,” for additional information on the layout of the QUERY_DESC record.

Related Entry Points

The following entry points are also used to parse queries:

- SIM_PARSE_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL example demonstrates the use of the SIM_PARSE_FORMATTED_QUERY entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare query_info, qtext, and query_desc arrays.
  ARRAY
      QUERY_INFO [0:PO_SZ]
    ,QTEXT [0:255]
    ,QUERY_DESC [0:QUERY_DESC_SZ]
   ;
 BOOLEAN
      RESULT
    ,FORMAT_QUERY
   ;
  POINTER PTR;
  DEFINE
      QUERY_DESC_SZ  = 1023 #;   % Initial size for query_desc array.
```

```
% Set up qtext array.
REPLACE PTR:POINTER (QTEXT [1]) BY
   "FROM MYCLASS RET MYDVA WHERE MYDVA EQL ", """, "MYSELF", """;
QTEXT [0]:= DELTA (POINTER (QTEXT [1]), PTR);

% This initial call builds the query_info and query_desc records.
FORMAT_QUERY:= FALSE;
WHILE NOT FORMAT_QUERY DO
    BEGIN
    IF RESULT:= SIM_PARSE_FORMATTED_QUERY (PASS_ARRAY (QUERY_INFO)
                                          ,PASS_ARRAY (QTEXT)
                                          ,PASS_ARRAY (QUERY_DESC)
                                         ) THEN
        BEGIN
        % Some kind of error occurred while parsing the query.
        IF EX_CATEGORY (RESULT) = EX_FAILED AND
           EX_SUBCATEGORY (RESULT) = EX_FAIL_FORMATS_TOO_SMALL THEN
            BEGIN
            % Resize  the query_desc array.
            RESIZE
             (QUERY_DESC, SIZE(QUERY_DESC)+QUERY_DESC_SZ,DISCARD);
            END
        ELSE
            BEGIN
            % Some kind of error occurred.
            PROCESS_ERROR;
            GO END_OF_EXAMPLE;
            END;
        END
    ELSE
        FORMAT_QUERY:= TRUE;
    END;

% Format the output.
PO_USER_FORMATTED (QUERY_INFO):= 1;
DESC_TABULAR (QUERY_DESC):= TRUE;
% Reparse the query again.
IF RESULT:= SIM_PARSE_FORMATTED_QUERY (PASS_ARRAY (QUERY_INFO)
                                      ,PASS_ARRAY (QTEXT)
                                      ,PASS_ARRAY (QUERY_DESC)
                                     ) THEN
    BEGIN
    % Some kind of error occurred while parsing the query.
    PROCESS_ERROR;
    END
ELSE
    BEGIN
    % Query is successfully parsed and is ready to be processed.
    % For the purposes of brevity, this code has been omitted.
    END;

END_OF_EXAMPLE:
```

SIM_PARSE_QUERY

Purpose

Use the SIM_PARSE_QUERY entry point to parse a SIM OML query. Refer to “Reading In the Query Text” in Section 7, “Using the SIM Library Entry Points,” for more information.

Note: *When you supply a query parameter for a SIM OML query, you must preface the identifier with a number sign (#). However, when you supply a query parameter for an SQLDB DML query, you can preface the identifier either with a number sign (#) or with a colon (:).*

Parameters

query_info

Type: Record

Allowable Values	Description
System-supplied	Tells SIM how to format certain data types. For example, attributes of type date can be returned as real numbers. After calling the SIM_PARSE_QUERY entry point, the QUERY_INFO record contains additional information which uniquely identifies the query to SIM. This information is required later when you execute the query, so you should not alter this record until after you close the query using the SIM_CLOSE_QUERY entry point. Therefore, your program must allocate one QUERY_INFO record for each query that has been parsed but not closed. Refer to “Using the QUERY_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

qtext

Type: String

Allowable Values	Description
Any SIM OML query text	Passes the query text to the SIM library. The first word of the string must define the length of the query text.

Related Entry Points

The following entry points are also used to parse queries:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_SQL

Example

The following ALGOL procedure calls the SIM_PARSE_QUERY entry point.

If the query is syntactically correct, the message "Query parsed" is displayed. If the query cannot be parsed, a result with a category of EX_FAILED and a subcategory of EX_FAIL_DML_SYNTAX indicates the OML of the query is syntactically incorrect, and the procedure displays the message "OML syntax error." If any other type of error occurs, the procedure PROCESS_ERROR is called. For an explanation of the PROCESS_ERROR procedure, refer to "Returning Error Messages" in Section 7, "Using the SIM Library Entry Points."

In this example, the QUERY_INFO array is declared outside of the procedure. The information which SIM places into the QUERY_INFO array is needed to execute the query. Declaring the QUERY_INFO array outside of the procedure, ensures that the information is retained after the procedure exits.

PASS_ARRAY is a define used when passing array parameters. Refer to "Passing Arrays to SIM from ALGOL Programs" in Section 7, "Using the SIM Library Entry Points," for an explanation of the PASS_ARRAY define.

```
% Declare query_info array
   ARRAY
      QUERY_INFO [0:PO_SIZE];   % PO_SIZE is defined in
                                % the SYMBOL/SIM/PROPERTIES file.
   PROCEDURE PARSE_QUERY;
       BEGIN
       ARRAY
           QUERY_TEXT [0:20];
       BOOLEAN
           RESULT;                   % Holds the SIM result word.

       % First, initialize the query_info record.  By setting the
       % entire record to zeroes, we use the default data formats
       % that SIM provides.
       REPLACE QUERY_INFO BY 0 FOR SIZE (QUERY_INFO) WORDS;

       % Query text begins in word 1 of the query_text parameter.
       % Word 0 of query_text is used to indicate the length of
       % the query text.
       REPLACE POINTER (QUERY_TEXT[1]) BY
           "From Person Retrieve Name";
       QUERY_TEXT[0]:= 25;          % The query text is 25
                                     % characters long.
```

```
IF RESULT:= SIM_PARSE_QUERY (PASS_ARRAY (QUERY_INFO)
                            ,PASS_ARRAY (QUERY_TEXT)
                           ) THEN
    BEGIN
    %  Some kind of error occurred.  See if it is an OML
    %  syntax error or some other kind of error.
    IF REAL(RESULT).EX_CATEGORYF    = EX_FAILED AND
       REAL(RESULT).EX_SUBCATEGORYF = EX_FAIL_DML_SYNTAX
    THEN
        % It is an OML syntax error.
        DISPLAY ("OML syntax error")
    ELSE
        % Some other kind of error occurred.
        PROCESS_ERROR;
    END
ELSE
    % No errors occurred.
    DISPLAY ("Query parsed");
END;
```

SIM_PARSE_SQL

Purpose

Use the SIM_PARSE_SQL entry point to parse queries written in SQLDB DML. Refer to "Reading In the Query Text" in Section 7, "Using the SIM Library Entry Points," for more information.

Note: *When you supply a query parameter for a SIM OML query, you must preface the identifier with a number sign (#). However, when you supply a query parameter for an SQLDB DML query, you can preface the identifier either with a number sign (#) or with a colon (:).*

Parameters

query_info

Type: Record

Allowable Values	Description
System-supplied	Tells SIM how to format certain data types. For example, attributes of type date can be returned as real numbers. After calling the SIM_PARSE_SQL entry point, the QUERY_INFO record contains additional information which uniquely identifies the query to SIM. This information is required later when you execute the query, so you should not alter this record until after you close the query using the SIM_CLOSE_QUERY entry point. Therefore, your program must allocate one QUERY_INFO record for each query that has been parsed but not closed. Refer to "Using the QUERY_INFO Record" in Section 7, "Using the SIM Library Entry Points," for more information.

qtext

Type: String

Allowable Values	Description
Any SQLDB DML query	Passes the query text to the SIM library. The first word of the string must define the length of the query text.

Related Entry Points

The following entry points are also used to parse queries:

- SIM_PARSE_FORMATTED_QUERY
- SIM_PARSE_QUERY

Example

The following ALGOL example demonstrates the use of the SIM_PARSE_SQL entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare query_info and qtext arrays.
ARRAY
    QUERY_INFO [0:PO_SZ]
  ,QTEXT [0:255]
 ;
BOOLEAN RESULT;
POINTER PTR;

% Set up qtext array.
REPLACE PTR:POINTER (QTEXT [1]) BY
   "SELECT MYDVA FROM MYCLASS WHERE MYDVA EQL ", "'", "MYSELF", "'";
QTEXT [0]:= OFFSET (PTR) - 6;

% This initial call builds the query_info and qdesc structures.
IF RESULT:=SIM_PARSE_SQL (PASS_ARRAY (QUERY_INFO)
                          ,PASS_ARRAY (QTEXT)
                         ) THEN
    BEGIN
    % Some kind of error occurred parsing the query.
    PROCESS_ERROR;
    END
ELSE
    BEGIN
    % Query is successfully parsed and is ready to be processed.
    % For the purposes of brevity, this code has been omitted.
       END;
```

SIM_PROCESS_CURSOR

Purpose

Use the SIM_PROCESS_CURSOR entry point to process one instance of a query. Refer to "Processing the Query" in Section 7, "Using the SIM Library Entry Points," for more information.

Parameters

cursor_info

Type: Record

Allowable Values	Description
System-supplied	Identifies the system-supplied cursor associated with the query. Refer to "Using the CURSOR_INFO Record" in Section 7, "Using the SIM Library Entry Points," for more information.

Related Entry Points

The following entry points are also used to process an instance of a query:

- SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA
- SIM_PROCESS_CURSOR_AND_UPDATE_DATA

Example

The following ALGOL procedure calls the entry point SIM_PROCESS_CURSOR. Then, using the field SIM_Q_STATEMENT, the program checks the type of query that is to be processed. If the query is a retrieval query, a call is made on the RETR_VALUE procedure. If the query is not a retrieval query, the query is assumed to be an update query, and a call is made on the UPD_VALUE procedure.

Once the query is processed, the result is checked for errors. If there is an error, the message "Next Query" is displayed.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to "Returning Error Messages" in Section 7, "Using the SIM Library Entry Points," for an explanation of the procedure PROCESS_ERROR.

In this example, it is assumed that the CURSOR_INFO array contains information from a previous call using the SIM_OPEN_CURSOR entry point.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare cursor_info array.
ARRAY CURSOR_INFO [0:IC_USER_CURSOR_SIZE];
BOOLEAN PROCEDURE PROCESS_CURSOR;
    BEGIN
    LABEL EXIT_HERE;
    PROCESS_CURSOR:= TRUE;
    FIRST_ENTITY:= NOT FALSE;
    WHILE NOT RSLT:= SIM_PROCESS_CURSOR(P(CURSOR_INFO)) DO
        IF  SIM_Q_STATEMENT(QUERY_INFO) EQL ST_RETRIEVE_V THEN
            RETR_VALUE
        ELSE
            UPD_VALUE;
    IF  REAL(RSLT).EX_CATEGORYF NEQ EX_NONE AND
        REAL(RSLT).EX_CATEGORYF NEQ EX_COMPLETE THEN
        EXIT("Next Query");
    PROCESS_CURSOR:= FALSE;
EXIT_HERE:
    END OF PROCESS_CURSOR;
```

SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA

Purpose

Use the SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA entry point to process an instance of a query and retrieve a record of information. When you use this entry point, the information is returned as a block of data which your program is responsible for interpreting. You can determine the layout of the information from the QDUMP file produced by using the QD_CONTROLLER entry point.

For information on the QD_CONTROLLER entry point and the QDUMP file, refer to “Returning Statistical Information” in Section 7, “Using the SIM Library Entry Points.”

When you are retrieving data from a query that involves multiple templates, you can use the field IC_TEMPLATE_NO(cursor_info) to determine which template number was returned by the last cursor call.

Note: *This entry point is not available in COBOL. To process an instance of a query and retrieve a record of information, use one of the following COBOL entry points:*

- *SCIM_PARSE_QUERY*
- *SCIM_PARSE_SQL*
- *SCIM_PROCESS_CURSOR*
- *SCIM_PUT_VALUE*
- *SCIM_GET_VALUE*

Parameters

cursor_info

Type: Record

Allowable Values	Description
System-supplied	Identifies the query cursor. This information is provided by SIM when you use the SIM_OPEN_CURSOR entry point. Refer to “Using the CURSOR_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

query_buffer

Type: Record

Allowable Values	Description
Extracted from the database	Refer to the sample program EXAMPLE/SIM/ALGOL/API/RECORDATATIME for information on decoding the record. You must ensure that the size of the query_buffer record is large enough to accommodate the maximum record size. You can calculate the total record length by adding the length of the last item to the offset of the last item.

Related Entry Points

The following entry points are also used to process an instance of a query:

- SIM_PROCESS_CURSOR
- SIM_PROCESS_CURSOR_AND_UPDATE_DATA

Example

The following ALGOL example demonstrates the use of the SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, it is assumed that the CURSOR_INFO array contains information from a previous call using the SIM_OPEN_CURSOR entry point. Also, the QUERY_BUFFER array size is assumed to be large enough to accommodate the largest record.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare cursor_info and query_buffer arrays.
ARRAY
    CURSOR_INFO [0:IC_USER_CURSOR_SIZE]
  ,QUERY_BUFFER [0:30]
 ;
BOOLEAN RESULT;

    IF  RESULT:= SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA
        (PASS_ARRAY (CURSOR_INFO)
       ,PASS_ARRAY (QUERY_BUFFER)
      )THEN
        BEGIN
             IF  EX_CATEGORY(RESULT) NEQ EX_COMPLETE THEN
                 EXIT("PROCESS CURSOR RETRIEVE")
             ELSE
                 EXIT("NO MORE RECORDS FOUND");
        END
```

SIM_PROCESS_CURSOR_AND_UPDATE_DATA

Purpose

Use the SIM_PROCESS_CURSOR_AND_UPDATE_DATA entry point to process an instance of a query and update the database using a record of information supplied by you. Refer to the program EXAMPLE/SIM/ALGOL/API/RECORDATATIME on the release media for more information.

When you use this entry point, you must supply the information as a properly formatted block of data. You can determine the required format of this block of data by using the QDUMP file, which is produced by using the QD_CONTROLLER entry point.

For information on the QD_CONTROLLER entry point and the QDUMP file, refer to “Returning Statistical Information” in Section 7, “Using the SIM Library Entry Points.”

Note: *This entry point is not defined in COBOL.*

Parameters

cursor_info

Type: Record

Allowable Values	Description
System-supplied	Identifies the query cursor. This information is provided by SIM when you use the SIM_OPEN_CURSOR entry point. Refer to “Using the CURSOR_INFO Record” in Section 7, “Using the SIM Library Entry Points,” for more information.

query_buffer

Type: Record

Allowable Values	Description
Updated into the database	Refer to the sample program EXAMPLE/SIM/ALGOL/API/RECORDATATIME for information on decoding the record. You must ensure that the size of the query_buffer record is large enough to accommodate the maximum record size. You can calculate the total record length by adding the length of the last item to the offset of the last item.

