UNISYS

Product Information Announcement

o New Release ● Revision o Update o New Mail Code

Title

MCP/AS ALGOL Programming Reference Manual, Volume 2: Product Interfaces (8600 0734–301)

This announces a retitling and reissue of the *ClearPath HMP NX and A Series ALGOL Programming Reference Manual Volume 2: Product Interfaces*. 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:		
	AS165	System:	MCP/AS
		Release:	HMP 4.0 and SSR 45.1
		Date:	June 1998
		Part number:	8600 0734–301

MCP/AS

UNISYS

ALGOL Programming Reference Manual

Volume 2: Product Interfaces

HMP 4.0 and SSR 45.1 June 1998

Printed in US America
Priced Item 8600 0734–301

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the Business Reply Mail form at the back of this document or by addressing remarks to Software Product Information, Unisys Corporation, 25725 Jeronimo Road, Mission Viejo, CA 92691–2792 U.S.A.

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

Contents

Section 7. Using the Semantic Information Manager (SIM) Interface

Tables

About This Manual

Extended ALGOL is a high-level, structured programming language. In addition to implementing virtually all of ALGOL 60, Unisys has developed extensions that enhance the basic capabilities of the language.

Purpose

The programming reference material for Extended ALGOL is divided into two volumes. The *ALGOL Programming Reference Manual*, *Volume 1: Basic Implementation* contains ALGOL language components that can be used for all Unisys products. This volume, Volume 2, contains the ALGOL interfaces specifically developed for the following products:

- Advanced Data Dictionary System (ADDS)
- Communications Management System (COMS)
- Data Management System II (DMSII)
- DMSII Transaction Processing System (TPS)
- Screen Design Facility Plus (SDF Plus)
- Semantic Information Manager (SIM)
- Open/OLTP

ADDS and SIM are part of the InfoExec (Information Executive) family of products. COMS and SDF Plus are members of the InterPro (Interactive Productivity) family of products.

Volume 2 is designed to be used in conjunction with product-specific documentation. Before developing an application program, consult the product's documentation for a discussion of the product, programming considerations, and concepts. (See “Related Product Information” later in this preface for a listing and brief description of these manuals.)

Scope

Volume 2 includes the syntax, explanation, and examples for ALGOL language interfaces with ADDS, COMS, DMSII, TPS, SDF Plus, and SIM. Volume 2 describes the following:

- The reason for developing ALGOL interfaces
- What product interfaces and extensions are available
- Prerequisites for and interrelationships among the interfaces and extensions
- What the interfaces and extensions do
- When and how to use the extensions

Audience

The primary audience for Volume 2 consists of the application programmers responsible for implementing programs that use one or a combination of the specified Unisys products.

Prerequisites

Volume 2 is written for application programmers who are familiar with Extended ALGOL as described in Volume 1. Readers should also be familiar with the product or products for which they are developing applications.

How to Use This Manual

The phrase “Volume 1” refers to the first volume of the *ALGOL Programming Reference Manual* set; “Volume 2” refers to the second volume.

For ALGOL syntax and rules not covered in this volume, refer to Volume 1. Also refer to Volume 1 for information on

- Compiling programs
- The interface to the library facility
- The compile-time facility
- The batch facility
- Data representation
- Run-time format-error messages

Consult the product documentation for product error messages.

Documents that pertain directly to Extended ALGOL and the interfaces covered in this volume are listed under “Related Product Information” in this section.

Railroad syntax diagram notation is used to represent ALGOL syntax. A complete description of this notation can be found in Appendix A, "Understanding Railroad Diagrams."

Organization

After a brief introduction to ALGOL program interfaces, this volume describes the individual program interfaces in product-specific sections. The sections are organized by product. Each section summarizes the product, examines the interface for the product, and when appropriate details the extensions developed for the product. Within a section, extensions are grouped by function. All required syntax and explanations, as well as program examples, are included.

Each section describes how to implement the functions covered in the product programming guide, both when the product is used by itself and when the product is used with other products. Where an extension serves as an interface to enable two products to work together, the products are cross-referenced. When appropriate, requirements and options for using a combination of interfaces are included.

Section 1. Introduction to ALGOL Program Interfaces

This section outlines the Unisys ALGOL interfaces for ADDS, COMS, DMSII, TPS, SDF Plus and SIM. The outline of each interface lists the Unisys extensions that make up the interface and briefly describes each extension. The brief descriptions can be used as a quick reference aid.

Section 2. Using Advanced Data Dictionary System (ADDS) Extensions

This section presents the changes and additions made to ALGOL that enable you to use ADDS to import record data definitions into an ALGOL program.

Section 3. Using Communications Management System (COMS) Features

This section discusses the additions made to ALGOL that make COMS features available to an ALGOL program. The section details how to implement headers, service functions, and DMSII statements.

Section 4. Using the Data Management System II (DMSII) Interface

This section contains the extensions developed for the DMSII interface. These extensions enable you to invoke a database, use data management statements and database items, and handle exception errors.

Section 5. Using DMSII Transaction Processing System (TPS) Extensions

This section examines the changes and additions that enable ALGOL to work with TPS to perform online collection of input and output data for specific transactions.

Section 6. Using the Screen Design Facility Plus (SDF Plus) Interface

This section contains the extensions developed for the SDF Plus interface. These extensions enable you to define a complete form-based user interface for ALGOL application systems.

Section 7. Using the Semantic Information Manager (SIM) Interface

This section describes how ALGOL can be used to manipulate data stored in a SIM database. It covers declaring queries, performing transactions, and handling exceptions.

Section 8. Using TransIT Open/OLTP

This section provides a brief description of TransIT Open/OLTP for enterprise servers. It also contains references that will provide further information regarding using Open/OLTP through ALGOL.

Appendix A. Understanding Railroad Diagrams

This appendix explains how to read and interpret the diagrams used to depict the syntax and use of ALGOL.

Appendix B. Extended ALGOL Reserved Words

This appendix explains and lists the three types of ALGOL reserved words

Results

After reading this document, you will be more familiar with the product interfaces to the Extended ALGOL programming language.

Additionally, you will be able to use this document to find answers to specific questions about the ALGOL product interfaces, and to interpret product interface syntax in existing ALGOL programs.

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.

The following documents provide information that is directly related to the primary subject of this publication.

ALGOL Programming Reference Manual, Volume 1: Basic Implementation

This manual describes the basic features of the Extended ALGOL programming language. This manual is written for programmers who are familiar with programming concepts.

ALGOL Test and Debug System (TADS) Programming Guide

This guide describes the features of ALGOL TADS, an interactive tool used for testing and debugging ALGOL programs and libraries. ALGOL TADS enables the programmer to monitor and control the execution of programs under test and examine the data at any given point during program execution. This guide is written for programmers who are familiar with ALGOL programming language concepts and terms.

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

DMSII Application Program Interfaces Programming Guide

This guide explains how to write effective and efficient application programs that access and manipulate a Data Management System II (DMSII) database using either the DMSII interpretive interface or the DMSII language extensions. This guide is written for application programmers and database administrators who are already familiar with the basic concepts of DMSII.

DMSII Transaction Processing System (TPS) Programming Guide

This guide describes the various modules of TPS and provides information on the TPS library of transaction processing procedures. This guide is intended for experienced systems programmers who are familiar with Data Management System II (DMSII).

File Attributes Programming Reference Manual

This manual contains information about each file attribute and each direct I/O buffer attribute. This manual is written for programmers and operations personnel who need to understand the functionality of a given attribute. The *I/O Subsystem Programming Guide* is a companion manual.

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 those who are familiar with the Data Management System II (DMSII), the Semantic Information Manager (SIM), and the Structured Query Language Database (SQLDB).

InfoExec Semantic Information Manager (SIM) Programming Guide

This guide describes how to use value-added 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.

Open/OLTP Programming Guide

This guide describes the X/Open TX and XATMI interfaces as implemented for use with ALGOL, C, COBOL85, and COBOL 74 application programs. This manual is written for users responsible for designing Open/OLTP application programs that can participate in a distributed transaction processing environment.

Screen Design Facility Plus (SDF Plus) Capabilities Manual

This manual describes the capabilities and benefits of SDF Plus. It gives a general introduction to the product and explains the differences between SDF and SDF Plus. This manual is written for executive and data processing management.

Screen Design Facility Plus (SDF Plus) Installation and Operations Guide

This guide explains how to use SDF Plus to create and maintain a user interface. It gives specific instructions for installing SDF Plus, using the SDF Plus forms, and installing and running a user interface created with SDF Plus.

Screen Design Facility Plus (SDF Plus) Technical Overview

This overview provides the conceptual information needed to use SDF Plus effectively to create user interfaces.

Software Release Installation Guide

This guide explains how to use the Simple Installation (SI) program to install a new software release. The guide also contains specific installation instructions for the current system software release. This guide is written for system administrators, operators, and others responsible for the installation of a new software release.

Task Attributes Programming Reference Manual

This manual describes all available task attributes. It also gives examples of statements for reading and assigning task attributes in various programming languages. The *Task Management Programming Guide* is a companion manual.

X.25 MCS Operations and Programming Reference Manual

This reference manual describes how to use the X.25 message control system (MCS) to interface with packet-switched data networks (PSDNs) that use the X.25 protocol recommended by the Consultative Committee on International Telegraphy and Telephony (CCITT). This manual describes the operations necessary for network data transfer and the functions available for application programming. The manual is written for system administrators, system programmers, and application programmers.

Section 1
Introduction to ALGOL Program Interfaces

A program interface consists of the conventions, protocols, and syntax available in a programming language to manipulate a software product to produce the desired output.

As new software products are developed, the existing program interface components are not always able to manipulate the products for their intended uses. When this occurs, additional program interface components are developed and implemented as required.

The additional program interface components presented in the *ALGOL Programming Reference Manual* are extensions of ALGOL 60. Collectively, ALGOL 60 and Unisys extensions to ALGOL 60 are referred to as Unisys Extended ALGOL. Extensions that are developed for use with a specific product or products are described here, in Volume 2. These products are

- Advanced Data Dictionary System (ADDS)
- Communications Management System (COMS)
- Data Management System II (DMSII)
- DMSII Transaction Processing System (TPS)
- Screen Design Facility Plus (SDF Plus)
- Semantic Information Manager (SIM)

The Open/OLTP product is not listed above because it does not contain extensions.

The following tables name and briefly describe the extensions used with each product. The products are presented alphabetically, one per table. The extensions are ordered alphabetically within the table.

Advanced Data Dictionary System (ADDS) Extensions

The ADDS program interface allows programs to retrieve and incorporate data descriptions as declarations. The ADDS extensions can be used to define records and items. An ALGOL program can use ADDS extensions with COMS, DMSII, and SIM extensions.

Outlined in Table 1–1 are the types, statements, dictionary entity declarations, compiler control options, and functions that comprise the interface. Refer to Section 2, "Using Advanced Data Dictionary System (ADDS) Extensions" for more information.

Table 1–1. ADDS Extensions

Extension	Explanation
Assignment statement	Causes item on the right of the assignment operator to be evaluated and the resulting value to be assigned to the item on the left of the assignment operator.
Data types	Specific types for ADDS items and embedded items.
DICTIONARY option	Establishes the dictionary to be used during compilation.
DICTIONARY ITEM declaration	Declares which nonstructural entity description is to be retrieved.
DICTIONARY RECORD declaration	Declares which record description is to be retrieved.
Entity qualification	Specifies the exact entity to be referenced.
LENGTH function	Returns the length of a specified entity.
OFFSET function	Returns the number of units a specified entity is offset from the beginning of the outermost record.
POINTER function	Returns a pointer to a specified input.