Related Entry Points

The following entry points are also used to process an instance of a query:

- SIM_PROCESS_CURSOR
- SIM_PROCESS_CURSOR_AND_RETRIEVE_DATA

Example

The following ALGOL example demonstrates the use of the SIM_PROCESS_CURSOR_AND_UPDATE_DATA entry point.

In this example, it is assumed that the CURSOR_INFO array contains information from a previous call using the SIM_OPEN_CURSOR entry point. Also, the QUERY_BUFFER array size is assumed to be large enough to accommodate the largest record.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare cursor_info and query_buffer arrays.
ARRAY
    CURSOR_INFO [0:IC_USER_CURSOR_SIZE]
  ,QUERY_BUFFER [0:30]
  ;
BOOLEAN RESULT;
    IF  RESULT:= SIM_PROCESS_CURSOR_AND_UPDATE_DATA
        (PASS_ARRAY (CURSOR_INFO)
       ,PASS_ARRAY (QUERY_BUFFER)
      )THEN
           IF  EX_CATEGORY(RESULT) NEQ EX_COMPLETE THEN
               EXIT("PROCESS CURSOR UPDATE");
```

SIM_PUT_VALUE

Purpose

Use the SIM_PUT_VALUE entry point to assign a value to a query parameter. Refer to "Passing Parameter Values to a Query" in Section 7, "Using the SIM Library Entry Points," for more information.

Note: *When you supply a query parameter for a SIM OML query, you must preface the identifier with a number sign (#). However, when you supply a query parameter for an SQLDB DML query, you can preface the identifier either with a number sign (#) or with a colon (:).*

Parameters

cursor_info

Type: Record

Allowable Values	Description
Same record area as that passed to the SIM_OPEN_CURSOR entry point	Identifies the query cursor. Refer to "Using the CURSOR_INFO Record" in Section 7, "Using the SIM Library Entry Points," for more information.

item_info

Type: Record

Allowable Values	Description
Same record area as that passed to the SIM_DESCRIBE_ PARAMETER entry point	Describes the parameter to which you want to supply a value. Information you must provide includes the item number, template, type, and whether the item is null. If the item is null, the information associated with the string_value and numeric_value parameters is not read. Refer to "Using the ITEM_INFO Record" in Section 7, "Using the SIM Library Entry Points," for more information.

string_value

Type: String

Allowable Values	Description
User-defined	Contains the string to be assigned to the parameter. The first word of the string_value parameter must provide the length of the string. You need only assign a string to the string_value parameter if your parameter takes a nonnumeric or Boolean value.

numeric_value

Type: Double

Allowable Values	Description
User-defined	Contains the numeric value to be assigned to the parameter. You need only assign a value to the numeric_value parameter if your parameter takes a numeric or Boolean value.

Notes:

You should be aware of the following truncation rules for numeric values:

- *When the value assigned to a numeric item is larger than the size of the item as declared in the database schema, SIM truncates the value by removing those digits which do not fit within the constraints of the item after decimal point alignment. These rules are consistent with the truncation rules in COBOL and in DMSII.*
- *If the item is declared in the database schema as a SUBRANGE, SIM performs the subrange verification after the truncation has occurred.*

Related Entry Points

The following entry point provides the data type associated with a query parameter:

- SIM_DESCRIBE_PARAMETER

Example

The following example demonstrates the use of the SIM_DESCRIBE_PARAMETER and the SIM_PUT_VALUE entry points. The example works for any query text.

In the example, an assumption is made that the query text has already been parsed and a cursor for the query has been opened, using the corresponding program variables QUERY_INFO and CURSOR_INFO.

In the example, all string parameters are assigned the letter X, the number of the parameter is assigned to all numeric parameters, (that is, parameter number 1 is assigned the value 1, parameter number 2 is assigned the value 2, and so on) Boolean parameters are assigned a true value.

PASS_ARRAY is a define used when passing array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
ARRAY
  QUERY_INFO  [0:PO_SIZE],
  CURSOR_INFO [0:IC_CURSOR_SIZE];

  PROCEDURE PARAMETER_EXAMPLE;
    BEGIN
    BOOLEAN
       UNKNOWN;
    INTEGER
       PARAMETER_NUMBER;
    DOUBLE
       NUMERIC_VALUE;
    ARRAY
       STRING_VALUE [0:100],
       ITEM_INFO    [0: SIM_ITEM_RECORD_SIZE];

    PARAMETER_NUMBER:= 1;
    SIM_ITEM_NUMBER (ITEM_INFO):= PARAMETER_NUMBER;

    %  The following WHILE statement is executed as long as
    %  the parameter number passed in SIM_ITEM_NUMBER is a valid
    % parameter number.  Once the parameter number exceeds the
    %  number of parameters actually needed by the query,
    %  SIM_DESCRIBE_PARAMETER returns an error code.

    WHILE NOT SIM_DESCRIBE_PARAMETER (PASS_ARRAY (QUERY_INFO)
                                     ,PASS_ARRAY (ITEM_INFO)
                                    ) DO
        BEGIN
        UNKNOWN:= FALSE;

        %  This CASE statement illustrates the information
        % returned by SIM_DESCRIBE_PARAMETER.

        DISPLAY ("Parameter #"!! STRING(PARAMETER_NUMBER, *));
        CASE SIM_ITEM_TYPE (ITEM_INFO) OF
            BEGIN
        (ITEM_IS_REAL_V):
                BEGIN
                DISPLAY ("    Real");
                END;
```

```
(ITEM_IS_INTEGER_V):
        BEGIN
        DISPLAY ("    Integer"):
        DISPLAY ("    "!!
                 STRING (SIM_ITEM_PRECISION (ITEM_INFO), *)
               !! " Digits"
              );
        END;

(ITEM_IS_BCD_V):
        BEGIN
        DISPLAY ("    BCD Number"):
        DISPLAY ("    Precision: "!!
                 STRING (SIM_ITEM_PRECISION (ITEM_INFO), *)
              );
        DISPLAY ("    Scale: "!!
                 STRING (SIM_ITEM_SCALE (ITEM_INFO), *)
              );
        IF SIM_ITEM_SIGNED (ITEM_INFO) THEN
            DISPLAY ("    Signed");
        END;
(ITEM_IS_STRING_V):
        BEGIN
        CASE SIM_ITEM_SUBTYPE (ITEM_INFO) OF
            BEGIN
        (ITEM_IS_EBCDIC_V):
            DISPLAY ("    Fixed length EBCDIC string"):
        (ITEM_IS_KANJI_V):
            DISPLAY ("    Fixed length KANJI string"):
        ELSE:
            DISPLAY ("    Fixed length string of "!!
                     "unknown type"
                  );
            END;  % CASE on subtype
        DISPLAY ("    Length: "!!
                 STRING (SIM_ITEM_LENGTH (ITEM_INFO), *)
              );
        DISPLAY ("    CCS Version: "!!
                 STRING (SIM_ITEM_CCSVERSION(ITEM_INFO), *)
              );
        END;
(ITEM_IS_VLSTRING_V):
        BEGIN
        CASE SIM_ITEM_SUBTYPE (ITEM_INFO) OF
```

```
                BEGIN
            (ITEM_IS_EBCDIC_V):
                DISPLAY ("    Variable length EBCDIC string"):
            (ITEM_IS_KANJI_V):
                DISPLAY ("    Variable length KANJI string"):
            ELSE:
                DISPLAY ("    Variable length string of "!!
                         "unknown type"
                      );
                END;  % CASE on subtype

            DISPLAY ("    Maximum length: "!!
                     STRING (SIM_ITEM_LENGTH (ITEM_INFO), *)
                  );
            DISPLAY ("    CCS Version: "!!
                     STRING (SIM_ITEM_CCSVERSION(ITEM_INFO), *)
                  );
            END;
    (ITEM_IS_BOOLEAN_V):
            BEGIN
            DISPLAY ("    Boolean"):
            END;
    ELSE:
            BEGIN
            DISPLAY ("    **** Unknown parameter type ****"):
            UNKNOWN:= TRUE;
            END;
        END;  % CASE

    %  This part of the example illustrates the use of the
    %  SIM_PUT_VALUE entry point.

    CASE SIM_ITEM_TYPE (ITEM_INFO) OF
        BEGIN
    (ITEM_IS_REAL_V):
    (ITEM_IS_INTEGER_V):
    (ITEM_IS_BCD_V):
        NUMERIC_VALUE:= PARAMETER_NUMBER;
    (ITEM_IS_STRING_V):
    (ITEM_IS_VLSTRING_V):
        BEGIN
        REPLACE POINTER(STRING_VALUE [1]) BY "X" FOR
            SIM_ITEM_LENGTH (ITEM_INFO);
        STRING_VALUE[0]:= SIM_ITEM_LENGTH (ITEM_INFO);
        END;
    (ITEM_IS_BOOLEAN_V):
        NUMERIC_VALUE:= PARAMETER_NUMBER.[0:1];
    ELSE:
        END;  % CASE
        IF NOT UNKNOWN THEN
            SIM_PUT_VALUE (PASS_ARRAY (CURSOR_INFO)
                         ,PASS_ARRAY (ITEM_INFO)
                         ,PASS_ARRAY (STRING_VALUE)
                         ,NUMERIC_VALUE
                        );
        % Increment PARAMETER_NUMBER and get the next parameter.
        PARAMETER_NUMBER:= PARAMETER_NUMBER + 1;
        SIM_ITEM_NUMBER (ITEM_INFO):= PARAMETER_NUMBER;
        END;
    END;
```

SIM_QD_CONTROLLER

Purpose

Use the SIM_QD_CONTROLLER entry point to provide statistical information on a query. Refer to "Returning Statistical Information" in Section 7, "Using the SIM Library Entry Points," for more information.

Parameters

qdc_text

Type: String

Allowable Values	Description
Refer to "Returning Statistical Information" in Section 7, "Using the SIM Library Entry Points."	Designates the type of information you want and which queries you want it for. Use the first word (48 bits) of the string to indicate the length of the text. A true or false value is returned as well as the information you request. If the request could not be processed correctly, a true value is returned and qdc_text contains the error message. If you do not designate a query number, information about all the queries currently known to the Query Driver are dumped. The information from this command is stored in a file called `QDUMP/<mix number>/<serial number>` The mix number is that of the user stack, and the serial number is an integer. A value of 1 is assigned as the serial number for the first query processed. The serial number is then incremented by 1 for each subsequent query processed. Use the CLOSE QUERY or the CLOSE QGRAPH command to end the process and to close the output file.

Related Entry Points

There are no related entry points.

Example

The following ALGOL example demonstrates the use of the SIM_QD_CONTROLLER entry point.

In this example, it is assumed a query has been parsed successfully.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare qdc_text array.
ARRAY QDC_TEXT [0:30];
BOOLEAN RESULT;
POINTER PTR;

% Set up qdc_text request.
REPLACE PTR:POINTER (QDC_TEXT [1]) BY "DUMP QUERY";
QDC_TEXT [0]:= DELTA (POINTER (QDC_TEXT [1]), PTR);

IF RESULT:= SIM_QD_CONTROLLER (PASS_ARRAY (QDC_TEXT)) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    END
ELSE
    BEGIN
    % Statistical information for the parsed query
    % has been obtained.
    END;
```

SIM_SAVE_QUERY

Purpose

Use the SIM_SAVE_QUERY entry point to save queries in a file. For more information, refer to “Saving and Restoring Queries” in Section 7, “Using the SIM Library Entry Points.”

Parameters

file_title

Type: String

Allowable Values	Description
Any valid A Series file title	Designates the file into which you want to save the query information. The first word (48 bits) of the string must contain the length of the string.

Related Entry Points

The following entry points are also used to save and restore queries:

- SIM_ADD_QUERY
- SIM_LOAD_QUERY

Example

The following ALGOL example demonstrates the use of the SIM_SAVE_QUERY entry point.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
% Declare file_title array.
ARRAY FILE_TITLE [0:255];
BOOLEAN RESULT;
POINTER PTR;

% Set up file_title.
REPLACE PTR:POINTER(FILE_TITLE [1]) BY "SAVED/QUERIES";
FILE_TITLE [0]:= OFFSET (PTR) - 6;

IF RESULT:= SIM_SAVE_QUERY (PASS_ARRAY (FILE_TITLE)) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    END
ELSE
    BEGIN
    % Parsed queries saved.
    END;
```

SIM_SAVEPOINT

Purpose

Use the SIM_SAVEPOINT entry point to place a marker allowing you to undo a subset of the changes requested during the current transaction. For more information, refer to “Performing Queries While in Transaction State” in Section 7, “Using the SIM Library Entry Points.”

Parameters

savepoint

Type: Integer

Allowable Values	Description
Any positive integer value	Identifies a marker after which all updates can be canceled.

Related Entry Points

The following entry points are also used with transactions:

- SIM_ABORT
- SIM_BEGIN_TRANSACTION
- SIM_CANCEL
- SIM_COMS_BEGIN_TRANSACTION
- SIM_COMS_ETR
- SIM_ETR

Example

The following ALGOL example demonstrates the use of the SIM_SAVEPOINT entry point.

In this example, a transaction to the database requires a lengthy process between two different SIM_OPEN_CURSOR calls. In the event the results of the intermediate process are incorrect the SIM_CANCEL entry point is used to undo any updates since the designated savepoint.

In the event of an error in calling any entry point, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the procedure PROCESS_ERROR.

In this example, the QUERY_INFO_x arrays are already declared and are assumed to contain information from previous calls using the SIM_PARSE_QUERY entry points. Also, it is assumed that the CURSOR_INFO_x arrays are declared previously.

PASS_ARRAY is a define used to pass array parameters. Refer to "Passing Arrays to SIM from ALGOL Programs" in Section 7, "Using the SIM Library Entry Points," for an explanation of the PASS_ARRAY define.

```
 % Enter transaction state.
   IF RESULT:= SIM_BEGIN_TRANSACTION (LEVEL_LOCKING) THEN
       BEGIN
       % There was an error trying to enter transaction
       % state.
       PROCESS_ERROR;
       GO END_OF_EXAMPLE;
       END;

   % The program is now in transaction state - process the compiled
   % query.
   IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_1)
                               ,PASS_ARRAY (CURSOR_INFO_1)
                              ) THEN
       BEGIN
       % Some kind of error occurred.
       PROCESS_ERROR;
       GO END_OF_TRANSACTION;
       END
   ELSE
       BEGIN
       IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (QUERY_INFO_1)) THEN
           BEGIN
           % Some kind of error occurred.
           PROCESS_ERROR;
           SIM_ABORT;
           GO END_OF_EXAMPLE;
           END;
       END;

   % Mark a savepoint.
   IF RESULT:= SIM_SAVEPOINT (1) THEN
       BEGIN
       % Some kind of error occurred while trying to mark the
       % savepoint.
       PROCESS_ERROR;
       SIM_ABORT;
       GO END_OF_EXAMPLE;
       END;
```

```
IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_2)
                            ,PASS_ARRAY (CURSOR_INFO_2)
                           ) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    GO END_OF_TRANSACTION;
    END
ELSE
    BEGIN
    IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_2)) THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        GO END_OF_TRANSACTION;
        END;
    END;

% Mark a savepoint.
IF RESULT:= SIM_SAVEPOINT (2) THEN
    BEGIN
    % Some kind of error occurred while trying to mark the
    % savepoint.
    PROCESS_ERROR;
    SIM_ABORT;
    GO END_OF_EXAMPLE;
    END;

% Here is a lengthy update process.
PROCESS_MRP (PROCESS_RESULT);

IF PROCESS_RESULT THEN
    BEGIN
    % The results of PROCESS_MRP are incorrect.
    % Undo the updates made since the second savepoint.
    IF RESULT:= SIM_CANCEL (2) THEN
        BEGIN
        % Some kind of error occurred. An error when calling
        % SIM_CANCEL indicates that the updates are not canceled
        % and that the database is still in transaction state.
        PROCESS_ERROR;
        SIM_ABORT;
        GO END_OF_EXAMPLE;
        END;
    END;
```

```
IF RESULT:= SIM_OPEN_CURSOR (PASS_ARRAY (QUERY_INFO_3)
                            ,PASS_ARRAY (CURSOR_INFO_3)
                           ) THEN
    BEGIN
    % Some kind of error occurred.
    PROCESS_ERROR;
    GO END_OF_TRANSACTION;
    END
ELSE
    BEGIN
    IF RESULT:= SIM_PROCESS_CURSOR (PASS_ARRAY (CURSOR_INFO_3)) THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        GO END_OF_TRANSACTION;
        END;
    END;

END_OF_TRANSACTION:

    % End transaction state.
    IF RESULT:= SIM_ETR THEN
        BEGIN
        % Some kind of error occurred.
        PROCESS_ERROR;
        SIM_ABORT;
        END;

END_OF_EXAMPLE:
```

SIM_SET_DICTIONARY

Purpose

Use the SIM_SET_DICITONARY entry point to identify the ADDS dictionary associated with a database. You need to use this entry point only if your database is defined in an ADDS dictionary.

Parameters

dictionary_name

Type: String

Allowable Values	Description
User-defined	Identifies the dictionary. You must supply the support library name associated with the dictionary. The first word (48 bits) of the string define the length of the text. Note that an error is returned if you supply a blank or empty string.

Related Entry Points

The following entry points are also used in database operations:

- SIM_CLOSE_DATABASE
- SIM_OPEN_DATABASE

Example

The following ALGOL procedure demonstrates the use of the SIM_SET_DICTIONARY entry point. If the dictionary is successfully opened, the message “Dictionary opened” is displayed.

If the dictionary is not opened, the procedure PROCESS_ERROR is called. Refer to “Returning Error Messages” in Section 7, “Using the SIM Library Entry Points,” for more information on the procedure PROCESS_ERROR.

This example assumes that you have provided a dictionary name and that the name is currently stored in the string variable DICTIONARY_NAME.

PASS_ARRAY is a define used to pass array parameters. Refer to “Passing Arrays to SIM from ALGOL Programs” in Section 7, “Using the SIM Library Entry Points,” for an explanation of the PASS_ARRAY define.

```
ARRAY
     DICTNAME [0:30];

 PROCEDURE OPEN_DICTIONARY_EXAMPLE;
     BEGIN
     BOOLEAN
         RESULT;                    % Holds the SIM result word.

     %  First, move the dictionary name from the string
     %  DICTIONARY_NAME to DICTNAME, which is the
     %  parameter to the SIM_SET_DICTIONARY entry point.
     REPLACE POINTER(DICTNAME[1]) BY DICTIONARY_NAME
         LENGTH (DICTIONARY_NAME);
     DICTNAME [0]:= LENGTH (DICTIONARY_NAME);

     %  Now you can open the dictionary.
     IF RESULT:= SIM_SET_DICTIONARY (PASS_ARRAY (DICTNAME))
     THEN
         BEGIN
         % An error occurred while opening the dictionary.
         PROCESS_ERROR;
         DISPLAY("Example failed while opening the dictionary");
         END
     ELSE
         DISPLAY ("Dictionary opened");
     END;
```

Appendix A
OML Error Messages

The following is an explanation of the most common error messages that SIM returns when you process a query. The error messages are described in alphabetic order. Where applicable, examples are provided of queries that generate the errors and the means for altering the query to make the query compile without errors. The example queries are based on the ORGANIZATION database, which is described in Appendix C, "Description of the ORGANIZATION Database."

Attribute name expected—<item name>.

SIM returns this message when an item in the RETRIEVE clause of a query is not a valid attribute name or expression. The error might have been caused by any of the following:

- Typographical errors
- A user-defined type (UDT) name used in place of an attribute name
- An attribute that is not in the perspective class

The following query returns this error message because ADDRESS is a UDT and not an attribute:

```
FROM PERSON OF ORGANIZATION
RETRIEVE NAME, ADDRESS
```

The following query corrects the error:

```
FROM PERSON OF ORGANIZATION
RETRIEVE NAME, CURRENT_RESIDENCE
```

Attribute not connected to a class of interest—<item name>.

SIM returns this message when the RETRIEVE clause contains an attribute from a class other than the perspective class, but the qualification chain, if any, does not relate the attribute to the perspective class. To resolve the problem, check for any of the following:

- EVAs that point away from the perspective class rather than to the perspective class
- Missing EVAs

The following query returns the error message "Attribute not connected to a class of interest —DEPT_LOCATION" because DEPT_LOCATION is an attribute belonging to the DEPARTMENT class. No qualification is provided to relate it to the perspective class PROJECT_EMPLOYEE.

```
RETRIEVE EMPLOYEE_ID OF PROJECT_EMPLOYEE,
         MANAGER_TITLE OF EMPLOYEE_MANAGER,
         START_DATE OF ASSIGNMENT_RECORD,
         PROJECT_NO OF PROJECTS_MANAGING,
         DEPT_LOCATION
```

You can correct the query in the following way:

```
RETRIEVE EMPLOYEE_ID OF PROJECT_EMPLOYEE,
         MANAGER_TITLE OF EMPLOYEE_MANAGER,
         START_DATE OF ASSIGNMENT_RECORD,
         PROJECT_NO OF PROJECTS_MANAGING,
         DEPT_LOCATION OF MANAGERS_DEPT
```

Class name cannot be within the argument to TRANSITIVE.

SIM returns this message if you include a class, rather than an EVA in the TRANSITIVE clause.

The following query returns this error:

```
RETRIEVE TRANSITIVE (MANAGER END LEVEL = 2)
```

You can correct the query in the following way:

```
FROM MANAGER
RETRIEVE TRANSITIVE (EMPLOYEE_MANAGER END LEVEL = 2)
```

Class name expected after FROM.

SIM returns this message when the item following the FROM keyword is not a class name. This problem can be caused by such things as a typographical error or by the database not being open.

The following query returns this error because the construct PRJECT_EMPLOYEE follows the FROM keyword instead of the PROJECT_EMPLOYEE attribute:

```
FROM PRJECT_EMPLOYEE
RETRIEVE EMPLOYEE_ID,
         EMPLOYEE_ID OF EMPLOYEE_MANAGER,
WHERE DEPT_TITLE OF DEPT_IN = "Engineering"
```

Error occurred at the end of the statement.

SIM returns this message if it detects that the error occurred at the end of the query. The error message is usually returned with another error message detailing the type of error. Refer to the description of the second error message for more information.

The following query returns this error message:

```
FROM EMPLOYEE
RETRIEVE NAME, AGE, NAME OF EMPLOYEE_MANAGER, AGE OF EMPLOYEE_MANAGER
```

The error in this query is in the element AGE OF EMPLOYEE_MANAGER. As EMPLOYEE_MANAGER is a reflexive attribute, this element can be fully qualified in either of the following ways:

- AGE OF EMPLOYEE_MANAGER OF EMPLOYEE_MANAGER OF EMPLOYEE
- AGE OF EMPLOYEE_MANAGER OF EMPLOYEE

You can correct the query by giving the appropriate full qualification or by using parentheses, as follows:

```
FROM EMPLOYEE
RETRIEVE NAME, AGE, (NAME, AGE) OF EMPLOYEE_MANAGER
```

Error precedes—<portion of OML statement>

SIM returns this message to identify the location of the error. The error message is usually returned with another message describing the form of the error. Refer to the description of the second error message for more information.

INCLUDE is allowed only for multivalued attributes.

SIM returns this message when you use the keyword INCLUDE in the assignment statement for a single-valued attribute.

The following query returns this error message because EMPLOYEE_MANAGER is a single-valued EVA of the EMPLOYEE class:

```
MODIFY EMPLOYEE
       (EMPLOYEE_MANAGER:= INCLUDE MANAGER
                            WITH (LAST_NAME OF NAME = "Bartlett" AND
                                  FIRST_NAME OF NAME = "Anne")
       )
WHERE (LAST_NAME OF NAME = "Munks" AND FIRST_NAME OF NAME = "Jerry")
```

To correct the query, remove the keyword INCLUDE. For more information, refer to “Adding Values to Attributes” in Section 5, “Structuring Update Queries.”

Invalid attribute chain at <attribute identifier>.

SIM returns this message when a path linking an attribute to the perspective class is incorrect. Check for EVAs that are pointing in the wrong direction.

The following query returns the error message “Invalid attribute chain at DEPT_MANAGERS”:

```
FROM EMPLOYEE
RETRIEVE DEPT_TITLE OF DEPT_MANAGERS OF EMPLOYEE_MANAGER
```

The qualification DEPT_MANAGERS provides a link between the DEPARTMENT and the MANAGER classes but in the wrong direction. The correct query is as follows:

```
FROM EMPLOYEE
RETRIEVE DEPT_TITLE OF MANAGERS_DEPT OF EMPLOYEE_MANAGER
```

Invalid introduction of class of interest—<class name>.

SIM returns this message when a query includes a CALLED keyword, but the element defined by the CALLED keyword is a perspective class in the query.

The following query returns the message “Invalid introduction of class of interest—PROJECT_EMPLOYEE”:

```
FROM PROJECT_EMPLOYEE
RETRIEVE EMPLOYEE_ID OF EMPLOYEE_MANAGER
WHERE EMPLOYEE_SALARY OF PROJECT_EMPLOYEE <= 30000
AND EMPLOYEE_SALARY OF PROJECT_EMPLOYEE CALLED OTHER_EMPLOYEE >= 50000
AND PROJECT_TITLE OF CURRENT_PROJECT = "Excalibur"
AND PROJECT_TITLE OF CURRENT_PROJECT OF OTHER_EMPLOYEE = "Lancelot"
```

The correct query looks as follows:

```
FROM PROJECT_EMPLOYEE, PROJECT_EMPLOYEE CALLED OTHER_EMPLOYEE
RETRIEVE EMPLOYEE_ID OF EMPLOYEE_MANAGER
WHERE EMPLOYEE_SALARY OF PROJECT_EMPLOYEE <= 30000
AND EMPLOYEE_SALARY CALLED OTHER_EMPLOYEE >= 50000
AND PROJECT_TITLE OF CURRENT_PROJECT = "Excalibur"
AND PROJECT_TITLE OF CURRENT_PROJECT OF OTHER_EMPLOYEE = "Lancelot"
```

For more information on using the CALLED keyword, refer to “Using a Reference Variable” in Section 2, “OML Language Conventions.”

Invalid mixing of types in a comparison.

SIM returns this message when the format of a value does not match the type of attribute with which it is being compared. Check the data types associated with the attributes in the WHERE and WITH clauses.

The following query returns this error message because the format of the date in the WHERE clause is incorrect:

```
RETRIEVE EMPLOYEE_ID OF PROJECT_EMPLOYEE,
         PROJECT_TITLE OF CURRENT_PROJECT OF PROJECT_EMPLOYEE
WHERE START_DATE OF ASSIGNMENT_RECORD OF PROJECT_EMPLOYEE = 010188
```

Following is the correct query:

```
RETRIEVE EMPLOYEE_ID OF PROJECT_EMPLOYEE,
         PROJECT_TITLE OF CURRENT_PROJECT OF PROJECT_EMPLOYEE
WHERE START_DATE OF ASSIGNMENT_RECORD OF PROJECT_EMPLOYEE = 1/1/88
```

For more information on the WHERE and WITH clauses, refer to “Limiting the Data Retrieved” in Section 3, “Structuring Retrieval Queries,” and “Limiting the Number of Entities Changed” in Section 5, “Structuring Update Queries.”

Invalid use of indirect recursion in TRANSITIVE (no cycle formed).

SIM returns this message when the TRANSITIVE function does not define a loop. This error usually occurs because the attribute in the TRANSITIVE function is not an EVA.

The following query returns this error:

```
FROM MANAGER
RETRIEVE TRANSITIVE (NAME END LEVEL = 2)
```

Following is the correct query:

```
FROM MANAGER
RETRIEVE NAME, TRANSITIVE (EMPLOYEE_MANAGER END LEVEL =2)
```

For more information on the TRANSITIVE function, refer to “Accessing EVA Information” in Section 6, “Writing Expressions for Retrieval and Update Queries.”

Name must match or be constituent of the EXCLUDED attribute—<attribute>.

SIM returns this message when you try to update a single-valued attribute and put anything after the EXCLUDE keyword.

The following query returns the error message “Name must match or be constituent of the EXCLUDED attribute—MANAGER”:

```
MODIFY EMPLOYEE
       (EMPLOYEE_MANAGER:= EXCLUDE MANAGER
                            WITH (LAST_NAME OF NAME = "Bartlett"
                            AND FIRST_NAME OF NAME = "Anne")
     )
WHERE (LAST_NAME OF NAME = "Munks" AND FIRST_NAME OF NAME = "Jerry")
```

The correct query looks as follows:

```
MODIFY EMPLOYEE
       (EMPLOYEE_MANAGER:= EXCLUDE)
WHERE (LAST_NAME OF NAME = "Munks" AND FIRST_NAME OF NAME = "Jerry")
```

For more information, refer to “Replacing Attribute Values” in Section 5, “Structuring Update Queries.”

Output for this query cannot be structured. Choose tabular output.

SIM returns this message when you include in a retrieval query options that are incompatible with the STRUCTURE option. For example, this error is returned when you use the keywords DISTINCT and STRUCTURE in the same retrieval query. To resolve this problem, either remove the distinct requirement, or alter the output to tabular. For more information on structured and tabular output, refer to Section 4, “Formatting the Output from Retrieval Queries.”

Qualification is insufficient.

SIM returns this message when a retrieval query includes two or more implied qualifications, all of which can be applied to an attribute occurring later in the retrieval query statement.