Table 1–1. ADDS Extensions

Extension	Explanation
RANGECHECK option	Causes range checking to be performed at run time.
REPLACE statement	Transfers data from one or more sources to a destination.
SCAN statement	Examines a contiguous portion of data in a field or record.
SIZE function	Returns the size of the array underlying a given record identifier.
STATUS option	Specifies the status of data descriptions to be retrieved from the ADDS.
TYPE declaration	Declares a user-defined type identifier with a format.
Type invocation	Declares records which have their structures stored in a specified type identifier.
UNITS function	Returns the default unit size of the data in the specified entity.

Communications Management System (COMS) Extensions

The COMS extensions, outlined in Table 1–2, allow you to write interactive and batch application programs that run under COMS. The extensions are detailed in Section 3, “Using Communications Management System (COMS) Features.”

Through extensions, the programs can also use ADDS functions, DMSII and SIM synchronized recovery, and COMS service functions. Statements used specifically for synchronized recovery with DMSII are included as COMS extensions. Statements for synchronized recovery with TPS are included in Section 5, “Using DMSII Transaction Processing System (TPS) Extensions.” Statements for synchronized recovery with SIM are included in Section 7, “Using the Semantic Information Manager (SIM) Interface.”

Additional information related to COMS extensions is included in Section 4, “Using the Data Management System II (DMSII) Interface,” and Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

Table 1–2. COMS Extensions

Extension	Explanation
BEGINTRANSACTION statement	Places the program in transaction state. It is used only with audited databases.
COMSRECORD declaration	Retrieves COMS-related format definitions from an external system library.
DISABLE statement	Logically disconnects COMS from a specified destination.
Designator type	Allows programs, running under COMS, to control messages symbolically.
ENABLE statement	Logically connects COMS from a specified destination.
ENDTRANSACTION statement	Takes the program out of transaction state. It is used only with audited databases.
INPUTHEADER declaration	Associates message routing or descriptive information with an identifier when a program receives a message from COMS.
LENGTH function	Returns the length of a specified entity.
MESSAGECOUNT statement	Returns the number of messages in specified queues.

Table 1–2. COMS Extensions

Extension	Explanation
OFFSET function	Returns the number of units a specified entity is offset from the beginning of the outermost record.
OUTPUTHEADER declaration	Associates message routing or descriptive information with an identifier when a program sends a message to COMS.
POINTER function	Returns a pointer to a specified input.
PROCEDURE declaration	Declares a service function entry point in a predeclared library.
RANGECHECK option	Causes range checking to be performed at run time.
RECEIVE statement	Requests a message to be transferred from the COMS program queue to the message area.
RESIZE function	Changes the size of the array underlying a given record identifier.
SIZE function	Returns the size of the array underlying a given record identifier.
SEND statement	Requests a message, or portion of a message, to be transferred from the message area to a specified destination.
UNITS function	Returns the default unit size of the data in the specified entity.

Data Management System II (DMSII) Extensions

The DMSII extensions, outlined in Table 1–3, allow you to declare and use databases in your application programs and to handle exception errors.

BDMSALGOL provides the extensions for declaring and using databases. Programs that declare and use databases still can use the Binder program and the separate compilation (SEPCOMP) facility.

DMSII and SIM databases can be manipulated in the same program. The DMSII extensions must be used with the DMSII databases. The SIM extensions must be used with the SIM databases. COMS can be used with DMSII for synchronized recovery. TPS can also be used with DMSII. ADDS can be used to import definitions.

For the details of the DMSII extensions, consult Section 4, “Using the Data Management System II (DMSII) Interface.”

Additional information related to DMSII extensions is included in Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions,” Section 7, “Using the Semantic Information Manager (SIM) Interface,” Section 5, “Using DMSII Transaction Processing System (TPS) Extensions,” and Section 3, “Using Communications Management System (COMS) Features.”

Table 1–3. DMSII Extensions

Extension	Explanation
ASSIGN statement	Establishes a link from one record in a data set to another record of the same data set.
BEGINTRANSACTION statement	Places a program in transaction state. It is used only with audited databases.
BDMS CLOSE statement	Closes a database when further access is no longer required.
BDMS FREE statement	Unlocks the current record.
BDMS LOCK statement	Finds a record and locks it against a concurrent modification by another user. The MODIFY and BDMS LOCK statements are synonyms.
BDMS OPEN statement	Opens a database for subsequent access and designates the access mode.

Table 1–3. DMSII Extensions

Extension	Explanation
BDMS SET statement	Alters the current path or changes the value of an item in the current record.
CREATE statement	Initializes the user work area of a data set record.
DATABASE declaration	Specifies which database or parts of a database are to be invoked.
database attribute assignment statement	Allows the database to be specified at run time, and allows access to databases under different usercodes and on packs not visible to a task.
DATADICTINFO option	Determines whether information on the use of database structure and items is placed in the object code file.
DELETE statement	Deletes a specific record.
DMTERMINATE statement	Aborts the current action.
DMTEST function	Determines whether an item is null.
ENDTRANSACTION statement	Takes a program out of a transaction state. It is used only with audited databases.
FIND statement	Transfers a record to the work area associated with a data set or global data.
GENERATE statement	Creates a subset in one operation. All subsets must be disjoint bit vectors.
GET statement	Transfers information from the user work area associated with a data set or global data record into program variables or arrays.
INSERT statement	Places a record into a manual subset.
LISTDB option	Determines whether information about the database is included in the printer listing.
MODIFY statement	Finds a record and locks it against a concurrent modification by another user. (See BDMS LOCK statement.)
NODEFINE option	Determines whether defines are expanded in BDMSALGOL constructs.

Table 1–3. DMSII Extensions

Extension	Explanation
PUT statement	Transfers information from program expressions into the user work area associated with a data set or global data record.
RECREATE statement	Partially initializes the user work area.
REMOVE statement	Removes a record from a subset.
Selection expression	Used in DELETE, FIND, MODIFY, and BDMS LOCK statements to identify a specific record in a data set.
STORE statement	Places a new or modified record into a data set.
STRUCTURENUMBER function	Determines the structure number of a data set, set, or subset. It can be used to analyze exception condition results.

DMSII Transaction Processing System (TPS) Extensions

The TPS extensions, outlined in Table 1–4, aid DMSII users in processing a high volume of transactions with synchronized recovery. Statements used specifically for synchronized recovery are available only in BDMSALGOL. Synchronized recovery can be provided through COMS.

Refer to Section 5, “Using the DMSII Transaction Processing System (TPS) Extensions,” for details of the extensions.

Additional information related to DMSII TPS extensions is included in Section 4, “Using the Data Management System II (DMSII) Interface.”

Table 1–4. TPS Extensions

Extension	Explanation
BEGINTRANSACTION statement	Places a program in transaction state. It is used only with audited databases.
BDMS OPEN statement	Opens a database for subsequent access and designates the access mode.
Compile-time functions	Provide access to properties of transaction record formats.
CREATE statement	Initializes a transaction record variable to a particular format.
ENDTRANSACTION statement	Takes a program out of a transaction state. It is used only with audited databases.
Item reference	Identifies and names a transaction record variable.
MIDTRANSACTION statement	Causes the compiler to generate calls on the given procedure prior to the call on the Data Management System (DMS) procedure in Accessroutines.
TRANSACTION BASE declaration	Specifies which transaction base or subbase is to be invoked.

Table 1–4. TPS Extensions

Extension	Explanation
TRANSACTION RECORD declaration	Associates a transaction record variable with a transaction base or subbase.
TRANSACTION RECORD ARRAY declaration	Allows transaction record to be passed to Transaction Library as a parameter.
Transaction record control items	System-defined items maintained by TPS. Control items are defined only after a transaction record has been created.
Transaction record variable assignment	Copies content of one transaction record variable to another transaction record variable in the same transaction base.

Screen Design Facility Plus (SDF Plus) Extensions

The SDF Plus extensions, outlined in Table 1–5, are used to write programs that directly take advantage of SDF Plus. Programs also can be written to take advantage of SDF Plus by way of the COMS interface.

Refer to Section 6, "Using the Screen Design Facility Plus (SDF Plus) Interface," for details of the SDF Plus extensions.

Additional information related to SDF Plus extensions is included in Section 3, "Using Communications Management System (COMS) Features," and Section 2, "Using Advanced Data Dictionary System (ADDS) Extensions."

Table 1–5. SDF Plus Extensions

Extension	Explanation
DICTIONARY option	Establishes the dictionary to be used during compilation.
DICTIONARY FORMRECORDLIBRARY declaration	Invokes an SDF Plus form record library from the specified ADDS dictionary.
Form record number attribute	Provides a means of performing I/O operations on form record libraries to enable individual form records to be specified at run time.
LENGTH function	Returns the length of an entity in the designated units.
OFFSET function	Returns the number of units a specified entity is offset from the beginning of the outermost record.
POINTER function	Returns a pointer to the specified input.
READFORM statement	Causes a form record to be read from the specified remote file and stored in the specified buffer.
RESIZE function	Changes the size of the array underlying a given record identifier.

Table 1–5. SDF Plus Extensions

Extension	Explanation
SIZE function	Returns the size of the array underlying a given record identifier.
Transaction number attribute	Provides a means of performing I/O operations on form record libraries to enable individual transactions to be specified at run time.
UNITS function	Accepts an entity as input and returns, as an integer value, the default unit size expected by the LENGTH and OFFSET functions.
WRITEFORM statement	Causes the contents of a form record to be written to the specified remote file.

Semantic Information Manager (SIM) Extensions

The SIM extensions are used to manipulate the actual data stored in a SIM database. These extensions are outlined in Table 1–6. Library programs can define and access SIM databases. Query records can be passed to and from library procedures.

COMS can be used with SIM for synchronized recovery. SIM and DMSII databases can be manipulated in the same program. The SIM extensions must be used with the SIM databases. The DMSII extensions must be used with the DMSII databases.

Data definitions can be retrieved from ADDS. Several ADDS functions can also be used.

Consult Section 7, “Using the Semantic Information Manager (SIM) Interface,” for details of the SIM extensions, including synchronized recovery.

Additional information relating to SIM extensions is included in Section 3, “Using Communications Management System (COMS) Features,” Section 4,“Using the Data Management System II (DMSII) Interface,” and Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

Table 1–6. SIM Extensions

Extension	Explanation
ABORTTRANSACTION statement	Aborts transaction state. It is used only with audited databases.
BEGINTRANSACTION statement	Places a program in transaction state. It is used only with audited databases.
CANCELTRPOINT statement	Cancels transaction state from a specified point. It is used only with audited databases.
CLOSE statement	Closes the specified database.
database attribute assignment statement	Alters immediate attributes of the perspective class.
DELETE statement	Deletes all entities from the class satisfying the selection expression.
DICTIONARY option	Establishes the dictionary to be used during compilation.

Table 1–6. SIM Extensions

Extension	Explanation
DISCARD statement	Frees control structure resources associated with query.
DMRECORD type	Provides a means to access the data returned by SIM in a RETRIEVE statement.
DM functions	Data Management (DM) arithmetic, string, symbolic, and Boolean functions forwarded to SIM for evaluation.
DMRECORD variable declaration	Structured variable used for information retrieved from SIM.
DM field reference	Accesses information in a DMRECORD variable.
ENDTRANSACTION statement	Takes a program out of transaction state. It is used only with audited databases.
ENTITY REFERENCE declaration	Contains an explicit reference to a database entity.
Exception expression	Provides additional information concerning data management exceptions.
INSERT statement	Causes attribute assignments to be applied to the database and creates a new entity.
LENGTH function	Returns the length of a specified entity.
MODIFY statement	Causes attribute assignments to be applied to the database.
OFFSET function	Returns the number of units a specified entity is offset from the beginning of the outermost record.
OPEN statement	Opens the specified database.
POINTER function	Returns a pointer to a specified input.
QUERY declaration	Declares classes or types used in query.
RANGECHECK option	Causes range checking to be performed at run time.
RESIZE function	Changes the size of the array underlying a given record identifier.

Table 1–6. SIM Extensions

Extension	Explanation
RETRIEVE statement	Retrieves the attributes associated with the query variable.
SAVETRPOINT statement	Saves transaction state from the specified point. It is used only with audited databases.
SELECT statement	Selects a set of entities from the perspective class and associates it with the query variable.
Selection expression	Used to determine which entities from the database are eligible for retrieval, deletion, or modification.
SEMANTIC DATABASE declaration	Specifies which SIM database and classes are available to the program.
SETTOCHILD statement	Adjusts level of the next retrieval away from the root of the query tree.
SETTOPARENT statement	Adjusts level of the next retrieval toward the root of the query tree.
SIZE function	Returns the size of the array underlying a given record identifier.
TYPE declaration	Defines a data structure description which can be used to define a structured variable.
UNITS function	Returns the default unit size of the data in the specified entity.

Section 2
Using Advanced Data Dictionary System (ADDS) Extensions

The Advanced Data Dictionary System (ADDS) provides for the creation, storage, and retrieval of data descriptions. A data description details the characteristics of the data (such as length and type). It does not identify or define the value of the data.

Through ADDS, a program can import record and item definitions. ALGOL programs can incorporate the descriptions as declarations but cannot alter the descriptions.

Consult Section 7, "Using the Semantic Information Manager (SIM) Interface," for an explanation of the relationship between ADDS and SIM. Refer to Section 4, "Using the Data Management System II (DMSII) Interface," for further information on the relationship between ADDS and DMSII.

Consult the *InfoExec ADDS Operations Guide* for a discussion of concepts, procedures, and programming considerations when defining, using, and invoking entities.

Conceptually, ALGOL regards ADDS as a "global" type description storage dictionary. Retrieved entities are seen as type descriptions which are applied to variables being declared in the ALGOL program. Variables declared using the same entity (type description) are distinct variables with separate data spaces.

Additional information related to ADDS extensions is included in Section 7, "Using the Semantic Information Manager (SIM) Interface," and Section 4, "Using the Data Management System II (DMSII) Interface."

Guidelines for Retrieving Data Descriptions

Data descriptions, or metadata, are the stored format descriptions of the data, not the data itself. The data descriptions reside in ADDS.

To retrieve data descriptions, use the DICTIONARY compiler control option to identify the data dictionary where the data descriptions reside. The data dictionary must be specified before the first syntactic element of the program. Once the data dictionary is identified, retrieval of a data description can be performed using a dictionary declaration.

The TYPE declaration can be used as a substitute for DICTIONARY RECORD declarations. The TYPE declaration associates a user-defined name with a record structure description. It can be used multiple times to define data spaces with the same description or to describe parameters to procedures.

Retrieving Descriptions

Use the DICTIONARY RECORD declaration to specify the record description you want to retrieve. Use the DICTIONARY ITEM declaration to specify the item description you want to retrieve. (An item is any nonstructural entity that can be retrieved directly from ADDS.) A data dictionary must be established, using the DICTIONARY option, before using these declarations.

Retrieving Entities of the Same Type

To retrieve several entities of the same type from ADDS, you can declare each corresponding variable separately or you can list the variables in one declaration list. The ordering of entities has no significance.

Record Restrictions

To be compatible for operations such as assignment, record variables must share the same entity description. The variables must be described by the same dictionary entity identifier and entity qualifiers.

Parameters must also share the same type description. The TYPE declaration can be used to retrieve a structure which is then used repeatedly. This guarantees that all declared variables are the same type. Note that TYPE declarations are not necessary for ADDS-retrieved entities.

Records that are described by separate, distinct entities, even if they are identical in format, are not compatible. Even if they match field for field, they are not compatible because they do not share the same type identifier.

Additional information related to data descriptions is included under “Entity Qualifiers” and “TYPE Declaration and Invocation” in this section.

Relating ADDS Data Types to ALGOL

All the data types supported in ADDS are not supported in ALGOL. Some ADDS types exist in ALGOL but cannot be retrieved through the interface. Therefore, ALGOL programs can retrieve ADDS entities only if both of the following conditions are met:

- ALGOL directly supports that data type.
- The ALGOL interface supports retrieving that description.

An entity can be any data type supported by both ADDS and the ALGOL interface to ADDS. The data type of the entity received from ADDS is verified against the type specified in the program.

ADDS entities that are not structures are called "items." The following list shows the ALGOL data types for items:

Binary	EBCDIC array
Boolean	Event
Digit	Real
Display	Task

Some ADDS entities, such as Records, can contain embedded items. Any embedded item within the structure must be one of the following ALGOL data types:

Binary	Display	Integer
Boolean	Double	Real
Digit	EBCDIC array	Record

All embedded items within a structure do not have to be the same data type. However, they must all be supported data types for the structure to be retrieved and acted upon correctly. An error will be reported during compilation if a structure is retrieved that contains a field of a type not supported by ALGOL.

Each retrieved ADDS item and entity type is mapped into an existing ALGOL type.

Mapping ADDS Types to ALGOL Types

Table 2-1 shows which ADDS types can be mapped into which ALGOL types. The table is in alphabetical order, by ALGOL type. In addition, the table notes whether the type can be mapped when the entity is an item or an embedded entity.

Table 2–1. Mapping ADDS Types to ALGOL Types

ALGOL Type	Item	Embedded	ADDS Type
Binary	x	x	Binary Numeric, Binary filler
Boolean	x	x	Boolean
Digits	x	x	Number-Comp, Comp filler
Display	x	x	Display Numeric, Numeric filler
Double		x	Double
EBCDIC array	x	x	Alpha Display, Alpha filler
Event	x		Event
Integer		x	Field
Real	x	x	Real
Record	x	x	Group, Record
Task	x		Task

Additional information related to types is included under “Guidelines for Using ADDS Types,” “Referencing Fields and Records,” and “ALGOL Data Types for ADDS” in this section.

ALGOL Data Types for ADDS

Table 2–2 briefly defines the ALGOL data types that ADDS items and embedded items can be mapped into. Consult Volume 1 for information on data representation and for in-depth definitions.

Table 2–2. Brief Description of ALGOL Data Types

ALGOL Type	Brief Definition
Binary	Can be used to map items and embedded items. A binary is a 48-bit operand in integer format with an optional scale factor. The sign can be ignored. As an embedded entity it is byte-aligned.
Boolean	Can be used to map items and embedded items. An ALGOL "Boolean" aligned on a digit boundary. A 4-bit type, all 4 bits are acted upon.
Digits	Can be used to map items and embedded items. As an embedded entity it is digit-aligned and has 1 to 23 hexadecimal characters with an optional sign and scale factor. In arithmetic expressions it is used as a number. Negative numbers are rounded away from zero (0).
Display	Can be used to map items and embedded items. A display is 1 to 23 EBCDIC numeric characters with an optional sign and an optional scale factor. In arithmetic expressions it is used as a number. Negative numbers are rounded away from zero (0). As an embedded entity it is byte-aligned.
Double	Can be used to map embedded items. An ALGOL "Double", aligned on a byte boundary.
EBCDIC array	Can be used to map items and embedded items. All EBCDIC characters are permitted. As an embedded entity it is aligned on a byte boundary.

Table 2–2. Brief Description of ALGOL Data Types

ALGOL Type	Brief Definition
Event	Can be used to map items. An ALGOL "Event".
Integer	Can be used to map embedded items. An integer is aligned on a digit boundary. It is a 1 to 48 bit integer, left-justifed at the boundary, and padded with "filler" bits on the right to the closest digit boundary. (The filler bits cannot be referenced.) It is unsigned but always considered to be positive.
Real	Can be used to map items and embedded items. An ALGOL "Real," as an embedded entity it is aligned on a byte boundary.
Record	Can be used to map items and embedded items. A sequence of fields, as an embedded entity it is aligned on a byte boundary.
Task	Can be used to map items. An ALGOL "Task".

Additional information related to ADDS and ALGOL data types is included under "Mapping ADDS Types to ALGOL Types," and "Guidelines for Using ADDS Types" in this section.

Guidelines for Using ADDS Types

All actions (reference or assignment) performed on a specified field must be contained within the boundaries of that field. No explicit actions on one field can explicitly or implicitly affect a neighboring field except as provided for by the POINTER function. Within this guideline

- Fields of type EBCDIC array can be used anywhere an EBCDIC array can be used.
- Fields of type Display, Digits, Binary, or Real can be used anywhere an arithmetic primary can be used.
- Fields of type Boolean can be used anywhere a Boolean primary can be used.
- Fields of type Integer can be used anywhere an integer primary can be used.
- Fields of type Double can be used anywhere a double primary can be used.
- Fields of any type filler can never be explicitly referenced.
- Fields of type Record can be used anywhere a record can be used, except where explicitly forbidden.

The ADDS extensions permit bit manipulation and partial reference of Real, Boolean, and Integer fields.

Items can be used where their corresponding field types can be used (as described above).

Arrays of fields are supported. An array of fields with a variable number of elements is treated as an array of fields having the maximum possible number of elements. A variable-length field is treated as a fixed-length field whose length is the maximum possible length of that field. For example, if the length can vary between 5 and 10 digits, a fixed length of 10 is assumed. Redefines are also supported.

Additional information related to the use of ADDS types is included under “POINTER function,” “Mapping ADDS Types to ALGOL Types,” “ALGOL Data Types for ADDS,” “Referencing Fields and Records,” and “RANGECHECK Option: Checking Ranges of Run-Time Values” in this section.

Entity Qualifiers

<entity qualifiers>

```
          ┌←─────────────── , ───────────────┐
— ( ──────┴┬─/1\─ NAME — = —<entity name>─┬──┴─ ) ─────────────────────┤
           └─/1\─┬─<repository qualifiers>─┤
                 └─<database qualifiers>───┘
```

<repository qualifiers>

```
 ┌←──────────────────────────── , ──────────────────────────┐
─┴─┬─/1\─ VERSION — = —<version number>─────────────────┬───┴──────────┤
   ├─/1\─ DIRECTORY — = — '' ─┬─<directory name>─┬─ '' ─┤
   │                          └─ * ──────────────┘      │
   └─/1\─ STATUS — = —<status value>────────────────────┘
```

<database qualifiers>

```
 ┌←───────────────────────── , ─────────────────────────┐
─┴─┬─/1\─ USERCODE — = — '' —<usercode name>— '' ──────┬─┴──────────────┤
   └─/1\─ PACKNAME —  = — ''—<pack name>— ''   ────────┘
```

Explanation

When entities are retrieved, to ensure the retrieval of the correct entity, the entity must be identified in such a way that it cannot be confused with any other entity. In an ALGOL program this is done with entity qualifiers. The entity qualifiers are name, version, directory, and status.

The entity qualifiers are assigned to the entity previously in ADDS. The ALGOL extensions only iterate the information. In the absence of specified entity qualifiers, ADDS will apply default rules to locate and identify the appropriate entity. Qualifiers do not have to be specified if the default rules uniquely identify the entity. Consult the *InfoExec ADDS Operations Guide* for the default rules.

Note: *Enabling the default rules to be applied can cause a previously compilable program to become noncompilable due to the creation of new entities in the dictionary.*

An attempt to retrieve an entity that is not recognized by ADDS results in an error at compile time.