The following query returns this error:

```
FROM EMPLOYEE
RETRIEVE NAME, AGE, NAME OF EMPLOYEE_MANAGER, AGE OF EMPLOYEE_MANAGER
```

The error occurs because the EMPLOYEE_MANAGER attribute is an EVA that can qualify both employee and manager; therefore, you can qualify the request to retrieve AGE OF EMPLOYEE_MANAGER in either of the following two ways:

- AGE OF EMPLOYEE_MANAGER OF EMPLOYEE_MANAGER OF EMPLOYEE
- AGE OF EMPLOYEE_MANAGER OF EMPLOYEE

You can correct the error, using the appropriate full qualification or, in place of the second qualification shown, using the following query:

```
FROM EMPLOYEE
RETRIEVE NAME, AGE, (NAME, AGE) OF EMPLOYEE_MANAGER
```

For more information on qualifying names, refer to “Qualifying Extended Attributes” in Section 2, “OML Language Conventions.”

Syntax error. Unexpected token—<string>.

SIM returns this message when you use illegal characters or when an element is missing. The following query returns the error “Syntax error. Unexpected token—EMPLOYEE_SALARY” because the comma has been omitted after the first item in the RETRIEVE clause:

```
RETRIEVE MANAGER_TITLE OF MANAGER
         EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF MANAGER
```

Following is the correct query:

```
RETRIEVE MANAGER_TITLE OF MANAGER,
         EMPLOYEE_SALARY OF EMPLOYEES_MANAGING OF MANAGER
```

Unknown name —<item name>.

SIM returns this message when an item in a query is not recognized as a user-defined type (UDT), class, database, or attribute name. SIM also returns this message if you incorrectly use a reference variable.

The following query returns this error message because SALARY is an unknown item:

```
FROM MANAGER, EMPLOYEE
RETRIEVE NAME OF MANAGER, NAME OF EMPLOYEE
WHERE SALARY OF MANAGER = SALARY OF EMPLOYEE
```

The correct query looks as follows:

```
FROM MANAGER, EMPLOYEE
RETRIEVE NAME OF MANAGER, NAME OF EMPLOYEE
WHERE EMPLOYEE_SALARY OF MANAGER = EMPLOYEE_SALARY OF EMPLOYEE
```

For information on how to use reference variables, refer to “Using a Reference Variable” in Section 2, “OML Language Conventions.”

Appendix B
Exception Fields and Categories

This appendix lists and explains the category and subcategory information you might receive when using the SIM_NEXT_EXCEPTION, SIM_EXCEPTION_MESSAGE, SIM_GET_EXCEPTION_INFO, and SIM_EXCEPTION_INFO entry points.

Category 0: EX_NONE

There is only one subcategory in this category—0 (zero).

If you receive a category and a subcategory of 0 (zero), then an exception did not occur, and the operation you requested processed successfully.

Category 1: EX_WARNING

All exceptions with a category of 1 return warnings. A warning informs you about occurrences that do not affect the results of the operation that you processed. Table B–1 describes the subcategories associated with category 1.

Table B–1. Warning Messages

Subcategory	Subcategory Mnemonic/Explanation
0	Explanation: There is no subcategory information.
1	EX_WARN_REPARSE Explanation: The query was processed correctly, but it was reparsed. To obtain better performance, recompile your program.
2	EX_WARN_TRANSLATE Explanation: Protocol translation is occurring. You can obtain better performance by recompiling your program.
3	EX_WARN_DML Explanation: An OML or DML warning was issued.
4	EX_WARN_S_CODE Explanation: The s-code compiler issued a warning while generating the code segment of your query.

Table B–1. Warning Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
10	EX_WARN_GLOBAL_SELECT Explanation: The selection expression in the MODIFY query did not select any entities.
11	EX_WARN_LOCAL_SELECT Explanation: A local selection expression in an INCLUDE or EXCLUDE statement did not select any entities.
12	EX_WARN_NOTHING_UPDATED Explanation: The query did not update any entities.
13	EX_WARN_ACTIVE_CURSOR Explanation: You are attempting to open a cursor that is currently active. This cursor will be closed and then reopened.
14	EX_WARN_NULL_ITEM Explanation: The item just returned has a null value.
15	EX_WARN_INQUIRY_ONLY Explanation: The database has been opened for inquiry-purposes only.
16	EX_WARN_VALIDATED_ONLY Explanation: The database opened has been validated but not generated. You may only compile queries for this database.
17	EX_WARN_OPEN_LINCDB Explanation: The database opened is a LINC II database and may be accessed for inquiry only.

Category 2: EX_COMPLETE

This category indicates that a sequence of events has completed. Table B–2 describes the subcategories associated with category 2.

Table B–2. Completion Messages

Subcategory	Subcategory Mnemonic/Explanation
0	Explanation: The sequence of events you requested has completed and there is no additional information to be returned.
1	EX_CMPL_NO_MORE Explanation: Your query is complete and there is no more data to be returned.
2	EX_CMPL_LEVEL Explanation: There is no more data to return at this level of the query structure. However, there might be more data at other levels.
3	EX_CMPL_INTERRUPT Explanation: The query processing was interrupted and can now be continued.

Category 3: EX_FAILED

This category is returned when a query fails or when a process fails to complete. Table B–3 describes the subcategories associated with category 3.

Table B–3. Failure Messages

Subcategory	Subcategory Mnemonic/Explanation
0	Explanation: The query or process failed to complete and there is no additional information to return.
1	EX_FAIL_DB_VERSION Explanation: There is a database version timestamp mismatch; that is, the version timestamp of the current database has changed since you compiled your program.
2	EX_FAIL_RELEASE_LEVEL Explanation: Your program is incompatible with the current release of SIM. You need to recompile your program.
3	EX_FAIL_DML_SYNTAX Explanation: There is a syntax error in your query. Use the SIM_EXCEPTION_MESSAGE entry point to return the text of the syntax error message. The field EI_DML_OFFSET contains the offset identifying the location of the error, and the field EI_TOKEN identifies the token that was being scanned when the error occurred. Refer to Appendix A, “OML Error Messages,” for an explanation of error messages for OML queries, and refer to the *InfoExec SQLDB Programming Guide* for an explanation of SQLDB DML error messages.
4	EX_FAIL_S_CODE_SYNTAX Explanation: The s-code compiler detected an error while generating the code segment for your query. This is an internal error. Please file a User Communication Form (UCF) to report the problem.
5	EX_FAIL_QUERY_MAX Explanation: The query number you have designated is larger than the maximum query number allowed.
6	EX_FAIL_FUNCTION_MAX Explanation: You have exceeded the limit on active user-defined functions.
7	EX_FAIL_DOUBLE_ENTRY Explanation: You tried to enter a SIM environment that is currently being used by another process.