The repository qualifiers identify the exact entity to be retrieved from a repository.You can use database qualifiers when you invoke a SIM database without retrieving it from a repository. To use the database qualifiers, you must not specify a DICTIONARY with a compiler control option, and the SIM configuration must include an optional repository. If a DICTIONARY has not been specified, the repository is optional, and no database qualifiers are specified, the default usercode and packname are used by SIM to locate the database.The usercode name is the usercode of the database control files. It is represented by a constant string that identifies a valid usercode on the system.The pack name designates the pack on which the database control files are stored. It is represented by a constant string that identifies a valid pack on the system.The entity qualifiers identify the exact entity to be retrieved from ADDS. Consult the *InfoExec ADDS Operations Guide* for a discussion of entity name, version, directory, and status and the default search rules.

An entity name is the name of the type description within ADDS. If it is not specified, the value in the identifier declaring the variable is used as the default. Note that an entity name might contain hyphens but an identifier cannot.

Hyphens (-) are permitted only in the entity name construct of an ALGOL declaration. At declaration time, hyphens are translated into underscores (_) within the compiler. An error is generated if, in the same scope, the translated identifier is already declared, or if a later declaration attempts to declare the translated identifier.

A version number is an integer assigned to the entity by the data dictionary.

The directory is a literal that represents a valid directory name recognized by ADDS. The directory name is a maximum of 17 characters. An asterisk (*) explicitly specifies that the entity to be retrieved has no directory name.

The status value enables a particular status to be retrieved. The qualifier specifies the expected status value of the entity and overrides the status specified by the STATUS compiler control option.

Valid status values are TEST and PRODUCTION. No other status can be invoked by the ALGOL compiler.

Additional information related to status values is included under “STATUS Option: Selecting the Status of Descriptions” in this section.

Example

In the following example, all possible qualifiers are used to identify the entity:

```
(NAME = RECORD, VERSION = 123456,
 DIRECTORY = "ACCOUNT", STATUS = TEST)
```

Referencing Fields and Records

<qualified field ID and qualified record ID>

```
                         ┌←─────────────────────────┐
── <record ID> ── . ─────┴─┬─ <field ID> ─────────────┬─┴──────────────────────┤
                           └─ <subscripted field ID> ─┘
```

<subscripted field ID>

```
── <field ID> ── [ ── <subscript> ── ] ──────────────────────────────────┤
```

Explanation

When referencing fields in a record, each field must be uniquely identified. The field is qualified by the record identifier, the field identifier, and, as needed, by a subscript field identifier.

The record ID construct identifies the record that qualifies the field.

The field ID construct identifies the ADDS name for the field. If the field was declared in the record as a subscripted field or as an EBCDIC array field, use the subscripted field ID syntax to specify the occurrence of the field or the element of an EBCDIC array.

The subscript can be any arithmetic expression. Arrays of fields (ADDS occurs) are one-bounded. EBCDIC array fields are zero-bounded.

ADDS field identifiers might contain both underscores and hyphens. However, in ALGOL, underscores must be used in place of the hyphens. This can cause two fields in the same record to have the same name. For example, in ADDS the fields can have the names NEW_ACCOUNT and NEW-ACCOUNT. In ALGOL they are both noted as NEW_ACCOUNT and only the first field of that name can be referenced.

Examples

In the following example, the field MAY is qualified by the record ACCOUNTS:

```
ACCOUNTS.MAY
```

Below, the field MAY is qualified by the form record ACCOUNTS and the form record library GENERALLEDGER.

```
GENERALLEDGER.ACCOUNTS.MAY
```

The following example illustrates how to reference occurrence three in the STUDENT field in the INSTRUCTOR record.

```
INSTRUCTOR.STUDENT[3]
```

The next example shows the syntax to reference character four of occurrence two in the field STUDENT in the record INSTRUCTOR.

```
INSTRUCTOR.STUDENT[2,3]
```

Compiler Control Options

One compiler control option is specific to ADDS: the STATUS option. The DICTIONARY option can be used as an ADDS and SIM extension. The RANGECHECK option can be used as an ADDS extension, as well as both a COMS and SIM extension.

- The DICTIONARY option establishes the dictionary to use during compilation.

Note: *A dictionary must be established before the first executable statement in the program. A program that retrieves an entity must specify a dictionary before it attempts the retrieval. If a dictionary is not previously specified, the program will not compile. Only one dictionary can be used by the program.*

- The STATUS option specifies the status value of the retrieved data description. The status value can be changed as needed. A status value is not required for successful compilation of the program.
- The RANGECHECK option causes the compiler to perform range checking on some run-time values; it is not required for successful compilation of the program.

DICTIONARY Option: Establishing a Data Dictionary

<dictionary option>

```
— DICTIONARY ─┬─────┬─ ''<dictionary ID>  ─┬─────┬─ ''──────────────────────┤
              └─ = ─┘                      └─ . ─┘
```

Explanation

The DICTIONARY compiler control value option establishes the ADDS to use during compilation. The option can be used without retrieving any descriptions from ADDS.

Note: *A data dictionary must be established before the first executable statement. Only one data dictionary can be used by the program. If the program attempts to retrieve a description and a data dictionary was not previously specified, the program does not compile.*

The compiler links to the specified ADDS (system library) when the first executable statement is encountered. The link is ended at the end of the compilation. The data dictionary specified in the first occurrence of a DICTIONARY option is used as the data dictionary. All other occurrences incur warning messages but are otherwise ignored.

If the compiler cannot link to the specified data dictionary, the following error message is generated and the compilation is terminated:

```
DICTIONARY NOT PRESENT OR UNABLE TO LINK
```

The dictionary ID must be constructed from a combination of uppercase letters and digits only.

When you use SIM, the dictionary ID must be the ADDS to which SIM is linked.

Example

In the following example, the dictionary with the name DATADICTIONARY will be used during program compilation:

```
$SET DICTIONARY = "DATADICTIONARY."
```

STATUS Option: Selecting the Status of Descriptions

<status option>

```
 — STATUS — = — <status value> ——————————————————————————————————————————————┤
```

<status value>

```
—┬— TEST ———————┬————————————————————————————————————————————————————————————┤
 ├— PRODUCTION —┤
 └— ANY ————————┘
```

Explanation

The STATUS option is a value option used to specify the status of the data descriptions to be retrieved. The STATUS option can appear anywhere within the program. The value can be changed as often as desired. This option has no effect on declarations which explicitly specify an entity status.

Additional information related to status values is included under “Entity qualifiers,” “Specifying a DICTIONARY RECORD,” and “Specifying a DICTIONARY ITEM” in this section.

Instances where no status value is specified and where the status value is ANY are treated in the same way. Refer to the *InfoExec ADDS Operations Guide* for more complete definitions of status values and default rules.

Examples

In the following example, the dictionary DATADICTIONARY will be used during program compilation. From this dictionary, the system will first try to retrieve the record MAYLEDGER with a PRODUCTION status. If none can be found, the system will try to retrieve the record MAYLEDGER with a TEST status.

```
$SET DICTIONARY = "DATADICTIONARY."
 $SET STATUS=ANY
 DICTIONARY RECORD MAYLEDGER;
```

In the following example, the dictionary DATADICTIONARY will be used during program compilation. From this dictionary, the system will only try to retrieve the record MAYLEDGER with a TEST status.

```
$SET DICTIONARY = "DATADICTIONARY."
 $SET STATUS=TEST
 DICTIONARY RECORD MAYLEDGER;
```

In the following example, the dictionary DATADICTIONARY will be used during program compilation. Although the status option is set to TEST, the system will only retrieve the record MAYLEDGER with a PRODUCTION status because the status is explicitly set in the declaration.

```
$SET DICTIONARY = "DATADICTIONARY."
 $SET STATUS=TEST
 DICTIONARY RECORD MAYLEDGER (STATUS=PRODUCTION);
```

RANGECHECK Option: Checking Ranges of Run-Time Values

<rangecheck option>

```
— RANGECHECK ————————————————————————————————————————————————————————————┤
```

Example

The RANGECHECK option is a Boolean option that causes range checking to be performed at run time. The option is set by default. Use $RESET to reset the option.

The ranges checked include

- During assignments, checking if the numbers assigned into Display, Digits, Integer, and Binary items or fields are too large to be assigned. (This also checks for truncation errors.)
- Checking if subscripts are within the range for arrays of fields and for EBCDIC array fields.

A run-time fault occurs if a value fails a range check; the program is discontinued and an “Invalid Operation” is reported.

In the following example, the RANGECHECK option is reset. The compiler does not perform range checking at run time. This means the compiler emits faster code but permits incorrect assignments or indexing.

```
$RESET RANGECHECK
```

Data Dictionary Declarations

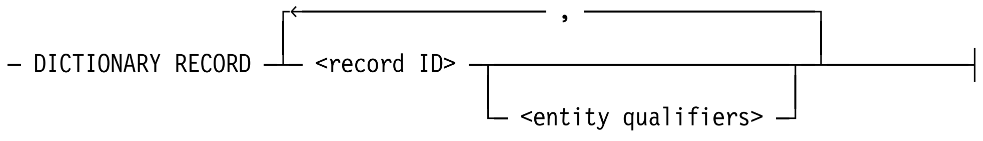
The DICTIONARY RECORD and DICTIONARY ITEM declarations are used to retrieve record descriptions and item descriptions from the specified ADDS.

A data dictionary must be set using the DICTIONARY option before the compiler encounters any data dictionary retrieval declaration.

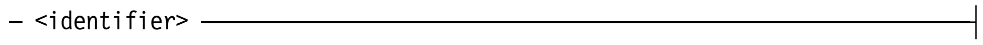
Additional information related to data dictionary declarations is included under "DICTIONARY Option: Establishing a Data Dictionary," and "Guidelines for Retrieving Data Descriptions" in this section.

Specifying a DICTIONARY RECORD

<dictionary record declaration>



<record ID>



Explanation

The DICTIONARY RECORD declaration specifies which record description is to be retrieved from ADDS.

A DICTIONARY RECORD can also be declared using a TYPE declaration and invocation. Because ADDS entities are considered to be global, the TYPE declaration and invocation are not required with ADDS entities.

Additional information related to the DICTIONARY RECORD is included under "TYPE Declaration and Invocation," and "Binding Considerations for ADDS" in this section.

Additional information related to items of the DICTIONARY RECORD is included under "Entity Qualifiers," and "Referencing Fields and Records" in this section.

The record ID is the name within the program of the variable being declared.

The record identifier can be qualified by any or all the entity qualifiers: entity name, version number, directory, and status.

If the identifier is qualified by an entity name, the name identifies the entity within ADDS. If an entity name is not specified, the record ID is used as the entity name.

Hyphens (-) are permitted only in the <entity name> construct. At declaration time, hyphens are translated into underscores (_) within the compiler. An error is generated if, in the same scope, the translated identifier is already declared, or if a later declaration attempts to declare the translated identifier.

More than one record description can be retrieved at one time using a single DICTIONARY RECORD declaration.