continued

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
8	EX_FAIL_COMPLEX_REPARSE Explanation: Your complex update query needs to be reparsed. Resubmit your query or recompile your program.
9	EX_FAIL_QUERY_DEAD Explanation: Your query cannot continue until the previously reported errors have been fixed.
10	EX_FAIL_GLOBAL_SELECT Explanation: The FROM clause of your INSERT statement selected more than one entity.
12	EX_FAIL_GLOBAL_LIMIT Explanation: Your update query has been rejected because the number of entities selected exceeds the limit defined by the LIMIT clause.
13	EX_FAIL_LOCAL_LIMIT Explanation: Your query has been rejected because the number of entities selected exceeds the limit defined by the LIMIT clause of an EXCLUDE statement.
14	EX_FAIL_CONSTRAINT Explanation: Your update query failed because it tried to assign a value that already exists in the database to an attribute declared with a UNIQUE constraint.
15	EX_FAIL_VERIFY Explanation: The named verify failed.
16	EX_FAIL_BAD_INSRT_ASGN Explanation: Your ASSIGN statement is not valid for a STARTINSERT query. (This exception can be returned only if you are using the host language extensions to query the database.)
17	EX_FAIL_BAD_INSERT_FROM Explanation: Your query failed because you tried to add a subrole that already exists.
20	EX_FAIL_DB_CLOSED Explanation: Your query failed because the database you want to use is not open.
21	EX_FAIL_DB_OPEN Explanation: Your request to open the database failed because the database is already open.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
22	EX_FAIL_DB_NOT_FOUND Explanation: Your request to open a database failed because the database cannot be found.
23	EX_FAIL_DB_MAX Explanation: Your request to open a database failed because too many databases are already open.
24	EX_FAIL_DB_INQUIRY Explanation: Your update query failed because the database is open for inquiry purposes only.
26	EX_FAIL_MAPPER_SL Explanation: The DMSIISUPPORT or FILESUPPORT library required to access the database or file is not available. Please verify your configuration specifications for the SIMDMSIISUPPORT and SIMFILESUPPORT library functions. Note that this error should not occur. If it does, there might be an installation problem with the SLCONFIGSUPPORT library.
30	EX_FAIL_NOT_TRANS Explanation: Your program must be in transaction state to accomplish the requested action.
31	EX_FAIL_IN_TRANS Explanation: Your transaction cannot start because another transaction is already in process.
32	EX_FAIL_WRONG_DB Explanation: You tried to update more than one database in a single update query. Updates within a single transaction must all occur against the same database.
33	EX_FAIL_DB_ABORTED Explanation: The operation you requested failed because the database has been aborted.
34	EX_FAIL_DB_DEADLOCKED Explanation: The operation you requested failed because a deadlock occurred; either a MAXWAIT timeout, a record lock, or too many records locked caused the deadlock.
39	EX_FAIL_BAD_SAVE_PNT Explanation: You designated an invalid savepoint number.
40	EX_FAIL_IC_CLOSED Explanation: You have not initialized the query variable.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
41	EX_FAIL_IC_OPEN Explanation: You requested a query variable that is in use.
42	EX_FAIL_CRNT_IC_CLOSED Explanation: The query variable for the CURRENT function is not open.
43	EX_FAIL_INVALID_CRNT_IC Explanation: The query variable for the CURRENT function does not reference an entity of the correct class.
44	EX_FAIL_WRONG_RECORD Explanation: The query variable in a RETRIEVE statement refers to a query record description that is not valid at the current time.
49	EX_FAIL_ADDS_NOT_SPECIFIED Explanation: You need to designate an ADDS dictionary. Use the SIM_SET_DICTIONARY entry point to designate a dictionary.
50	EX_FAIL_CANT_CHANGE_ADDS Explanation: You attempted to bind programs that do not use the same ADDS dictionary. Programs that you want to bind together must use the same data dictionary.
51	EX_FAIL_BAD_ADDS_NAME Explanation: You have designated either an invalid ADDS dictionary name or a dictionary that is not defined.
52	EX_FAIL_CANT_OPEN_ADDS_DB Explanation: You do not have the required privileges to query the ADDS dictionary.
53	EX_FAIL_DB_NOT_SIM Explanation: You tried to use a database that is not compatible with SIM.
54	EX_FAIL_INVLD_DICTNAME Explanation: You have provided an invalid dictionary name.
55	EX_FAIL_BAD_SYMB_VALUE Explanation: You have designated a symbolic value that is either invalid or outside of the allowed range.
56	EX_FAIL_DB_NOT_GNRTD Explanation: You tried to access a database that has not been generated.
57	EX_FAIL_DB_NOT_DEFINED Explanation: The database you want to use is not defined in an ADDS dictionary.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
58	EX_FAIL_LINCDLTD_NVLD Explanation: Your query contains requests that are valid only for retrievals on LINC II databases.
59	EX_FAIL_DB_INQUIRY_ONLY Explanation: The database you want to update is opened for inquiry purposes only.
60	EX_FAIL_ASSIGN_EXISTING Explanation: You tried to assign a value to a preexisting role. Attributes of preexisting roles cannot be assigned values.
61	EX_FAIL_DB_SUSPENDED Explanation: The DMSIISUPPORT library is suspended. The text of this message includes the mix number you should investigate.
62	EX_FAIL_DB_TOO_OLD Explanation: You tried to use a database that was generated using software from an earlier software release than the software release currently in use. Use the SIM Utility UPDATE command to regenerate your database.
63	EX_FAIL_DB_TOO_NEW Explanation: You tried to use a database that was generated using software from a later software release than the software release currently in use. You can only use the database with the newer software release.
64	EX_FAIL_QTEXT_TOO_SMALL Explanation: The first word of the qtext parameter does not equal the number of characters present in the rest of the parameter. The qtext parameter is a string that provides the query text; the first word of the qtext parameter must define the string.
65	EX_FAIL_TYPE_MISMATCH Explanation: The type of data value you are supplying for the item is not correct. For example, you might be supplying an integer when a string is expected.
66	EX_FAIL_STRING_TOO_SMALL Explanation: The string or record parameter you have defined is not large enough for the entry point you are using.
67	EX_FAIL_SQL_NQ_TOO_MANY Explanation: Your unquantified, nested query is selecting more than one row. This message can be returned only if you are querying a database using SQLDB DML.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
68	EX_FAIL_SQL_CHECK_OPTION Explanation: Your query violates the view integrity constraints. This message can be returned only if you are querying an SQLDB database.
69	EX_FAIL_TWO_ROWS_SELECTED Explanation: The SELECT statement in your procedure attempted to select more than one row. This message can be returned only if you are querying a database using SQLDB DML.
70	EX_FAIL_NEED_INDICATOR Explanation: A null value was retrieved, but that there is no indicator parameter present for the value-passing parameter. This message can be returned only if you are querying a database using SQLDB DML.
71	EX_FAIL_DIGITS_LOST Explanation: The precision assigned to the designated parameter is insufficient for the data returned by your query. This message can be returned only if you are querying a database using SQLDB DML.
72	EX_FAIL_INVLD_CLSNAME Explanation: Your query designates an invalid class name.
73	EX_FAIL_SQL_NO_CURRENT_ROW Explanation: The positioned update statement in your module does not reference a cursor positioned on a row. A positioned update statement must reference a cursor that is positioned on a row. This message can be returned only if you are querying a database using SQLDB DML.
74	EX_FAIL_CENTRAL_SUPPORT Explanation: An error occurred while calling a CENTRALSUPPORT function.
75	EX_FAIL_INVALID_NUMERIC Explanation: A program variable contains an incorrectly formed numeric value.
77	EX_FAIL_DISTINCT Explanation: You tried to add a value to a multivalued attribute that was already defined for another occurrence of the multivalued attribute in the same entity.
78	EX_FAIL_RANGE Explanation: You designated a value for an attribute that is outside the range allowed for that attribute. Check the schema definition to determine the valid range of values for the attribute.
79	EX_FAIL_INTL_NO_MESSAGE Explanation: The program could not get a message from the CENTRALSUPPORT library.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
80	EX_FAIL_UPDATE_OF_STATIC Explanation: You tried to assign a new value to an attribute using the STATIC option. An attribute with the STATIC option can be assigned a new value only when its current value is null.
81	EX_FAIL_SQF_TITLE Explanation: You provided an invalid file title for the save query file.
82	EX_FAIL_SQF_AVAILABLE Explanation: You designated a save query file that is not available.
83	EX_FAIL_READING_SQF Explanation: Your program encountered an error while reading the save query file.
84	EX_FAIL_WRITING_SQF Explanation: Your program encountered an error while writing the save query file.
85	EX_FAIL_QID_INFO_TOO_SMALL Explanation: The size you designated for the query_info array (the array in which saved queries are returned to your program) is too small.
87	EX_FAIL_BUFFER_TOO_SMALL Explanation: The buffer you designated for information to be returned to your program is not large enough.
88	EX_FAIL_RETRIEVE_STATEMENT Explanation: You are trying to use an entry point for an update query that can be used only with a retrieval query.
89	EX_FAIL_FORMATS_TOO_SMALL Explanation: The size you designated for the query description parameter (query_desc) is too small.
90	EX_FAIL_INTERNAL_BUFFER Explanation: You cannot call RETRIEVE_DATA after you have called PROCESS_CURSOR. If you want to perform record-at-a-time access, do not call PROCESS_CURSOR.
91	EX_FAIL_USER_BUFFER Explanation: You cannot call PROCESS_CURSOR after you have called RETRIEVE_DATA.
92	EX_FAIL_NO_SML_KEY Explanation: The database cannot be opened because the keys for the SIM library interface (style identifier SML) are not present.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
93	EX_FAIL_NUMSIZE_MISMATCH Explanation: The precision and the scale of the COBOL numeric argument you have designated do not match that of the corresponding module parameter. This message can be returned only if you are processing a database using SQLDB DML.
94	EX_FAIL_CHARSIZE_MISMATCH Explanation: The length of the COBOL alphanumeric argument you have designated does not match the length of the corresponding module parameter. This message can be returned only if you are processing a database using SQLDB DML.
95	EX_FAIL_SQLCODE_MISMATCH Explanation: You have passed an incorrect type or size for the SQLCODE parameter of the indicated procedure. This message can be returned only if you are processing a database using SQLDB DML.
96	EX_FAIL_INVALID_BOOLEAN_VALUE Explanation: You are trying to represent a Boolean value with something other than 0 (zero) or 1. The value used to represent a Boolean value must evaluate to 0 or 1.
97	EX_FAIL_SLIM_NOT_ALLOWED Explanation: You must have the key for the SML product before proceeding.
99	EX_FAIL_QRECORD_TOO_SMALL Explanation: The size you designated for the query description parameter (query_record) is too small.
103	EX_FAIL_SVEVA_MULT Explanation: Your query was rejected because you tried to assign more than one entity to a single-valued entity-valued attribute (EVA).
104	EX_FAIL_MISSING_REQUIRED Explanation: You tried to update a required attribute but no value was assigned.
105	EX_FAIL_BAD_DATE Explanation: The date value was invalid or did not follow the form MM/DD/YY or MM/DD/YYYY.
106	EX_FAIL_BAD_TIME Explanation: The time value was invalid or did not follow the form HH:MM:SS or HH:MM.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
107	EX_FAIL_BAD_DATE_INCR Explanation: The date increment you requested produced an invalid date. That is, the date is either less than 00/00/000 or greater than 12/31/9999.
108	EX_FAIL_RATSELECTIONFAILEDV Explanation: An entity was not selected because the Remap and Test (RAT) process failed.
109	EX_FAIL_NOCURRENTMASTERV Explanation: The operation you requested failed because there is no current master entity.
110	EX_FAIL_INVALIDSYMBOLICSTRINGV Explanation: The query failed because it uses an invalid symbolic string.
111	EX_FAIL_DUPONEORMOREV Explanation: You tried to duplicate an attribute that does not allow duplicates.
112	EX_FAIL_DUPUNIQUEV Explanation: You tried to assign a value that already exists to a unique attribute.
113	EX_FAIL_DUPDISTINCTV Explanation: You tried to assign a value to an attribute that must be distinct.
114	EX_FAIL_USERLIMITEXCEEDEDV Explanation: The user limit was exceeded.
115	EX_FAIL_CURSORLIMITEXCEEDEDV Explanation: The cursor limit was exceeded.
116	EX_FAIL_INVALIDLUCNBRV Explanation: The logical underlying component (LUC) number of a SIM component is invalid.
117	EX_FAIL_SCHEMAERRORV Explanation: The schema is incorrect.
118	EX_FAIL_NOTSUPPORTEDV Explanation: You tried to use a feature that is not supported.
119	EX_FAIL_INVALIDDBTITLEV Explanation: You tried to use a database title that is invalid.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
120	EX_FAIL_NODBWITHGIVENTITLEV Explanation: You tried to use a title of a database that does not exist.
121	EX_FAIL_DBLIMITEXCEEDEDV Explanation: The database limit was exceeded.
122	EX_FAIL_CFERRORV Explanation: The database control file has an error.
123	EXC_FAIL_RATCOMPILEFAILEDV Explanation: The compilation of the RAT process failed.
124	EX_FAIL_FEATURENOTCOMPILEDV Explanation: You tried to use a feature that is not compiled.
125	EX_FAIL_INVALIDSECSPACEV Explanation: The security space was invalid.
126	EX_FAIL_CLASSATTNFERRORV Explanation: Your query produced an incorrect NOTFOUND message while trying to access a class attribute.
127	EX_FAIL_TANKFILEIOERRORV Explanation: An internal file used by the DMSIISUPPORT library has an input/output error.
128	EX_FAIL_UNLOCKDATABASEERRORV Explanation: The program produced an error while unlocking the database during a schema change operation.
129	EX_FAIL_LOCKDATABASEERRORV Explanation: The program produced an error while locking the database during a schema change operation.
130	EX_FAIL_DIRECTORYINTEGRITYERRORV Explanation: A SIM directory update has failed. The record to be written to the SIM directory is incorrect
131	EX_FAIL_GLOBALSURROGATELOADV Explanation: The system cannot determine the value to allocate to the next surrogate.
132	EX_FAIL_MAXMAPPERRATSEXCEEDEDV Explanation: The limit for the DMSIISUPPORT library dynamic code has been reached.
133	EX_FAIL_INVALIDDBNBRV Explanation: You tried to use a database number that is invalid.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
134	EX_FAIL_DBNOTOPENEDV Explanation: You tried to access a database that is not open.
135	EX_FAIL_RATFAULTEDV Explanation: An SCODE fault has occurred.
136	EX_FAIL_NOTLOCKEDV Explanation: The system did not lock an entity before attempting an update.
137	EX_FAIL_INVALIDMASTERV Explanation: The master record is invalid.
138	EX_FAIL_INVALIDCURSORV Explanation: The cursor is invalid.
139	EX_FAIL_INVALIDPARAMETERV Explanation: The parameter is invalid.
140	EX_FAIL_DBALREADYOPENEDV Explanation: You tried to open a database that is already open.
141	EX_FAIL_DBFOROMLSYNTAXONLYV Explanation: The database can be accessed only for compiling queries.
142	EX_FAIL_INVALIDDMSIISUPPORTV Explanation: The DMSIISUPPORT library that you are using does not match the rest of your SIM software.
143	EX_FAIL_NOACCESSV Explanation: You tried to use a class for which you have no privileges.
144	EX_FAIL_NOACCESSONSOMEATTRIBUTESV Explanation: You tried to use an attribute for which you have no privileges.
145	EX_FAIL_WRONGVERSIONV Explanation: You are trying to use software that is incompatible with your other software.
146	EX_FAIL_SUBCLASSALREADYEXISTSV Explanation: You tried to create a subrole that already exists for an entity.
147	EX_FAIL_SUPERCLASSMISSINGV Explanation: You tried to use a superclass entity that is missing.
148	EX_FAIL_CANNOTCOMMITV Explanation: A transaction was not completed.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
149	EX_FAIL_CANNOTCONTINUEV Explanation: The program cannot continue.
150	EX_FAIL_MUSTROLLBACKV Explanation: A transaction failed and the only way to correct the database is to roll back the transaction.
151	EX_FAIL_MVEVACARDLIMITEXCEEDEDV Explanation: You tried to assign a value that would have exceeded the maximum number of occurrences for an entity-valued attribute (EVA).
152	EX_FAIL_MVDVACARDLIMITEXCEEDEDV Explanation: You tried to assign a value that would have exceeded the maximum number of occurrences for a data-valued attribute.
153	EX_FAIL_SUBROLECARDLIMITEXCEEDEDV Explanation: You tried to insert a subclass that would have exceeded the maximum number of occurrences for a multivalued subrole.
154	EX_FAIL_REQUIREDSUBROLEMISSINGV Explanation: You tried to create an entity that was missing a required subrole. In this case, the subrole must be inserted from the perspective of the subclass.
155	EX_FAIL_REQUIREDMVDVAMISSINGV Explanation: Your update failed because a required multivalued data-valued attribute is missing.
156	EX_FAIL_REQUIREDMVEVAMISSINGV Explanation: Your update failed because a required multivalued entity-valued attribute (EVA) is missing.
157	EX_FAIL_REQUIREDEVAMISSINGV Explanation: Your update failed because a required single-valued entity-valued attribute (EVA) is missing.
158	EX_FAIL_MISSINGSOURCEVALUEV Explanation: A required source value is missing for your update of a foreign key EVA. This error can only occur if you are querying a DMSII database processed by DMS.View.
159	EX_FAIL_DICTNOTPRESENTV Explanation: You tried to use a dictionary that does not exist.
160	EX_FAIL_DICTFILELAYOUTERRV Explanation: The program produced an error while trying to access a file description in ADDS.

Table B–3. Failure Messages (cont.)

Subcategory	Subcategory Mnemonic/Explanation
161	EX_FAIL_BUILDTABLESERRV Explanation: SYSTEM/SIM/FILESUPPORT received an error while building its internal tables.
162	EX_FAIL_DATABASEINREORGV Explanation: You tried to open a database that was locked by a pending schema change.
163	EX_FAIL_ONLYONEUTILITYALLOWEDV Explanation: You tried to run a SIM Utility for a database that already had one running.
164	EX_FAIL_DATABASENOTSIMCAPABLEV Explanation: You tried to query a database that is not accessible using SIM or SQLDB.
165	EX_FAIL_INCOMPATIBLEDIRECTORYV Explanation: The SIM directory for your database is not compatible with your software.
166	EX_FAIL_GENDIRECTORYMISSINGV Explanation: The generated SIM directory for your database is not available.
167	EX_FAIL_INVALIDDIRECTORYV Explanation: The SIM directory for your database is not valid.
168	EX_FAIL_NODMSUPPORTV Explanation: The DMSUPPORT library for your database is not available.
169	EX_FAIL_DMSUPPORTNOTUSABLEV Explanation: The DMSUPPORT library for your database is not usable.
170	EX_FAIL_VALDIRECTORYMISSINGV Explanation: The validated SIM directory for your database is not available.
171	EX_FAIL_SIMDIRECTORYMISSINGV Explanation: The DESCRIPTION/SIMDIR/SIM-DIRECTORY file is not available.

Category 4: EX_SYSTEM

This category indicates exceptions that are of a fatal nature to your program and possibly to SIM. If you receive these exceptions you should either close the database, or issue a DS command to stop your program. Table B–4 describes the subcategories associated with category 4.

Table B–4. Fatal Messages

Subcategory	Subcategory Mnemonic
0	Explanation: A fatal system error occurred. There is no additional information available with this subcategory.
1	EX_SYS_PROTOCOL Explanation: A software protocol violation occurred. SIM has encountered an internal fault while processing your request. Please file a UCF to report this problem.
2	EX_SYS_NOT_IMPLEMENTED Explanation: You attempted to use a feature that is not currently implemented.
3	EX_SYS_FAULT Explanation: SIM encountered an internal fault while processing your request.
4	EX_SYS_INTEGRITY Explanation: A system integrity failure occurred. This message is followed by another message that provides more detailed information.
5	EX_SYS_LOGICAL_DB Explanation: An error occurred during the logical database processing. Use the SIM_NEXT_EXCEPTION_INFO entry point to access more information. If this error occurs there is probably a problem with the SIM/DRIVER or the SIM/DMSIISUPPORT programs. Refer to the result word in the exception_info record field.
6	EX_SYS_PHYSICAL_DB Explanation: An error occurred during the physical database processing. Use the SIM_NEXT_EXCEPTION_INFO entry point to access more information. This error is usually related to a DMSII problem. Refer to the result word in the exception_info record field.
7	EX_SYS_ADDS Explanation: An error occurred during the processing of a data dictionary request. This message provides an ADDS error code and subcode. Refer to the SYMBOL/ADDS/PROPERTIES file.

Table B–4. Fatal Messages (cont.)

Subcategory	Subcategory Mnemonic
8	EX_SYS_WORKFILE Explanation: A work file failure occurred. This message provides an error code and a subcode that are returned from the I/O subsystem. Refer to the *System Messages Manual* for more information.
9	EX_SYS_MAPPER Explanation: SIM encountered an internal fault while processing your request. Please file a UCF to report this problem.

Appendix C
Description of the ORGANIZATION Database

The ORGANIZATION database is used throughout this guide to illustrate the use of OML. Figure C–1 shows the schema of the database, that is, the relationships between the classes in the database. The schema illustration follows these conventions:

- Each oval represents a class.
- Class names are indicated within each oval.
- Broken lines between classes represent superclass-subclass relationships.
- Light lines with arrows represent single-valued EVA relationships between classes.
- Heavy lines with arrows represent multivalued EVA relationships between classes.
- EVA names are indicated above the light and heavy lines.

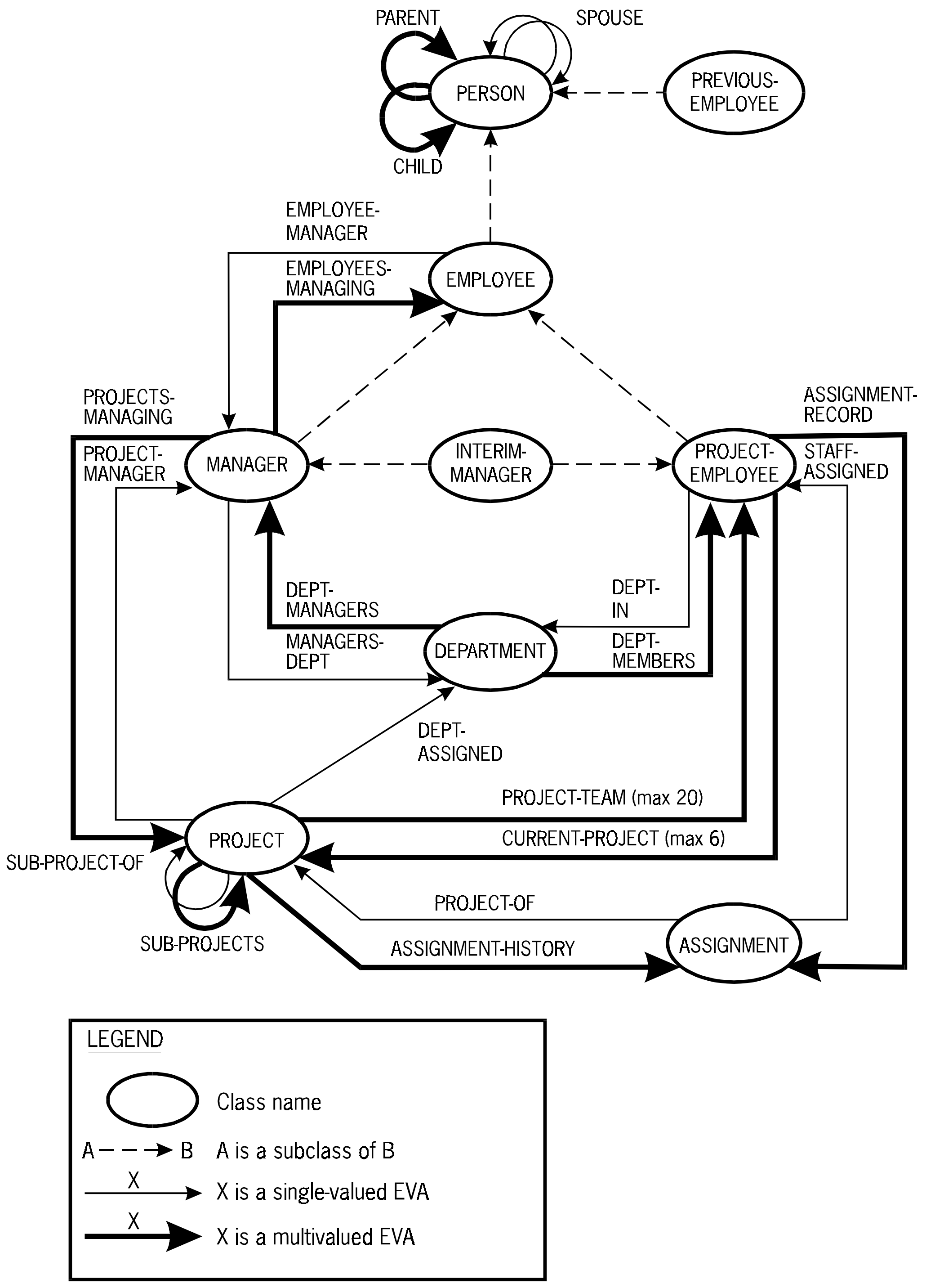


Figure C–1. ORGANIZATION Database Schema

The tables in this appendix describe the user-defined types (UDTs), classes, and attributes that make up the ORGANIZATION database. In the tables, the following conventions are used:

- Uppercase letters are used for class and attribute names and, in some cases, for allowable values.
- Numbers in square brackets indicate the number of characters or digits that can comprise a string or number.
- Words or values within parentheses indicate symbolic values and ranges of those values.

Description of User-Defined Types (UDTs)

Table C–1 describes the UDTs used in the ORGANIZATION database.

Table C–1. Description of UDTs in the ORGANIZATION Database

UDT and Component Names	Base Data Type	Values and Options
FLOOR	Symbolic	(FIRST_FLOOR, SECOND_FLOOR, THIRD_FLOOR, FOURTH_FLOOR, FIFTH_FLOOR)
DEGREE	Symbolic	(HS, BA, BS, MA, MS, PHD); ordered
ADDRESS		
STREET	String [30]	
CITY	String [20]	
STATE	String [2]	
ZIPCODE	String [9]	
NAME_OF_PERSON		
FIRST_NAME	String [15]	
MID_INITIAL	String [1]	
LAST_NAME	String [20]	
NEXT_OF_KIN		
RELATIONSHIP	String [20]	
NAME_OF_KIN	NAME_OF_PERSON	
PHONE_NO	String [10]	
RANK_OF_MANAGER	Symbolic	(SUPERVISOR, DEPT_MANAGER, DIV_MANAGER, EXECUTIVE); ordered

Classes and Attributes

The following tables describe the classes and attributes in the ORGANIZATION database.

PERSON Class

Table C–2 describes the immediate attributes of the PERSON class. The PERSON class does not have a superclass, but it has two subclasses: EMPLOYEE and PREVIOUS_EMPLOYEE. The immediate attributes of the PERSON class are inherited attributes of the EMPLOYEE and PREVIOUS_EMPLOYEE classes.

Table C–2. Immediate Attributes of the PERSON Class

Attribute Name	Data Type	Values and Options
EMPLOYED	Subrole	(EMPLOYEE, PREVIOUS_EMPLOYEE)
PERSON_ID	Integer	Unique; required
NAME	NAME_OF_PERSON	Required
BIRTH_DATE	Date	Required
AGE	Integer	(17 to 99)
MARITAL_STATUS	Symbolic	(SINGLE, MARRIED, DIVORCED); initial value is SINGLE
CURRENT_RESIDENCE	Address	Required
NEXTOFKIN	NEXT_OF_KIN	
US_CITIZEN	Boolean	Required
GENDER	Symbolic	(MALE, FEMALE)
SPOUSE	PERSON	Inverse is SPOUSE
CHILD	PERSON	Inverse is PARENT; multivalued
PARENT	PERSON	Inverse is CHILD; multivalued (MAX 2)
EDUCATION		Multivalued
	DEGREE_OBTAINED	DEGREE
	YEAR_OBTAINED	Date
	GPA	Real
ID_INDEX	INDEX ON PERSON	Ascending PERSON_ID
EDUCATION_INDEX	INDEX ON PERSON	Ascending EDUCATION

PREVIOUS_EMPLOYEE Class

Table C–3 describes the immediate attributes of the PREVIOUS_EMPLOYEE class. Because the PREVIOUS_EMPLOYEE class is a subclass of the PERSON class, it also inherits all the attributes of the PERSON class.

Table C–3. Immediate Attributes of the PREVIOUS_EMPLOYEE Class

Attribute Name	Data Type	Values and Options
LEAVE_STATUS	Symbolic	(DISABILITY, LEAVE, RETIRED, QUIT); required
TERMINATION_REASON	String [30]	
LAST_WORK_DATE	Date	
HIRE_DATE	Date	

EMPLOYEE Class

Table C–4 describes the immediate attributes of the EMPLOYEE class. Because the EMPLOYEE class is a subclass of the PERSON class, it also inherits all the attributes of the PERSON class.

Table C–4. Immediate Attributes of the EMPLOYEE Class

Attribute Name	Data Type	Values and Options
PROFESSION	Subrole	(PROJECT_EMPLOYEE, MANAGER); multivalued
EMPLOYEE_ID	Integer	Unique; required
EMPLOYEE_HIRE_DATE	Date	
EMPLOYEE_SALARY	Real	
EMPLOYEE_STATUS	Symbolic	(EXEMPT, NON_EXEMPT)
EMPLOYEE_MANAGER	MANAGER	Inverse is EMPLOYEES_MANAGING

MANAGER Class

Table C–5 describes the immediate attributes of the MANAGER class. Because the MANAGER class is a subclass of the PERSON and EMPLOYEE classes, it also inherits all the attributes of those two classes.

Table C–5. Immediate Attributes of the MANAGER Class

Attribute Name	Data Type	Values and Options
MANAGER_TITLE	RANK_OF_MANAGER	Required
BONUS	Real	
EMPLOYEES_MANAGING	EMPLOYEE	Inverse is EMPLOYEE_MANAGER; multivalued
PROJECTS_MANAGING	PROJECT	Inverse is PROJECT_MANAGER; multivalued
MANAGERS_DEPT	DEPARTMENT	Inverse is DEPT_MANAGERS
TEMP_STATUS	Subrole	(INTERIM_MANAGER)

PROJECT_EMPLOYEE Class

Table C–6 describes the immediate attributes of the PROJECT_EMPLOYEE class. Because the PROJECT_EMPLOYEE class is a subclass of the PERSON and EMPLOYEE classes, it also inherits all the attributes of those two classes.

Table C–6. Immediate Attributes of the PROJECT_EMPLOYEE Class

Attribute Name	Data Type	Values and Options
TITLE	Symbolic	(SENIOR, JUNIOR, STAFF, SPECIALIST); required
OVERALL_RATING	Number [3,1]	
CURRENT_PROJECT	PROJECT	Inverse is PROJECT_TEAM; multivalued (MAX 6); required
DEPT_IN	DEPARTMENT	Inverse is DEPT_MEMBERS; required
ASSIGNMENT_RECORD	ASSIGNMENT	Inverse is STAFF_ASSIGNED; multivalued; required
TEMP_STATUS	Subrole	(INTERIM_MANAGER)

INTERIM_MANAGER Class

Table C–7 describes the immediate attributes of the INTERIM_MANAGER class. Because the INTERIM_MANAGER class is a subclass of the PERSON, EMPLOYEE, and MANAGER classes, it also inherits all the attributes of those three classes.

Table C–7. Immediate Attributes of the INTERIM_MANAGER Class

Attribute Name	Data Type	Values and Options
INTERIM_HISTORY		Multivalued
	START_DATE	Date
	END_DATE	Date

PROJECT Class

Table C–8 describes the immediate attributes of the PROJECT class. In addition, the PROJECT class has the class attribute NEXT_PROJECT_NO, which is a required, unique integer.

Table C–8. Immediate Attributes of the PROJECT Class

Attribute Name	Data Type	Values and Options
PROJECT_NO	Integer	Unique; required
PROJECT_TITLE	String [20]	
PROJECT_TEAM	PROJECT_EMPLOYEE	Inverse is CURRENT_PROJECT; multivalued (MAX 20)
DEPT_ASSIGNED	DEPARTMENT	
PROJECT_MANAGER	MANAGER	Inverse is PROJECTS_MANAGING
ASSIGNMENT_HISTORY	ASSIGNMENT	Inverse is PROJECT_OF; multivalued
SUB_PROJECT_OF	PROJECT	Inverse is SUB_PROJECTS
SUB_PROJECTS	PROJECT	Inverse is SUB_PROJECT_OF; multivalued

DEPARTMENT Class

Table C–9 describes the immediate attributes of the DEPARTMENT class.

Table C–9. Immediate Attributes of the DEPARTMENT Class

Attribute Name	Data Type	Values and Options
DEPT_TITLE	String [20]	
DEPT_NO	Integer	Unique; required
DEPT_LOCATION	FLOOR	
DEPT_MEMBERS	PROJECT_EMPLOYEE	Inverse is DEPT_IN; multivalued
DEPT_MANAGERS	MANAGER	Inverse is MANAGERS_DEPT; multivalued

ASSIGNMENT Class

Table C–10 describes the immediate attributes of the ASSIGNMENT class.

Table C–10. Immediate Attributes of the ASSIGNMENT Class

Attribute Name	Data Type	Values and Options
START_DATE	Date	
END_DATE	Date	
EST_PERSON_HOURS	Number [3]	
RATING	Number [3,1]	
STAFF_ASSIGNED	PROJECT_EMPLOYEE	Inverse is ASSIGNMENT_RECORD
PROJECT_OF	PROJECT	Inverse is ASSIGNMENT_HISTORY
ASSIGNMENT_NO	Integer	Unique; required

Appendix D
Sample Retrieval Queries

The following retrieval queries illustrate some of the more complex features of OML. All the examples are based on the ORGANIZATION database. For a description of the ORGANIZATION database, refer to Appendix C.

Finding the Maximum Number of Subprojects

Use the following query to find the maximum number of subprojects of any project:

```
RETRIEVE MAX(COUNT(TRANSITIVE(SUB_PROJECTS)) OF PROJECT
         WITH (NOT SUB_PROJECT_OF EXISTS))
```

The WITH clause does not affect the data returned by the query, but it does make the query processing quicker. It does so by eliminating the projects that have no subprojects.

Finding the Average Age of Members of a Department

Use the following query to calculate the average age of the employees and of the managers for any department in which more than ten people are assigned to each of the department's projects:

```
RETRIEVE (AVG(AGE OF DEPT_MEMBERS),
          AVG(AGE OF DEPT_MANAGERS))
         OF DEPARTMENT
WHERE ALL(COUNT(PROJECT_TEAM) OF INVERSE(DEPT_ASSIGNED)) > 10
```

Finding the Number of Hours Spent on a Project

Use the following query to find the total number of hours devoted to a project, including the time spent on any subprojects:

```
FROM PROJECT
RETRIEVE EST_PERSON_HOURS OF ASSIGNMENT_HISTORY +
         SUM(EST_PERSON_HOURS OF ASSIGNMENT_HISTORY OF
             TRANSITIVE(SUB_PROJECTS))
```

Finding the Manager of a Project Employee's Project

Use the following query to retrieve the names of any project employee whose manager is also his or her project manager:

```
RETRIEVE NAME OF PROJECT_EMPLOYEE
WHERE EMPLOYEE_MANAGER = PROJECT_MANAGER OF SOME(CURRENT_PROJECT)
```

Finding the Employees Who Are Having Their One-Year Anniversary

Use the following query to identify those employees who, as of the current date, have been with the company for exactly one year:

```
RETRIEVE NAME OF EMPLOYEE
WHERE ELAPSED_DAYS(EMPLOYEE_HIRE_DATE, CURRENT_DATE) = 365
```

Appendix E
Sample Update Queries

When you want to add information to a database, you should follow these steps:

- Determine the information you want to add.
- Determine the information you must supply because of database constraints. For example, if you are adding a new project to the database, you must supply a project number.
- If you need to write more than one query to insert your data into the database, determine the order of the queries. For example, if you are adding a new project employee to the database and you want to assign him or her to a new department, you must define the department before defining the project employee because a project employee must be assigned to a department.

This appendix discusses one method for determining the content and order of a series of update queries to add new information to the ORGANIZATION database.

In this appendix, the task is broken down as follows:

- Defining the scenario
- Identifying the attributes that must be supplied values
- Identifying the order in which the information must be supplied
- Creating the queries

Defining the Scenario

The scenario is that a number of new employees are being added to the database. One of the employees, Carol Minuet, is a new manager, and the other employees are project employees who are going to work for Carol Minuet. All the new employees are going to work on the new project Airlines in the new Investigations department.

Identifying the Required Attributes

Since a manager, a department, a project, and several project employees are being added to the database, the classes that obviously are going to be involved in the query are MANAGER, DEPARTMENT, PROJECT, and PROJECT_EMPLOYEE. When you check to see what information must be added to these classes because of database constraints, it becomes apparent that you must also add information to the ASSIGNMENT class for each project employee being added to the database.

You can identify the required attributes for each class by referring to Appendix C, "Description of the ORGANIZATION Database." Note that if a class has superclasses, you must check the superclasses for their required attributes. The required attributes for the scenario being described in this appendix are listed in Table E–1.