Examples

In the following example, the records MONTH, DATE, and YEAR are retrieved from the ADDS with the name DATADICTIONARY.

```
$SET DICTIONARY = "DATADICTIONARY."
 DICTIONARY RECORD MONTHS, DATE, YEAR;
```

In this example, version 2 of record YEARLY stored under the directory ALL is retrieved from the ADDS with the name DATADICTIONARY.

```
$SET DICTIONARY = "DATADICTIONARY."
 DICTIONARY RECORD YEARLY (VERSION = 2, DIRECTORY = "ALL");
```

In the following example, version 2 of record B is retrieved from the ADDS with the the name DATADICTIONARY. The default version of record A is retrieved.

```
$SET DICTIONARY = "DATADICTIONARY."
 DICTIONARY RECORD A, B (VERSION = 2);
```

This example retrieves the record description FACTORY from the data dictionary DATADICTIONARY. The description is applied to the record variable MANUFACTURE.

```
$SET DICTIONARY = "DATADICTIONARY."
 DICTIONARY RECORD MANUFACTURE (NAME = FACTORY);
```

In the following example, several records are declared in distinct declarations, and the same records are declared in one declaration.

```
Separately                  Single Declaration

DICTIONARY RECORD X;        DICTIONARY RECORD L, T, X;
DICTIONARY RECORD T;
DICTIONARY RECORD L;
```

TYPE Declaration and Invocation

<type declaration>

```
                                 ┌←──────────── , ────────────┐
– TYPE  – DICTIONARY RECORD ─┴─ <type ID> ─┬─────────────────────┬─┴────┤
                                           └─ <entity qualifiers> ─┘
```

<type ID>

```
 – <identifier> ──────────────────────────────────────────────────┤
```

<type invocation>

```
              ┌←───── , ────┐
– <type ID> ─┴─ <record ID> ─┴───────────────────────────────────┤
```

Explanation

ADDS entities are assumed to be defined globally to the program. Thus, the TYPE declaration and invocation are not required with ADDS entities. However, their use does provide for a faster compilation when a dictionary record is declared multiple times.

The TYPE declaration associates a user-defined type identifier with a data description and must precede the type invocation. The type invocation declares records that have the structure associated with the type identifier.

In the TYPE declaration, a type identifier is associated with DICTIONARY RECORD declaration. In effect, the type identifier is the name of a record structure description. The TYPE declaration does not create a variable, it simply defines a type identifier that can be used to declare record variables. The variables are declared using the syntax notation shown below.

Only variables that share the same entity description and type are compatible. Records described by separate, distinct entities and identical in content are not compatible if they do not share the same type identifier.

The DICTIONARY RECORD declaration in the TYPE declaration identifies the record to be used as the data definition. When the declaration is used as part of the syntax of a TYPE declaration and invocation, the type ID is the name of the record structure description.

The type identifier is the user-defined name associated with the format. The type ID construct includes the name of the record declared in the TYPE declaration. Each record specified by a record identifier in the type invocation has the structure defined by the type identifier.

The type identifier can be qualified by any or all the entity qualifiers: entity name, version number, directory, and status.

Additional information related to type declarations is included under “Record Restrictions” in this section.

Additional information related to items in the type declaration is included under “Specifying a DICTIONARY RECORD,” “Referencing Fields and Records,” and “Entity Qualifiers” in this section.

Examples

In the example shown below, a TYPE declaration equates the identifier NEWRECORDTYPE with the record structure of INSTRUCTOR. The record PROFESSOR is then defined. By using the type invocation, the structure of INSTRUCTOR becomes the structure of PROFESSOR.

```
TYPE DICTIONARY RECORD NEWRECORDTYPE (NAME=INSTRUCTOR);
 NEWRECORDTYPE PROFESSOR;
```

In the following example, a TYPE declaration equates the identifier MYRECORD with the record structure of PAYABLE. The type invocation is then used to impose the record structure onto the record NEXTPAYABLE.

```
TYPE DICTIONARY RECORD MYRECORD (NAME = PAYABLE,
 VERSION = 123456, DIRECTORY = "ACCOUNTING");
 MYRECORD NEXTPAYABLE;
```

Specifying a DICTIONARY ITEM

<dictionary item declaration>

```
                                ┌←─────────────── , ───────────────┐
─ DICTIONARY ─┬─ REAL ─────────┬┴─ <item ID> ─┬────────────────────────┬┴─┤
              ├─ BOOLEAN ──────┤              └─ <entity qualifiers> ─┘
              ├─ DISPLAY ──────┤
              ├─ DIGITS ───────┤
              ├─ BINARY ───────┤
              ├─ EBCDIC ARRAY ─┤
              ├─ EVENT ────────┤
              └─ TASK ─────────┘
```

Explanation

The DICTIONARY ITEM declaration specifies which item description is to be retrieved from ADDS. An item is an entity that is neither a structure nor embedded in a structure.

Real, Boolean, Display, Digits, Binary, EBCDIC array, Event, and Task are ALGOL-supported types.

The item ID is the name of the item. It can be qualified by name, version number, directory, and status.

Hyphens (-) are permitted only in the entity name construct. At declaration time, hyphens are translated into underscores (_) within the compiler. An error is generated if, in the same scope, the translated identifier is already declared, or if a later declaration attempts to declare the translated identifier.

Additional information related to dictionary items is included under “Entity Qualifiers,” “Referencing Fields and Records,” and “ALGOL Data Types for ADDS” in this section.

Example

In the following example, after establishing DATADICTIONARY, the dictionary items X, Y, and Z are retrieved. All three items are type Real.

```
$SET DICTIONARY = "DATADICTIONARY."
 DICTIONARY REAL X,
 Y (VERSION = 2),
 Z (NAME = A, DIRECTORY = "*");
```

Passing Entities as Parameters

<specification>

```
                  ┌<────── , ─────┐
──┬─ <specifier> ─┴─ <identifier> ─┴──┬──────────────────────────────────┤
  ├─ <procedure specification> ───────┤
  ├─ <array specification> ───────────┤
  ├─ <dictionary entity declaration> ─┤
  ├─ <type invocation> ───────────────┤
  └─ <type declaration> ──────────────┘
```

Explanation

To specify a formal parameter that has a description residing in ADDS, the dictionary entity declaration or type invocation constructs found in the specification construct of the PROCEDURE declaration must be used to declare the formal parameter. Note that the TYPE declaration can be used with the type invocation construct. (The type identifier will not be interpreted as a parameter.)

Dictionary Records, Displays, Digits, and Binaries must be specified in this manner. Dictionary Reals, Booleans, EBCDIC arrays, Tasks, and Events can be specified in this manner or by using normal ALGOL declarations.

Refer to Volume 1 for a full discussion of the PROCEDURE declaration.

All records are passed by name only. The actual and formal parameters must have the same dictionary entity as their type description.

When passing embedded items to items or items to items, the entity type determines the requirements, as shown below:

- Types EBCDIC array, Display, Digits, or Record that are embedded entities:

  When passed by reference, they are passed as a by-value pointer and a lower bound. They cannot be passed by value only. In addition, for records to be compatible, the actual and formal parameters must have the same dictionary type description. For Display and Digits, signs, lengths, and scale information is ignored.

- Types Real, Boolean, Binary, and Integer:

  As fields, all specified types can be passed by value only. As items, types Real and Boolean are treated normally. As items, when passed by name, type Binary can be passed only to type Binary. Sign and scale information is ignored.

As implemented, records are logical structures imposed by the compiler on "*"-bound EBCDIC arrays. Items can be passed as normal ALGOL variables. A field in a record cannot be specified as the formal parameter.

Additional information related to entities is included under ” “Specifying a DICTIONARY RECORD,” “Specifying a DICTIONARY ITEM,” and “TYPE Declaration and Invocation” in this section.

Additional information related to ADDS entities is included under “Relating ADDS Data Types to ALGOL” in this section.

Examples

The following two coding examples can be used to accomplish the same programming task. In the first example, the record REC1 is declared. The formal parameter for procedure P is REC2. REC2 is the same type as REC1. When procedure P is called, REC1 is passed as a formal parameter.

```
$SET DICTIONARY = "DATADICTIONARY."
 BEGIN
 DICTIONARY RECORD REC1 (NAME=X, VERSION=2);
 PROCEDURE P (REC2);
 DICTIONARY RECORD REC2 (NAME=X, VERSION=2);
 BEGIN
 ...
 ...
 END;
 P (REC1)
 ...
 ...
 END.
```

In the following example, the record REC1 is declared. The identifier X is assigned the type. The formal parameter for procedure P is REC2, declared to be type X. By using X, REC2 is noted as the same type as REC1. When procedure P is called, REC1 is passed as a formal parameter.

```
$SET DICTIONARY = "DATADICTIONARY."
 BEGIN
 TYPE DICTIONARY RECORD X (VERSION = 2);
 X REC1;
 ...
 ...
 PROCEDURE P (REC2);
 X REC2;
 BEGIN
 ...
 ...
 END;
 P (REC1);
 ...
 ...
 END.
```

Binding Considerations for ADDS

A DICTIONARY RECORD variable can be bound to another DICTIONARY RECORD variable or to an "*"-bound EBCDIC array. A DICTIONARY RECORD can also be bound to any other record type that can be bound to an "*"-bound EBCDIC array. The Binder program does not check the record structures for compatibility; therefore, it binds DICTIONARY RECORD variables to similarly defined DICTIONARY RECORDs.

Procedures that have DICTIONARY RECORD formal parameters can also be bound, but type checking will not be performed at bind time. The user must ensure that the types of the formal and actual parameters are identical.

How the variable is declared in a subprogram determines what the subprogram can do with the variable and whether the variable is properly protected against write access.

- If the subprogram declares the variable as a DICTIONARY RECORD variable, the DICTIONARY RECORD variable can be accessed through the described fields.
- If the subprogram declares the variable as another type of record variable, the variable can be accessed through the field names of the record. The semantic rules for that type of record variable are enforced.
- If the subprogram declares the variable as an EBCDIC array, no field-oriented access can be used. Assignment to the variable is permitted.

Refer to the *Binder Programming Reference Manual* for more information. Additional information related to ADDS items used as parameters is included under "Passing Entities as Parameters" in this section.

Statements Used as ADDS Extensions

The assignment, REPLACE, and SCAN statements can be used with ADDS entities. The assignment statement syntax is shown below. Consult Volume 1 for the syntax of REPLACE and SCAN statements. Additional information related to ADDS and pointers is included under “POINTER Function” in this section.

Assignment Statement

<arithmetic assignment statement>

```
─┬─ <display ID> ─────────────────┬─ := ─ <arithmetic expression> ─────┤
 ├─ <qualified display field ID> ─┤
 ├─ <digits ID> ──────────────────┤
 ├─ <qualified digits field ID> ──┤
 ├─ <binary ID> ──────────────────┤
 ├─ <qualified binary field ID> ──┤
 ├─ <real ID> ────────────────────┤
 ├─ <qualified real field ID> ────┤
 ├─ <qualified integer field ID> ─┤
 ├─ <double ID> ──────────────────┤
 └─ <qualified double ID> ────────┘
```

<Boolean assignment statement>

```
─┬─ <Boolean ID> ─────────────────┬─ := ─ <Boolean expression> ────────┤
 └─ <qualified Boolean field ID> ─┘
```

<record assignment statement>

```
─ <record> ─ := ─ <record> ─────────────────────────────────────────────┤
```

Explanation

The assignment statement causes the item on the right of the assignment operator (:=) to be evaluated and the resulting value to be assigned to the item on the left of the assignment operator.

Three types of assignment statements can be used: arithmetic, Boolean, and record. Refer to Volume 1 for a discussion of the assignment statement, specifically arithmetic and Boolean assignment statements.

If the arithmetic value to be assigned into a field or item of type Display or Digits does not fit, the value is rounded and/or the high-order characters are truncated. In effect, a MOD operation for remainder division is performed (as described in Volume 1). If the RANGECHECK compiler control option is set, a run-time fault will be generated if any characters are truncated.

In assignments between Display fields or items, or between Digit fields or items, the resulting value is converted into a 48- or 96- bit operand and then back into characters. Blank fill is performed on unneeded character spaces.

The assignment of a Boolean value to a field of type Boolean affects all four bits of the field.

If the arithmetic value to be assigned into a type Integer is too large, then the high-order bits are truncated. In effect, a MOD operation is performed. If the RANGECHECK compiler control option is set, a run-time fault will be generated if any bits are truncated.

Records can only be assigned to records that share the same dictionary entity type description. Two records described by disjoint type descriptions but that are logically identical will not be compatible for the purposes of assignment.