Table E–1. Required Attributes for the Entities Being Added to the Database

Class	Required Attributes
MANAGER	PERSON_ID
	NAME
	BIRTH_DATE
	CURRENT_RESIDENCE
	US_CITIZEN
	EMPLOYEE_ID
	MANAGER_TITLE
PROJECT_EMPLOYEE	PERSON_ID
	NAME
	BIRTH_DATE
	CURRENT_RESIDENCE
	US_CITIZEN
	EMPLOYEE_ID
	TITLE
	CURRENT_PROJECT
	DEPT_IN
	ASSIGNMENT
PROJECT	PROJECT_NO
DEPARTMENT	DEPT_NO
ASSIGNMENT	ASSIGNMENT_NO

Identifying the Order in Which to Add Information

To identify the order in which to add information, you need to check the class dependencies. For example, in this scenario, since the project and the project employees are new, you need to add the project information before you add the project employee information. Since each project employee must also be assigned a unique assignment record, you must also add the assignment record before adding the project employee information. However, you can choose to add all the assignments to the database before adding any project employee information. Or you can choose to establish a loop in which you add the assignment record for a project employee and then the project employee information. You would repeat this loop once for each project employee.

You must also remember that if an EVA assignment is made, the reverse assignment is automatically made. For example, if you assign a manager to a project employee using the EVA *EMPLOYEE_MANAGER*, the information is automatically assigned to the EVA *EMPLOYEES_MANAGING*.

For this scenario, assume that besides the required attribute information, you are also going to want to do the following:

- Connect the Manager (Carol Minuet) with her employees.
- Connect Carol Minuet with her project, Airlines, and her department, Investigation.
- Connect the project Airlines with the department Investigation and with an assignment history.

You can satisfy all these requirements by entering the information in the following order:

1. Create the department Investigation.
2. Create the project Airlines.
3. Create one assignment for each project employee you are going to enter in the database. In this example, five project employees are being entered into the database, so five unique assignment identifiers are required. Each of the assignment records must be associated with the project Airlines.
4. Enter information about Carol Minuet as a manager; this information includes assigning her to the department Investigation and to the project Airlines. At this point, you do not assign her any employees.
5. Enter information about the five new project employees, including information about their project, department, assignment, and manager.

Creating the Queries

If, during the course of your planning, you identify some attributes that require unique values, you might choose to form a retrieval query to identify the values that are already in use.

Once you establish the order and content of the update queries, you can form the actual queries. Following are a set of queries that can be used to accomplish the task defined in this appendix.

Establishing the Department

The following query creates the department Investigation and assigns it the number 98564:

```
INSERT DEPARTMENT
(
DEPT_TITLE:= "Investigation",
DEPT_NO:= 98564
)
```

Establishing the Project

The following query creates the project Airlines and assigns it the unique identifier 70770:

```
INSERT PROJECT
(
PROJECT_NO:= 70770,
PROJECT_TITLE:="Airlines",
DEPT_ASSIGNED:= DEPARTMENT WITH (DEPT_TITLE = "Investigation")
)
```

Establishing the Assignments

The following query creates the assignment number 22334 and connects it to the project Airlines. This query is repeated once for each of the five project employees being assigned to the Airlines project. You must provide a different assignment number for each of the project employees.

```
INSERT ASSIGNMENT
(
ASSIGNMENT_NO:= 22334,
PROJECT_OF:= PROJECT WITH (PROJECT_TITLE = "Airlines")
)
```

Establishing the Manager

The following query enters information about Carol Minuet into the database. It also associates her with the department Investigation and the project Airlines.

```
INSERT MANAGER
(
PERSON_ID:= 9458674,
NAME:= (LAST_NAME:="Minuet",
         FIRST_NAME:= "Carol"
      ),
BIRTH_DATE:= 3/5/48,
CURRENT_RESIDENCE:= (STREET:="24Rippling Brook",
                     CITY:= "Pinner",
                     STATE:= "MI",
                     ZIPCODE:= "45364"
                    ),
US_CITIZEN:= FALSE,
EMPLOYEE_ID:= 45832,
MANAGER_TITLE:= SUPERVISOR,
PROJECTS_MANAGING:= INCLUDE PROJECT WITH (PROJECT_TITLE = "Airlines"),
MANAGERS_DEPT:= DEPARTMENT WITH (DEPT_TITLE = "Investigation")
)
```

Note: *When you associate a project with a manager, you must use the INCLUDE keyword, because PROJECTS_MANAGING is a multivalued EVA. However, when you associate a department with a manager, do not use the INCLUDE keyword because MANAGERS_DEPT is a single-valued EVA.*

Establishing the Project Employees

The following query enters information about the project employee Peter Salt into the database and associates him with the department Investigation, the project Airlines, the assignment number 22334, and the manager Carol Minuet. This query must be repeated once for each project employee being added to the database. When you change the values in the query for each project employee, remember to change the assignment number.

```
INSERT PROJECT_EMPLOYEE
(
PERSON_ID:= 9238574,
NAME:= (LAST_NAME:= "Salt",
         FIRST_NAME:= "Peter"
      ),
BIRTH_DATE:= 7/8/58,
CURRENT_RESIDENCE:= (STREET:= "92 Joshua Oak",
                     CITY:= "Pinner",
                     STATE:= "MI",
                     ZIPCODE:= "45359"
                    ),
US_CITIZEN:= TRUE,
EMPLOYEE_ID:= 45857,
TITLE:= STAFF,
CURRENT_PROJECT:= INCLUDE PROJECT WITH (PROJECT_TITLE = "Airlines"),
DEPT_IN:= DEPARTMENT WITH (DEPT_TITLE = "Investigation"),
ASSIGNMENT_RECORD:= INCLUDE ASSIGNMENT WITH (ASSIGNMENT_NO = 22334),
EMPLOYEE_MANAGER:= MANAGER WITH (LAST_NAME OF NAME = "Minuet" AND
                                  FIRST_NAME OF NAME = "Carol")
)
```

Appendix F
Quick-Reference Query Syntax

This appendix provides the syntax you should use for your queries. Section 3, “Structuring Retrieval Queries,” Section 4, “Formatting the Output from Retrieval Queries,” and Section 5, “Structuring Update Queries,” provide detailed information and examples of queries. The information presented here is for quick-reference purposes only.

Retrieval Query Syntax

The following syntax diagrams describe the format of a retrieval query. For more detailed information, refer to Section 3, “Structuring Retrieval Queries,” and Section 4, “Formatting the Output from Retrieval Queries.”

```
──┬───────────────────────────────────┬─ RETRIEVE ──────────────────→
  │          ┌←───────── , ──────┐   │
  └─ FROM ───┴─<perspective class>─┴──┘

→─┬─────────────────────────┬─┌←────── , ─────┐
  │                         │ └─<target list>─┴──────────────────────→
  ├─ TABULAR ─┬─────────────┤
  │           └─ DISTINCT ─┤
  └─ STRUCTURED ───────────┘

→─ ORDERED BY ──<order items>─────────────────────────────────────────→
→─┬───────────────────────────────────────────┬───────────────────────┤
  └─ WHERE ──<global selection expression>─┘
```

<perspective class>

```
──┌←──────────── , ──────────────┐
──┴─<class>──┬─────────────────┬─┴─────────────────────────────────────┤
             └─ OF ──<database>─┘
```

<target list>

```
──┌←─────────────── , ───────────────┐
──┴─┬─<attribute definition>─────────┬┴─────────────────────────────────┤
    └─<target attribute expression>─┘
```

<attribute definition>

```
──┬─<attribute>─┬─┬────────────────────────────────┬──────────→
  └─ * ─────────┘ │ ┌←─────────────────────────┐   │
                  └─┴─ OF ─<compound attribute>─┴─┘

→─┬────────────────────┬─┬──────────────┬─┬─────────────────┬──→
  │ ┌←───────────┐     │ └─ OF ─<class>─┘ └─ OF ─<database>─┘
  └─┴─ OF ─<EVA>─┴─────┘

→─┬───────────────────────────────────┬────────────────────────┤
  └─ WITH ─<local filter expression>──┘
```

<order items>

```
──┬───────────────┬─┬──────────────┬─┬─<attribute>──────────────┬───────┤
  ├─ ASCENDING ───┤ ├─ COLLATING ──┤ └─<target list item number>─┘
  └─ DESCENDING ──┘ ├─ BINARY ─────┤
                    └─ ORDERING ───┘
```

Update Query Syntax

The following diagrams present the syntax for different update queries. For more detailed information, refer to Section 5, "Structuring Update Queries."

Understanding Basic Update Query Syntax

```
── MODIFY ─┬────────────────────────┬─<class>─┬──────────────────┬──────→
           └ LIMIT ── = ─┬─ ALL ─────┘          └ OF ──<database>─┘
                         └─<integer>─┘

           ┌←─────────────────────── , ─────────────────────────┐
→─ ( ──────┴<attribute>── := ─┬──────────┬─┬────────────────┬──┴─ ) ───────→
                              ├ EXCLUDE ─┤ └─<expression1>──┘
                              └ INCLUDE ─┘

→─ WHERE ─┬─ TRUE ──────────┬──────────────────────────────────────────┤
          └─<expression2>───┘
```

Adding Values to Single-Valued Attributes

```
── MODIFY ─┬────────────────────────┬─<class>─┬──────────────────┬──────→
           └ LIMIT ── = ─┬─ ALL ─────┘          └ OF ──<database>─┘
                         └─<integer>─┘

           ┌←────────────────────── , ──────────────────┐
→─ ( ──────┴<attribute>── := ──<expression1>────────────┴─ ) ── WHERE ──────────→

→─┬─ TRUE ──────────┬──────────────────────────────────────────────────┤
  └─<expression2>───┘
```

Adding Values to Multivalued Attributes

```
── MODIFY ─┬────────────────────────┬─<class>─┬──────────────────┬─ ( ──→
           └ LIMIT ── = ─┬─ ALL ─────┘          └ OF ──<database>─┘
                         └─<integer>─┘

  ┌←──────────────────────────────────── , ─────────────────────────────────────┐
  │                                                      ┌←────── , ──────┐      │
→─┴─<multivalued attribute>── := ── INCLUDE ── ( ───────┴─<expression1>──┴─ ) ──┴──→

→─ ) ── WHERE ─┬─ TRUE ──────────┬────────────────────────────────────────┤
               └─<expression2>───┘
```

Adding Values to Single-Valued EVAs

```
── MODIFY ──<class>─┬──────────────────┬─ ( ──<EVA>── := ──────────────→
                    └ OF ──<database>──┘

→─<class pointed to>── WITH ── ( ──<expression1>── ) ── ) ── WHERE ────→

→─<expression2>────────────────────────────────────────────────────────┤
```

Deleting Values from Single-Valued Attributes

```
── MODIFY ─┬────────────────────────────┬─<class>─┬──────────────────┬────→
           └─ LIMIT ── = ─┬─ ALL ──────┘          └─ OF ──<database>─┘
                          └─<integer>──┘

→─ ( ─┬<────────────── , ─────────────┐─ ) ── WHERE ─┬─ TRUE ─────────┬──┤
      └─<attribute>── := ── EXCLUDE ──┘              └─<expression>───┘
```

Deleting Values from Multivalued Attributes

```
── MODIFY ─┬────────────────────────────┬─<class>─┬──────────────────┬────→
           └─ LIMIT ── = ─┬─ ALL ──────┘          └─ OF ──<database>─┘
                          └─<integer>──┘

→─ ( ──<expression defining occurrences>── ) ── WHERE ─────────────────→

→─┬─ TRUE ─────────┬───────────────────────────────────────────────────┤
  └─<expression1>──┘
```

<expression defining occurrences>

```
──┬<──────────────────────────────────── , ──────────────────────────────┐──┤
  └─<attribute>── := ── EXCLUDE ─┬──────────────────┬─┬──────────────────┬─┘
                                 └─<limit clause>───┘ └─<local filter>───┘
```

<limit clause>

```
── LIMIT ── = ──<integer>──────────────────────────────────────────────┤
```

<local filter>

```
──<attribute>── WITH ── ( ──<expression2>── ) ─────────────────────────┤
```

Deleting Values from Single-Valued EVAs

```
── MODIFY ──<class>─┬──────────────────┬─ ( ──<EVA>── := ── EXCLUDE ───→
                    └─ OF ──<database>─┘

→─<EVA>─┬───────────────────────────────────┬─ ) ── WHERE ──<expression>─┤
        └─ ( ── WITH ──<expression1>── ) ───┘
```

Deleting Entities

```
── DELETE ─┬────────────────────────────┬─<class>─┬──────────────────┬────→
           └─ LIMIT ── = ──<integer>───┘          └─ OF ──<database>─┘

→─ WHERE ──<expression defining entities to be deleted>────────────────┤
```

Replacing Values in Single-Valued Attributes

```
── MODIFY ─┬──────────────────────────────┬─<class>─┬───────────────────┬──────→
           └─ LIMIT ── = ─┬─ ALL ─────────┤         └─ OF ──<database>──┘
                          └─<integer>─────┘

      ┌←───────────────── , ──────────────────┐
→─ ( ─┴─<attribute>── := ──<expression1>──────┴─ ) ── WHERE ──────────────→

→─┬─ TRUE ──────────┬──────────────────────────────────────────────────┤
  └─<expression2>───┘
```

Replacing Values in Multivalued Attributes

To replace values in multivalued attributes requires two steps: removing values and then adding values. The syntax required for the two steps are as follows:

Step 1: Removing Values

```
── MODIFY ─┬──────────────────────────────┬─<class>─┬───────────────────┬──────→
           └─ LIMIT ── = ─┬─ ALL ─────────┤         └─ OF ──<database>──┘
                          └─<integer>─────┘

→─ ( ──<expression defining occurrences>── ) ── WHERE ──────────────→

→─┬─ TRUE ──────────┬──────────────────────────────────────────────────┤
  └─<expression1>───┘
```

<expression defining occurrences>

```
   ┌←──────────────────────────────────── , ────────────────────────────────────┐
───┴─<attribute>── := ── EXCLUDE ─┬───────────────────┬─┬───────────────────┬───┴─┤
                                  └─<limit clause>────┘ └─<local filter>────┘
```

<limit clause>

```
── LIMIT ── = ──<integer>──────────────────────────────────────────────┤
```

<local filter>

```
──<attribute>── WITH ── ( ──<expression2>── ) ──────────────────────────┤
```

Step 2: Adding Values

```
── MODIFY ─┬──────────────────────────────┬─<class>─┬───────────────────┬── ( ───→
           └─ LIMIT ── = ─┬─ ALL ─────────┤         └─ OF ──<database>──┘
                          └─<integer>─────┘

  ┌←──────────────────────────────────────────── , ──────────────────────────────────────┐
  │                                                         ┌←──────── , ────────┐        │
→─┴─<multivalued attribute>── := ── INCLUDE ── ( ───────────┴─<expression1>──────┴─ ) ───┴──→

→─ ) ── WHERE ─┬─ TRUE ──────────┬──────────────────────────────────────────┤
               └─<expression2>───┘
```