Additional information relating to assignment statements is included under “RANGECHECK Option: Checking Ranges of Run-time Values,” “Referencing Fields and Records,” “Entity Qualifiers,” and “Relating ADDS Data Types to ALGOL” in this section.

Examples

In the following example of an arithmetic assignment, the Integer field MONTH is embedded in record YEAR:

```
YEAR.MONTH := 10;
```

REPLACE and SCAN Statements

The REPLACE statement, as described in Volume 1, causes character data from one or more sources to be stored in a designated portion of an array row.

The SCAN statement, as described in Volume 1, examines a contiguous portion of character data in an array row, one character at a time, in a left-to-right direction. The source is always a pointer expression.

For both statements, fields and items of type EBCDIC array are considered to be pointer expressions.

Functions Used as ADDS Extensions

The ALGOL functions LENGTH, OFFSET, POINTER, RESIZE, SIZE, and STACK option are extended for use with ADDS. ADDS also provides the UNITS function. Record, field, display, and digit identifiers are valid input for all these functions.

- LENGTH function

  The LENGTH function returns the length of a specified entity in the designated units.

- OFFSET function

  The OFFSET function returns the number of units that the specified entity is indexed from the beginning of the outermost record in which it is declared.

- POINTER function

  The POINTER function returns a pointer to the specified input.

- RESIZE function

  The RESIZE function changes the size of the array underlying a given record identifier.

- SIZE function

  The SIZE function returns the size of the array underlying a given record identifier.

- STACK option

  The STACK compiler control option directs the ALGOL compiler to print data definition descriptions that are imported from SDF Plus during compilation.

- UNITS function

  The UNITS function accepts an entity as input and returns, as an integer value, the default unit size expected by the LENGTH and OFFSET functions.

LENGTH Function

<length function>

```
– LENGTH – ( ─┬─ <record ID> ──────────┬─┬──────────────────────┬─ ) ─────┤
              ├─ <qualified field ID> ─┤ └─ , – <units value> ─┘
              ├─ <display ID> ─────────┤
              └─ <digit ID> ───────────┘
```

Explanation

The extended LENGTH function returns, as an integer value, the length of the specified entity in designated units.

The length of a record, field, display, or digit can be returned. If a field is specified, the field must be qualified.

The valid values for units are 1, 4, 8, and 0 (zero). If a value is not specified, a default is used. See “UNITS Function” for a list of defaults.

An error results if the length of the entity cannot be expressed as an integral number of units. For example, the length of a 3-character EBCDIC array field cannot be expressed in words.

Additional information related to the LENGTH function is included under “Referencing Fields and Records,” “Relating ADDS Data Types to ALGOL,” and “UNITS Function” in this section.

Examples

Shown below, A is assigned the Boolean field's length of 1. The field Booleanfield is qualified by the record R.

```
A := LENGTH (R.Booleanfield);     %A = 1
```

In the following example, A is assigned the record's length of R. The default unit size is bits.

```
A := LENGTH (R);                  %A = number bits in R
```

In this example, A is assigned the record's length of R. The default unit size is bits, but digits are specified.

```
A := LENGTH (R,4);                %A = number digits in R
```

OFFSET Function

<offset function>

```
─ OFFSET ─ ( ─┬─ <record ID> ───────────┬─┬───────────────────────┬─ ) ────┤
              │                         │ │                       │
              ├─ <qualified field ID> ──┤ └─ , ─ <units value> ───┘
              │                         │
              ├─ <display ID> ──────────┤
              │                         │
              └─ <digit ID> ────────────┘
```

Explanation

The OFFSET function returns, as an integer value, the number of units that the designated entity is indexed from the beginning of the outermost record in which the entity is declared.

The valid values for units are 1, 4, 8, and 0 (zero). If no value is specified, a default is used. See “UNITS Function” for a list of defaults.

An error results if the offset of the field, record, display, or digit cannot be expressed in an integral number of units or if the offset can only be determined at run time and might not be expressible as an integral number of units. This can occur when units larger than the default unit are specified or when a field is an element in an array of fields.

Additional information related to the syntax of the OFFSET function is included under “Referencing Fields and Records,” “Relating ADDS Data Types to ALGOL,” and “UNITS Function” in this section.

Examples

In the following example, A is first assigned the offset of field X in record R. The units are returned in digits. A is then assigned the offset of field Y. The units are returned in bytes.

```
A := OFFSET (R.X,4) %A = the offset of X in R in digits
A := OFFSET (R.Y,8) %A = the offset of Y in R in bytes
```

In this example, A is assigned the offset of T from the beginning of R, in digits:

```
A := OFFSET (R.S.T,4)
```

In the next example, A is assigned an offset in bits to be determined at run time. The assignment is permitted because the offset is known to be expressible in bits.

```
A := OFFSET (R.Q[N]);
```

POINTER Function

<pointer function>

```
─ POINTER ─ ( ─┬─ <qualified field ID> ─┬─┬───────────────────────┬─ ) ──┤
               ├─ <record ID> ──────────┤ └─ , ─ <character size> ─┘
               ├─ <display ID> ─────────┤
               └─ <digit ID> ───────────┘
```

Explanation

The POINTER function returns a pointer to the designated input. Records, while implemented as EBCDIC arrays, cannot be referenced as such without the explicit use of the POINTER function.

The pointer acts as if it were pointing to data of the specified character size. If the character size is not specified, and the designated field holds 4-bit characters, a character size of 4 is assumed. In all other cases the default character size is 8.

The POINTER function bypasses all compiler restrictions related to field integrity and type. A record can thus be referenced as a one-dimensional array.

Additional information related to the syntax of the POINTER function is included under “Referencing Fields and Records,” and “Guidelines for Using ADDS Types” in this section.

Examples

In the following example, R is an EBCDIC field which is filled with spaces.

```
REPLACE POINTER F.R BY " " FOR LENGTH (F.R)
```

Below, the quoted string “ABCDEF” is used to fill the entire length of R. The string is repeated as many times as necessary to fill the entire length.

```
REPLACE POINTER (R,8) BY "ABCDEF" FOR LENGTH (R,8);
```

RESIZE Function

<resize function>

```
— RESIZE — ( — <record ID> — , — <new size> ─┬────────────────────┬─ ) ───┤
                                             │                    │
                                             └ , ─┬─ RETAIN ──────┘
                                                  ├─ DISCARD ─┤
                                                  └─ PAGED ───┘
```

Explanation

The RESIZE function changes the size of the array underlying a given record identifier. This function can also change the size of the array containing a record by changing the upper bound of the array. The size of the *entire* array is changed, regardless of the record's position in the array.

The record ID construct is the identifier of any valid record within the ALGOL program.

The new size construct is an integer that represents the number of elements in the array after the RESIZE function is performed. The size of each element depends on the type of the underlying array. The element sizes of some common record arrays are shown in the following table.

Record	Element Size
Advanced Data Dictionary System (ADDS) records	Bytes
Communication Management System (COMS) Input Headers, Output Headers, or COMS records	Words
Screen Design Facility Plus (SDF Plus) form record libraries	Bytes
Semantic Information Manager (SIM) records	Bytes

More detailed information on the RESIZE function is included in Volume 1, under “RESIZE Statement” in Section 5, “Statements”. Additional related information is included under “Referencing Fields and Records,” and “Relating ADDS Data Types to ALGOL” in this section.

Examples

In the following example, the array containing the record INPUTRECORD is changed to the value of NEWSZ and the previous contents of the array are discarded.

```
RESIZE(INPUTRECORD, NEWSZ, DISCARD)
```

In the following example, the size of the array containing INPUTRECORD is changed to be the same as the value of the MAXRECSIZE attribute of the file INPUTFILE. The previous contents of the array are retained.

```
RESIZE(INPUTRECORD, INPUTFILE.MAXRECSIZE, RETAIN)
```

In this example, the size of the array containing INPUTRECORD is increased by 100 elements. The previous contents of the array are retained, but the array is changed to be a paged (segmented) array.

```
RESIZE(INPUTRECORD, SIZE(INPUTRECORD)+100, PAGED)
```

SIZE Function

<size function>

```
 — SIZE — ( — <record ID> — ) ——————————————————————————————————————————————————┤
```

Explanation

The SIZE function returns the size of the array underlying a given record identifier.

The SIZE function accepts a record identifier and returns the number of elements in the array that contains the record. This function returns an integer representing the size of the *entire* array, regardless of the record's position in the array.

The size of each element depends on the type of the underlying array. The element sizes of some common record arrays are shown in the following table.

Record	Element Size
Advanced Data Dictionary System (ADDS) records	Bytes
Communication Management System (COMS) Input Headers, Output Headers, or COMS records	Words
Screen Design Facility Plus (SDF Plus) form record libraries	Bytes
Semantic Information Manager (SIM) records	Bytes

More detailed information on the SIZE function is included in Volume 1, under “Intrinsic Function Descriptions” in Section 6, “Expressions”. Additional related information is included under “Referencing Fields and Records,” and “Relating ADDS Data Types to ALGOL” in this section.

Examples

In the following example, ARRYLIMIT is assigned the size of the array that contains the record INPUTRECORD.

```
ARRYLIMIT := SIZE(INPUTRECORD)
```

STACK Option

<stack option>

```
──┬─ STACK ─┬──────────────────────────────────────────────────────┤
  └─ MAP ───┘
```

Explanation

The STACK option directs the ALGOL compiler to print data definition descriptions that are imported from SDF Plus during compilation.

When the stack option is set, the ALGOL compiler includes the definitions of the imported form record field types, form record types, form record library types, and file types in the compilation.

(Type: Boolean, Default value: FALSE)

MAP is a synonym for the STACK option.

UNITS Function

<units function>

```
– UNITS – ( ─┬─ <record ID> ──────────┬─ ) ────────────────────────────────┤
             ├─ <qualified field ID> ─┤
             ├─ <display ID> ─────────┤
             └─ <digit ID> ───────────┘
```

Explanation

The UNITS function accepts a specified entity as input and returns, as an integer value, the default unit size expected by the LENGTH and OFFSET functions.

The default unit size is the lowest common unit of the target type in which the length and offset of the target can be expressed. In general, if a target contains 4-bit or 8-bit character data, the value returned is 4 or 8, respectively. Otherwise, the value returned is 1.

The following table shows how the unit sizes are interpreted:

Unit	Meaning
1	Bits
4	Digits
8	Bytes
0	Words

Default unit sizes for ADDS fields and records are shown in the table below:

Field or Record	Default Unit Size
Display fields	8
EBCDIC Array fields	8
Digits fields	4
Binary fields	1
Boolean fields	1
Double fields	1
Entity Reference fields	1
Integer fields	1
Real fields	1
Record fields	1
Records	1

Note that, by definition

```
LENGTH(R.X) = LENGTH (R.X,UNITS(R.X))
```

Additional information related to the syntax of the UNITS function is included under "Referencing Fields and Records," "Relating ADDS Data Types to ALGOL," "LENGTH Function," and "OFFSET Function."

Example

The default unit size of field X in record R is returned.

```
A := UNITS (R.X)
```

Section 3
Using Communications Management System (COMS) Features

The Communications Management System (COMS) is a message control system (MCS) developed to control interactive environments. COMS supports the processing of multiple program transactions as well as single-station and multiple-station remote files.

The ALGOL interface to COMS enables programs to communicate through COMS with terminals or other programs. ALGOL programs interact with COMS through the COMS direct-window interface. The following features and functions are available to the programs:

- Message routing by transaction codes (trancodes) and agendas
- Security checking of messages that programs receive and send
- Service functions for manipulating COMS entities by translating COMS values to names and translating names to COMS values
- Dynamic opening of direct windows to terminals not attached to COMS, and dynamic communication over a modem
- Synchronized recovery for multiple database processing programs running asynchronously
- External definition of record formats related to COMS (COMSRECORD declarations)

For COMS to perform these functions, the required version of COMS must be installed and the ALGOL program must link to a COMS library and declare an input header, an output header, and a message area.

COMS can be used with Advanced Data Dictionary System (ADDS), Data Management System II (DMSII), and Semantic Information Manager (SIM). This section provides a brief overview of the ALGOL functions that can be used with COMS and details the statements that can be used for synchronized recovery with DMSII. Refer to Section 7, "Using the Semantic Information Manager (SIM) Interface," in this volume and to the *InfoExec Semantic Information Manager (SIM) Programming Guide* for information on synchronized recovery with SIM.

Refer to the *Communications Management System (COMS) Programming Guide* for a discussion of COMS programming issues and a detailed explanation of the COMS features and functions available with each version of COMS.

The COMS interface has created the following new ALGOL type 2 reserved words:

AFTER	INPUTHEADER	OUTPUTHEADER
BEFORE	MESSAGECOUNT	RECEIVE
EGI	NOCR	SEND
EMI	NOLF	TERMINAL
ESI		

Additional information relating to COMS and SIM is included in Section 7, "Using the Semantic Information Manager (SIM) Interface."

Using ALGOL Functions as COMS Extensions

The RANGECHECK compiler control option, as well as the LENGTH, OFFSET, POINTER, and UNITS functions, can be used as COMS extensions. More detailed information about these ALGOL functions is included in Section 2, "Using Advanced Data Dictionary System (ADDS) Extensions."

Purpose of the RANGECHECK Option

The RANGECHECK option is a Boolean option that causes the compiler to generate code that performs range checking at run time on values that were not known at compile time. The option is set by default. A run-time fault occurs if a value fails a range check; the program is discontinued and an "Invalid Operation" is reported.

Purpose of Functions

The following ALGOL functions can be used with COMS:

- LENGTH function

  The LENGTH function returns the length of a specified entity in the designated units.

- OFFSET function

  The OFFSET function returns the number of units that the specified entity is offest from the beginning of the outermost record in which it is declared.

- POINTER function

  The POINTER function returns a pointer to the specified input.

- RESIZE function

  The RESIZE function changes the size of the array underlying a given record identifier. For COMS input and output headers and COMS records, the size is given in words. The size of the *entire* array is changed, regardless of the record's position in the array.

- SIZE function

  The SIZE function returns the size of the array underlying a given record identifier. For COMS input and output headers and COMS records, the size is given in words. The size returned is an integer representing the size of the *entire* array, regardless of the record's position in the array.

- UNITS function

  The UNITS function accepts an entity as input and returns, as an integer value, the default unit size expected by the LENGTH and OFFSET functions.

Linking to COMS

An ALGOL program accesses a COMS library by declaring a record format related to COMS (for example, INPUTHEADER, OUTPUTHEADER, or COMSRECORD).

The library linkage is implicitly declared to the COMS library entry point entitled DCIENTRYPOINT. A link to the library entry point is established when the first COMS statement is encountered at run time. It is preferable to establish a link to the library entry point rather than declaring the COMS DCILIBRARY and calling the entry point explicitly.

The default library access is BYFUNCTION with a FUNCTIONNAME of COMSSUPPORT. A LIBPARAMETER is generated by the compiler. The title, function name, and library access attributes of the COMSSUPPORT library can be changed in the same way as any other declared library, by using the internal name of COMSSUPPORT. If these changes are made, they must be made prior to the first executable statement in the program.

For further information on libraries and library declarations, consult Volume 1. Refer to the *System Software Utilities Operations Reference Manual* for details on library attributes, and the *System Commands Operations Reference Manual* for a description of the SL (Support Library) command.

Additional information relating to COMS libraries is included under “COMS Statements,” “COMS Service Functions,” “Declaring an Input or Output Header,” and “Declaring a COMSRECORD” in this section.

Linking to COMS by Title

It is possible for an ALGOL program to link to COMS by title. The following is an example of the statements that must be included at the beginning of the program.

Example

```
         COMSSUPPORT.LIBACCESS := VALUE (BYTITLE);
          REPLACE SCRATCH BY MYSELF.EXCEPTIONTASK.EXCEPTIONTASK.NAME;
          COMSSUPPORT.TITLE := STRING(SCRATCH[0], 256);
    % Store the family name so it can be temporarily changed
         REPLACE SCRATCH BY MYSELF.FAMILY;
    % Reset family name to null in case running on disk named DISK
         REPLACE MYSELF.FAMILY BY ".";
          ENABLE (<inputheadername>, "ONLINE");
    % Restore family name for accessing files, etc.
         REPLACE MYSELF.FAMILY BY SCRATCH;
```

Note: *It is recommended that you do not link to COMS by title. Linking by function is the recommended method.*

Declaring an Input or Output Header

A header is a record structure with predefined field names and purposes. Consult the *COMS Programming Guide* for information on the structure of the header.

An input or output header declaration associates a header identifier with a header. It declares a header record as a variable. An input or output header can also be declared by using TYPE declarations and invocations. You can use the TYPE declaration to declare a header record as a type identifier.

You can also declare input and output header formats by using the COMSRECORD declaration. The COMSRECORD declaration is the best method to use when declaring header formats. Refer to “Declaring a COMSRECORD” later in this section for more information about the COMSRECORD declaration.

Input and output headers are used when a program communicates with COMS through a direct-window interface. Each header is one record and is composed of multiple fields. The fields contain routing or descriptive information for the actual message.

Although the message is not part of the header, it is associated with the header for routing when it is named as the message area variable in a RECEIVE or SEND statement.

Both input and output headers can have an optional conversation area field at the end of the structure. The conversation area field is the only user-defined field in an input or output header. Consult the *COMS Programming Guide* for a definition of the contents in the conversation area field of an input or output header.

The conversation area field is accessed in the same manner as the predefined fields. If a header has a conversation area field and the header is passed as a parameter, a TYPE declaration is required.

Input and output headers can be bound to other input or output headers. The headers must have the same conversation area description.

Because the layout of input or output headers can change with each software release, a program must not preserve any designators across executions. Designators must not be used as key data in a database. To guarantee the validity of the data, save all necessary information in the appropriate header every time the header is used.

Additional information relating to input and output headers is included under “Input or Output Header Type Declaration” in this section.

Input or Output Header Declaration

<header declaration>

```
──┬─ INPUTHEADER  ─ <inputheadername> ───┬┬──────────────────────────┬────────→
  └─ OUTPUTHEADER  ─ <outputheadername> ─┘└─ <conversation area> ─┘

→─┬────────────────────┬──────────────────────────────────────────────────────┤
  └─ <addr equation> ─┘
```

<inputheadername>

```
 ─ <identifier> ──────────────────────────────────────────────────────────────┤
```

<outputheadername>

```
 ─ <identifier> ──────────────────────────────────────────────────────────────┤
```

<conversation area>

```
─ ( ─┬─ <Boolean declaration> ──────────┬─ ) ─────────────────────────────────┤
     ├─ <integer declaration> ──────────┤
     ├─ <real declaration> ─────────────┤
     └─ <conversation array declaration> ─┘
```

<conversation array declaration>

```
─┬───────────┬─ ARRAY  ─ <identifier> ─ [─ <bound pair> ─ ] ──────────────────┤
 ├─ Real ────┤
 ├─ Integer ─┤
 └─ Boolean ─┘
```

<addr equation>

```
─ = ─┬─ <ADDS record ID> ──┬──────────────────────────────────────────────────┤
     ├─ <inputheader ID> ──┤
     ├─ <outputheader ID> ─┤
     ├─ <DMRECORD ID> ─────┤
     ├─ <real array ID> ───┤
     └─ <EBCDIC array ID> ─┘
```

Explanation

The inputheadername construct identifies an input header used to receive messages through COMS. The outputheadername construct identifies an output header used to send messages through COMS. A program can have one or more input or output headers.

The conversation area declaration is optional. However, if a header has a conversation area, this declaration defines the type and length of the conversation area field.

The addr equation construct is optional. This construct is similar to the array row equivalence construct of an array declaration in that it causes the declared input or output header to refer to the same data as the specified record or array row.

Examples

The following example declares the input header RECEIVECOMS. It has no conversation area field.

```
INPUTHEADER RECEIVECOMS;
```

In the following example, the input header MYINPUT is declared as having a one- word Real conversation area identified as MYAREA:

```
INPUTHEADER MYINPUT (REAL MYAREA);
```

The conversation area field is declared as a REAL array in the following example:

```
OUTPUTHEADER SENDCOMS (REAL ARRAY CONVERSATION[0:90]);
```

Input or Output Header Type Declaration

<header type declaration>

```
            ┌←──────────────────────────── , ─────────────────────────────┐
── TYPE ────┴─┬─ INPUTHEADER ──┬─ <header type ID> ─┬──────────────────────┬┴──┤
              └─ OUTPUTHEADER ─┘                    └─ <conversation area> ─┘
```

<header type invocation>

```
                       ┌←────────── , ──────────┐
── <header type ID> ───┴─┬─ <inputheadername> ──┬┴──────────────────────────┤
                         └─ <outputheadername> ─┘
```

<header type ID>

```
── <identifier> ────────────────────────────────────────────────────────────┤
```

Explanation

The TYPE declaration can be used to associate a user-defined name with a header format specified in an input or output header declaration. The format can then be used as a data description. The TYPE declaration is required if a header has a conversation area field and the header is passed as a parameter.

Normally, declaring an input or output header creates a structure as a variable. In contrast, the TYPE declaration does not create a variable; it simply defines a type identifier that can be used to declare record variables. A type identifier is associated with an input or output header declaration. In effect, the type identifier is the name of a record structure description.

Only variables that share the same entity description and type are compatible. The TYPE declaration provides compatibility for the headers. Records described by separate, distinct entities and identical in content are compatible only if they share the same type identifier.

A TYPE declaration must precede a type invocation. The type invocation declares records that have the structure associated with the type identifier.

Additional information on the inputheadername, outputheadername, and conversation area constructs is included under “Declaring an Input or Output Header” in this section. Related information is also included under “Accessing Header Fields” in this section, and under “Referencing Fields and Records” in Section 2, “Using the Advanced Data Dictionary System (ADDS) Extensions.”

The header type identifier is the user-defined name associated with the format. The header type ID construct includes the name of the input or output header, as declared in the TYPE declaration. Each record specified by an inputheadername or outputheadername construct in the type invocation has the structure defined by the header type identifier.

Examples

In this example, a TYPE declaration creates a data definition from the input header MYINPUTHEADER. The type invocation is then used to impose the structure onto the records NEXTHEADER and PREVHEADER.

```
TYPE INPUTHEADER MYINPUTHEADER; MYINPUTHEADER NEXTHEADER, PREVHEADER;
```

The following example creates a data definition from the output header OUTMSG. The definition includes the conversation area CONAREA. The structure is then imposed on the record ROUTE.

```
TYPE OUTPUTHEADER OUTMSG (REAL CONAREA); OUTMSG ROUTE;
```

Input Header Structure and Type

Table 3–1 shows the predefined fields of the input header that are available to an ALGOL program. The fields are listed as they appear in the structure, including the optional conversation area field. The listing gives the ALGOL name, data type, and a brief description for each field.

COMS places values (designators and integers) in the input header fields when an ENABLE, MESSAGECOUNT, or RECEIVE statement is executed. You can use a service function to translate a designator to a name representing a COMS entity.

Input headers are used in receiving messages. For messages that are received, the input header fields are used for the following tasks:

- Confirming message status
- Passing data in the conversation area field
- Detecting queued messages
- Determining message origin
- Obtaining direct-window notifications
- Processing transaction codes (trancodes) for routing

The fields, their COMS names and values, and their purposes are detailed in the *COMS Programming Guide.*

Additional information relating to the fields of a COMS input header is included under “COMS Service Functions” in this section.