Replacing Values in Compound Attributes

```
── MODIFY ─┬─────────────────────────────┬─<class>── ( ──<compound>── := ─→
           └─ LIMIT ── = ─┬─ ALL ────────┘
                          └─<integer>────┘

→─<EXCLUDE clause>── , ──<compound>── := ──<INCLUDE clause>── ) ─────→

→─ WHERE ─┬─ TRUE ───────┬───────────────────────────────────────────┤
          └─<expression>─┘
```

<EXCLUDE clause>

```
── EXCLUDE ──<compound>── WHERE ── ( ──<component expression>── ) ────┤
```

<INCLUDE clause>

```
── INCLUDE ── ( ──────────────────────────────────────────────────────→

   ┌←──────────────────────── , ──────────────────────┐
→──┴─<component>── := ─┬─<value>──────────────────────┬┴── ) ───────┤
                       │┌←──────────── , ───────────┐ │
                       └┴─<component>── := ──<value>┴─┘
```

Replacing Values in Single-Valued EVAs

```
── MODIFY ──<class>─┬──────────────────┬─ ( ──<EVA>── := ────────────→
                    └─ OF ──<database>─┘

→─<class pointed to>── WITH ── ( ──<expression1>── ) ── ) ── WHERE ───→

→─<expression2>──────────────────────────────────────────────────────┤
```

Replacing Values in Multivalued EVAs

```
── MODIFY ──<class>─┬──────────────────┬─ ( ──<EVA>── := ── EXCLUDE ──→
                    └─ OF ──<database>─┘

→─<EVA>─┬──────────────────────────────────┬─ ) ── WHERE ──<expression>─┤
        └─ ( ── WITH ──<expression1>── ) ──┘
```

Replacing Class Attributes

```
── MODIFY ─┬─────────────────────────────┬─<class>─┬──────────────────┬──→
           └─ LIMIT ── = ─┬─ ALL ────────┘         └─ OF ──<database>─┘
                          └─<integer>────┘

             ┌←─────────────────────── , ───────────────────────┐
→─ ( ────────┴─<attribute>── := ─┬─────────────┬┬───────────────┬┴── ) ───────┤
                                 ├─ EXCLUDE ───┤└─<expression1>─┘
                                 └─ INCLUDE ───┘
```

Inserting Entities from One Class to Another Class

```
── INSERT ──<subclass>── FROM ──<superclass>── WHERE ──────────────────→

→─<expression defining entity to be transferred>── ( ──────────────────→

→─<attributes being assigned values>── ) ──────────────────────────────┤
```

Inserting Entities into the Database

```
── INSERT ──<class>─┬────────────────────┬─ ( ──<assignment statements>── ) ─┤
                    └─ OF ──<database>───┘
```

Appendix G
Understanding Railroad Diagrams

This appendix explains railroad diagrams, including the following concepts:

- Paths of a railroad diagram
- Constants and variables
- Constraints

The text describes the elements of the diagrams and provides examples.

Railroad Diagram Concepts

Railroad diagrams are diagrams that show you the standards for combining words and symbols into commands and statements. These diagrams consist of a series of paths that show the allowable structures of the command or statement.

Paths

Paths show the order in which the command or statement is constructed and are represented by horizontal and vertical lines. Many commands and statements have a number of options so the railroad diagram has a number of different paths you can take.

The following example has three paths:

```
── REMOVE ─┬──────────┬────────────────────────────────────────┤
           ├─ SOURCE ─┤
           └─ OBJECT ─┘
```

The three paths in the previous example show the following three possible commands:

- REMOVE
- REMOVE SOURCE
- REMOVE OBJECT

A railroad diagram is as complex as a command or statement requires. Regardless of the level of complexity, all railroad diagrams are visual representations of commands and statements.

Railroad diagrams are intended to show

- Mandatory items
- User-selected items
- Order in which the items must appear
- Number of times an item can be repeated
- Necessary punctuation

Follow the railroad diagrams to understand the correct syntax for commands and statements. The diagrams serve as quick references to the commands and statements.

The following table introduces the elements of a railroad diagram.

Table G–1. Elements of a Railroad Diagram

The diagram element . . .	Indicates an item that . . .
Constant	Must be entered in full or as a specific abbreviation
Variable	Represents data
Constraint	Controls progression through the diagram path

Constants and Variables

A constant is an item that must be entered as it appears in the diagram, either in full or as an allowable abbreviation. If a constant is partially boldfaced, you can abbreviate the constant by

- Entering only the boldfaced letters
- Entering the boldfaced letters plus any of the remaining letters

If no part of the constant is boldfaced, the constant cannot be abbreviated.

Constants are never enclosed in angle brackets (< >) and are in uppercase letters.

A variable is an item that represents data. You can replace the variable with data that meets the requirements of the particular command or statement. When replacing a variable with data, you must follow the rules defined for the particular command or statement.

In railroad diagrams, variables are enclosed in angle brackets.

In the following example, BEGIN and END are constants, whereas <statement list> is a variable. The constant BEGIN can be abbreviated since it is partially boldfaced.

```
── BEGIN ──<statement list>── END ──────────────────────────────────┤
```

Valid abbreviations for BEGIN are

- BE
- BEG
- BEGI

Constraints

Constraints are used in a railroad diagram to control progression through the diagram. Constraints consist of symbols and unique railroad diagram line paths. They include

- Vertical bars
- Percent signs
- Right arrows
- Required items
- User-selected items
- Loops
- Bridges

A description of each item follows.

Vertical Bar

The vertical bar symbol (|) represents the end of a railroad diagram and indicates the command or statement can be followed by another command or statement.

```
── SECONDWORD ── ( ──<arithmetic expression>── ) ───────────────────┤
```

Percent Sign

The percent sign (%) represents the end of a railroad diagram and indicates the command or statement must be on a line by itself.

```
── STOP ────────────────────────────────────────────────────────────%
```

Right Arrow

The right arrow symbol (>)

- Is used when the railroad diagram is too long to fit on one line and must continue on the next
- Appears at the end of the first line, and again at the beginning of the next line

```
── SCALERIGHT ── ( ──<arithmetic expression>── , ─────────────────────→
→─<arithmetic expression>── ) ────────────────────────────────────────┤
```

Required Item

A required item can be

- A constant
- A variable
- Punctuation

If the path you are following contains a required item, you must enter the item in the command or statement; the required item cannot be omitted.

A required item appears on a horizontal line as a single entry or with other items. Required items can also exist on horizontal lines within alternate paths, or nested (lower-level) diagrams.

In the following example, the word EVENT is a required constant and <identifier> is a required variable:

```
── EVENT ──<identifier>───────────────────────────────────────────────┤
```

User-Selected Item

A user-selected item can be

- A constant
- A variable
- Punctuation

User-selected items appear one below the other in a vertical list. You can choose any one of the items from the list. If the list also contains an empty path (solid line) above the other items, none of the choices are required.

In the following railroad diagram, either the plus sign (+) or the minus sign (–) can be entered before the required variable <arithmetic expression>, or the symbols can be disregarded because the diagram also contains an empty path.

```
──┬─────┬─<arithmetic expression>─────────────────────────────────┤
  ├─ + ─┤
  └─ – ─┘
```

Loop

A loop represents an item or a group of items that you can repeat. A loop can span all or part of a railroad diagram. It always consists of at least two horizontal lines, one below the other, connected on both sides by vertical lines. The top line is a right-to-left path that contains information about repeating the loop.

Some loops include a return character. A return character is a character—often a comma (,) or semicolon (;)—that is required before each repetition of a loop. If no return character is included, the items must be separated by one or more spaces.

```
  ┌←────── ; ─────┐
──┴─<field value>─┴──────────────────────────────────────────────┤
```

Bridge

A loop can also include a bridge. A bridge is an integer enclosed in sloping lines (/ \) that

- Shows the maximum number of times the loop can be repeated
- Indicates the number of times you can cross that point in the diagram

The bridge can precede both the contents of the loop and the return character (if any) on the upper line of the loop.

Not all loops have bridges. Those that do not can be repeated any number of times until all valid entries have been used.

In the first bridge example, you can enter LINKAGE or RUNTIME no more than two times. In the second bridge example, you can enter LINKAGE or RUNTIME no more than three times.

```
  ┌←─────────── , ──────┐
──┴─/2\─┬─ LINKAGE ─┬───┴────────────────────────────────────────┤
        └─ RUNTIME ─┘

  ┌←─/2\────────────┐
──┴─┬─ LINKAGE ─┬───┴────────────────────────────────────────────┤
    └─ RUNTIME ─┘
```

In some bridges an asterisk (*) follows the number. The asterisk means that you must cross that point in the diagram at least once. The maximum number of times that you can cross that point is indicated by the number in the bridge.

```
  ┌←──────── , ────────┐
──┴┬─/2*\─ LINKAGE ─┬──┴──────────────────────────────────────┤
   └─ RUNTIME ──────┘
```

In the previous bridge example, you must enter LINKAGE at least once but no more than twice, and you can enter RUNTIME any number of times.

Following the Paths of a Railroad Diagram

The paths of a railroad diagram lead you through the command or statement from beginning to end. Some railroad diagrams have only one path; others have several alternate paths that provide choices in the commands or statements.

The following railroad diagram indicates only one path that requires the constant LINKAGE and the variable <linkage mnemonic>:

```
── LINKAGE ──<linkage mnemonic>───────────────────────────────┤
```

Alternate paths are provided by

- Loops
- User-selected items
- A combination of loops and user-selected items

More complex railroad diagrams can consist of many alternate paths, or nested (lower-level) diagrams, that show a further level of detail.

For example, the following railroad diagram consists of a top path and two alternate paths. The top path includes

- An ampersand (&)
- Constants that are user-selected items

  These constants are within a loop that can be repeated any number of times until all options have been selected.

The first alternative path requires the ampersand and the required constant ADDRESS. The second alternative path requires the ampersand followed by the required constant ALTER and the required variable <new value>.

```
             ┌←────── , ─────┐
── & ──┬──┴── TYPE ─────┬──┴────┬─────────────────────────────────────────┤
       │     ├─ ASCII ────┤        │
       │     ├─ BCL ──────┤        │
       │     ├─ DECIMAL ─┤        │
       │     ├─ EBCDIC ──┤        │
       │     ├─ HEX ──────┤        │
       │     └─ OCTAL ────┘        │
       ├─ ADDRESS ───────────────┤
       └─ ALTER ──<new value>───┘
```

Railroad Diagram Examples with Sample Input

The following examples show five railroad diagrams and possible command and statement constructions based on the paths of these diagrams.

Example 1

<lock statement>

```
── LOCK ── ( ── <file identifier> ── ) ──────────────────────────────┤
```

Table G–2. LOCK Statement Example

Sample Input	Explanation
LOCK (FILE4)	LOCK is a constant and cannot be altered. Because no part of the word is boldfaced, the entire word must be entered. The parentheses are required punctuation, and FILE4 is a sample file identifier.

Example 2

<open statement>

```
── OPEN ─┬────────────┬─<database name>─────────────────────────────────┤
         ├─ INQUIRY ─┤
         └─ UPDATE ──┘
```

Table G–3. OPEN Statement Example

Sample Input	Explanation
OPEN DATABASE1	The constant OPEN is followed by the variable DATABASE1, which is a database name. The railroad diagram shows two user-selected items, INQUIRY and UPDATE. However, because an empty path (solid line) is included, these entries are not required.
OPEN INQUIRY DATABASE1	The constant OPEN is followed by the user-selected constant INQUIRY and the variable DATABASE1.
OPEN UPDATE DATABASE1	The constant OPEN is followed by the user-selected constant UPDATE and the variable DATABASE1.

Example 3

<generate statement>

```
── GENERATE ──<subset>── = ─┬─ NULL ──────────────────────┬──────────────┤
                            └─<subset>─┬──────────────────┤
                                       ├─ AND ─┬─<subset>─┘
                                       ├─ OR ──┤
                                       ├─ + ───┤
                                       └─ – ───┘
```

Table G–4. GENERATE Statement Example

Sample Input	Explanation
GENERATE Z = NULL	The GENERATE constant is followed by the variable Z, an equal sign (=), and the user-selected constant NULL.
GENERATE Z = X	The GENERATE constant is followed by the variable Z, an equal sign, and the user-selected variable X.
GENERATE Z = X AND B	The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the AND command (from the list of user-selected items in the nested path), and a third variable, B.
GENERATE Z = X + B	The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the plus sign (from the list of user-selected items in the nested path), and a third variable, B.

Example 4

<entity reference declaration>

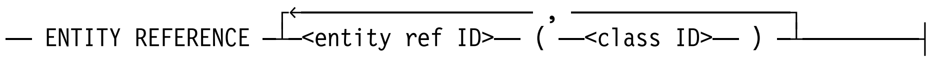
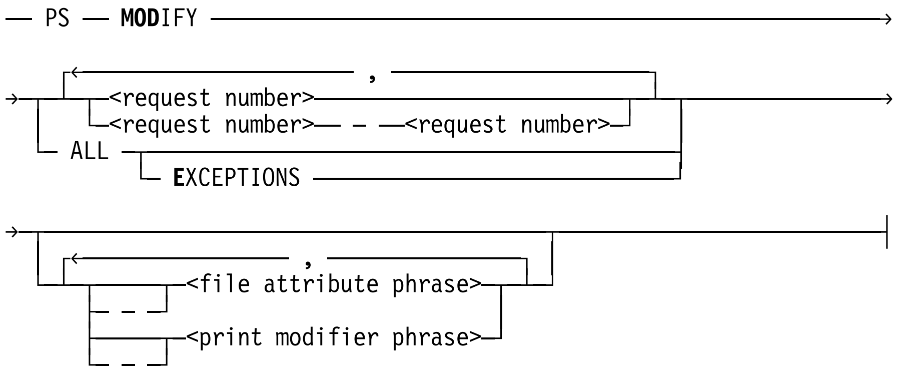


Table G–5. ENTITY REFERENCE Declaration Example

Sample Input	Explanation
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR)	The required item ENTITY REFERENCE is followed by the variable ADVISOR1 and the variable INSTRUCTOR. The parentheses are required.
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR), ADVISOR2 (ASST_INSTRUCTOR)	Because the diagram contains a loop, the pair of variables can be repeated any number of times.

Example 5



Sample Input	Explanation
PS MODIFY 11159	The constants PS and MODIFY are followed by the variable 11159, which is a request number.
PS MODIFY 11159,11160,11163	Because the diagram contains a loop, the variable 11159 can be followed by a comma, the variable 11160, another comma, and the final variable 11163.
PS MOD 11159–11161 DESTINATION = "LP7"	The constants PS and MODIFY are followed by the user-selected variables 11159–11161, which are request numbers, and the user-selected variable DESTINATION = “LP7”, which is a file attribute phrase. Note that the constant MODIFY has been abbreviated to its minimum allowable form.
PS MOD ALL EXCEPTIONS	The constants PS and MODIFY are followed by the user-selected constants ALL and EXCEPTIONS.

Index