Table 3–1. Input Header Structure and Type

Field Name	Data Type	Brief Description
PROGRAMDESG	Designator	Designator that COMS has assigned to the program or designator of the program that sent the message.
FUNCTIONINDEX	Integer	Module Function Index (MFI) that can be used in conjunction with COMS trancode-based routing.
FUNCTIONSTATUS	Integer	Positive value: COMS-defined error value. Negative value: Reports the status of a dynamic attachment, a confirmation request for output messages, or a COMS notification to a direct window.
USERCODE	Designator	Designator for the usercode associated with the program or station originating the message.
SECURITYDESG	Designator	Designator that can be used for security checking.

Table 3–1. Input Header Structure and Type

Field Name	Data Type	Brief Description
FIELDS.VTFLAG	Boolean	Virtual terminal (VT) flag returned by COMS.
FIELDS.TRANSPARENT	Boolean	Designator that shows whether the input message is being passed in transparent mode.
TIMESTAMP	Real	Time and date message is first encountered by COMS.
STATION	Designator	Terminal number for the terminal being dynamically attached or detached, or the station originating the message.
TEXTLENGTH	Integer	Number of characters in the text of incoming message, length of destination telephone number, length of delivery confirmation, or notification of a direct window on/open activity.
STATUSVALUE	Integer	Status of an input message.
MESSAGECOUNT	Integer	Number of messages queued to the program.
RESTART	Designator	Last message that COMS audited in the DMSII transaction trail.
AGENDA	Designator	Designator of the most recently applied input agenda.
SDFINFO	Real	Designator that identifies errors that occurred during the processing of a form message. See “Using COMS Input/Output Headers” in Section 6, “Using the Screen Design Facility Plus (SDF Plus) Interface,” for more information about the values of this field.
SDFFORMRECNUM	Real	Designator that identifies the form record that is received.
SDFTRANSNUM	Real	The number of the SDF Plus transaction that is received. This field must not be altered by the user application.
CONTDATASTATUS	Integer	Status of the continuator data in the input header conversation area.

Table 3–1. Input Header Structure and Type

Field Name	Data Type	Brief Description
CONTDATAOFFSET	Integer	Starting position (in bytes) of the continuator data in the input header conversation area (zero-relative to the starting position of the conversation area).
CONTDATALENGTH	Integer	Length (in bytes) of the continuator data in the input header conversation area.
CONTENTRYNUM	Integer	Indicates the phase of a transaction cycle.
Conversation Area	User-defined	Information passed by program, processing item, or telephone number for a direct-window interface.

Additional information relating to the fields of a COMS input header is included under “Using COMS Input/Output Headers” in Section 6, “Using the Screen Design Facility Plus (SDF Plus) Interface.”

Output Header Structure and Type

Table 3–2 shows the predefined fields of the output header that are available to an ALGOL program. The fields are listed as they appear in the structure, including the optional conversation area field. The listing gives the ALGOL name, data type, and a brief description of the fields.

The output header is used in sending messages. You can place designators into the fields to route outgoing messages and describe their characteristics. You can also obtain designators by calling service functions to translate names representing COMS entities to designators.

For messages that are output, the header fields are used in

- Specifying a destination
- Routing by transaction code (trancode)
- Sending messages using direct windows
- Confirming message delivery
- Checking the status of output messages

The fields, their COMS names and values, and their purposes are detailed in the *COMS Programming Guide*.

Additional information relating to the fields of a COMS output header is included under "COMS Service Functions" in this section.

Table 3–2. Output Header Structure and Type

Field Name	Data Type	Brief Description
DESTCOUNT	Integer	Number of destinations to which the program sends the message.
TEXTLENGTH	Integer	Number of characters contained in the text of an outgoing message.
STATUSVALUE	Integer	Notes whether the message was successfully sent to its destination or if an error occurred.
FIELDS.VTFLAG	Boolean	Virtual terminal (VT) flag set by direct-window program.
FIELDS.CONFIRMFLAG	Boolean	Requests delivery confirmation of an output message.
FIELDS.CONFIRMKEY	EBCDIC array [0:2]	User-defined tag for delivery confirmation of an output message.
FIELDS.TRANSPARENT	Boolean	Used to specify transparent mode for an output message.

Table 3–2. Output Header Structure and Type

Field Name	Data Type	Brief Description
DESTINATIONDESG	Designator	Destination for a message.
NEXTINPUTAGENDA	Designator	Agenda to be applied to the next input for the current dialogue.
TOGGLES.SETNEXT-INPUTAGENDA†	Boolean	Used to specify whether COMS is to use the contents of the NEXTINPUTAGENDA field to change the agenda for the next input to the current dialogue of the destination station.
TOGGLES.RETAIN-TRANSACTIONMODE†	Boolean	Specifies whether or not Transaction Mode is to be retained for the current dialogue.
AGENDA	Designator	Specifies an agenda for postprocessing of the message a program is sending.
SDFFORMRECNUM	Real	Designates the form record to be written.
CONTMODE	Integer	Tells COMS how it should handle continuator data for the destination station.
CONTDATASTATUS	Integer	Indicates if the continuator data in the output header conversation area was successfully stored.
CONTDATAOFFSET	Integer	Starting position (in bytes) of the continuator data in the output header conversation area (zero-relative to the starting position of the conversation area).
CONTDATALENGTH	Integer	Length (in bytes) of the continuator data in the output header conversation area.
CONTENTRYNUM	Integer	Indicates the phase of a transaction cycle.
MAPALIAS	EBCDIC array [0:5]	Specifies a Map Alias name for post-process mapping of the message sent by a program. If the name length is less than six characters, pad it with spaces on the right.
Conversation Area	User-defined	Passes information, in addition to the message data, to processing items.

† Do not include the hyphen in the name.

Additional information relating to the fields of a COMS output header is included under “Using COMS Input/Output Headers” in Section 6, “Using the Screen Design Facility Plus (SDF Plus) Interface.”

Designator Data Type

The data type Designator is used only for specific fields of the COMS headers and with COMS service functions. It is an internal code understood by COMS and used to control messages symbolically in the data communications environment. COMS can determine the kind of entity represented by a particular designator such as a station or usercode.

In ALGOL, the data type Designator is acted upon as if it were the data type Real. The compiler does not differentiate between the two types. However, COMS operations require that no arithmetic operations be performed on a field of type Designator. The Designator type can be altered within a program only if some type of operation is done by a COMS service function that decodes or returns a value for the designators. You can return designators to their initial values by setting them to 0 (zero).

Additional information relating to the Designator data type is included under “COMS Service Functions” in this section.

Declaring a Message Area

The message area is the variable reserved for the actual message. You must declare a message area variable before you can send or receive a message. (The program builds messages in the message area.) Once information is returned from COMS in the message area, the program determines any further processing.

The variable can be an EBCDIC array or an ADDS record, including SDF Plus form record libraries stored in ADDS. If the variable is not large enough to contain all the text of the message, COMS truncates the message. The TEXTLENGTH field of the header is used to report the length of the valid text in the message area.

Refer to the *COMS Programming Guide* for details of how COMS uses and interprets the message area and for information on the fields of the headers.

Additional information on the message area is included under “COMS BEGINTRANSACTION Statement,” “COMS ENDTRANSACTION Statement,” “RECEIVE Statement,” and “SEND Statement,” in this section. Related information is also included under “Using SDF Plus with COMS” in Section 6, “Using the Screen Design Facility Plus (SDF Plus) Interface.”

Declaring a COMSRECORD

<COMSRECORD declaration>

```
── COMSRECORD ──────────────────────────────────────────────────────────────→

  ┌←─────────────────────── , ──────────────────────────────┐
→─┴─ <format type> ── <record id> ─┬──────────────────────┬┬────────────────┬┴──────────┤
                                   └─ <conversation area> ┘└─ <addr equation> ┘
```

Explanation

The COMSRECORD declaration is a way to obtain the declarations for COMS record formats from an external system library, instead of from information contained in the ALGOL compiler.

When the ALGOL compiler encounters a COMSRECORD declaration, it extracts a character string (called a format type) from the declaration. The character string is passed to the COMSLANGSUPPORT external system library. The library checks the character string against an internal list of COMS record formats.

- If the character string *is* a valid format type, the COMSLANGSUPPORT library returns a description of the format to the compiler. This description contains the explicit declarations and definitions for the desired record format (including the names, types, and locations of the fields in the record).
- If the character string *is not* a valid format type, the COMSLANGSUPPORT library returns an error condition to the compiler. The compiler generates a syntax error.

The ALGOL compiler has no information about the format types or record formats. It simply passes the format type to the COMSLANGSUPPORT library and receives either the record format definitions or the error condition.

The keyword COMSRECORD causes the compiler to request the desired record format from the COMSLANGSUPPORT external system library.

The format type construct is the identifier of a character string. The character string can be a maximum of 64 characters in length.

The following list identifies the three format types.

- INPUTHEADER

  This format type represents the normal COMS input header record format described earlier in this section. The results of a COMSRECORD declaration with a format type of INPUTHEADER are identical to explicitly declaring a COMS input header in your application program.

  Refer to “Declaring an Input or Output Header” for the structure, field names, and field types of a COMS input header record in this section. Consult the *COMS Programming Guide* for information about the use and meaning of the input header fields.

- OUTPUTHEADER

  This format type represents the normal COMS output header record format described earlier in this section. The results of a COMSRECORD declaration with a format type of OUTPUTHEADER are identical to explicitly declaring a COMS output header in your application program.

  Refer to “Declaring an Input or Output Header” in this section for the structure, field names, and field types of a COMS output header record. Consult the *COMS Programming Guide* for information about the use and meaning of the output header fields.

- X25

  This format type represents the record format used with the X.25 MCS product.

  Refer to “COMSRECORD Structures and Types” later in this section for the structure, field names, and field types of a COMSRECORD in the X.25 format. Consult the *X.25 MCS Operations and Programming Reference Manual* for information about the use and meaning of the individual fields in the record.

The record id construct identifies the individual COMSRECORD.

The conversation area construct is optional. If a COMSRECORD has a conversation area, this declaration defines the type and length of the conversation area field. The syntax used to declare a conversation area is described under “Input or Output Header Declaration” in this section.

The addr equation construct is optional. This construct is similar to the array row equivalence construct of an array declaration in that it causes the declared COMSRECORD to refer to the same data as the specified record or array row.

Type Declaration of a COMSRECORD

<COMSRECORD type declaration>

```
── TYPE ───────────────────────────────────────────────────────────────→

   ┌←─────────────────────────────── , ─────────────────────────────┐
→──┴─ COMSRECORD ── <format type> ─┬─────────────────────┬─ <type id> ─┴──┤
                                   └─<conversation area>─┘
```

Explanation

A COMSRECORD type declaration associates a user-defined name (called a type id) with a specific COMSRECORD format. After a COMSRECORD type is declared, the user-defined name can be used as a data description. COMSRECORD type declarations are used in the same way as type declarations for normal COMS input and output headers. Refer to “Input or Output Header Type Declaration” earlier in this section for more information.

The format type construct is the identifier of a character string. The character string can be a maximum of 64 characters in length. The three valid format types include INPUTHEADER, OUTPUTHEADER, and X25.

The conversation area construct is optional. If a COMSRECORD has a conversation area, this declaration defines the type and length of the conversation area field. The syntax used to declare a conversation area is described under “Input or Output Header Declaration” in this section.

The type id construct is a user-defined name that is associated with the specific COMSRECORD format.

Type Invocation of a COMSRECORD

<COMSRECORD type invocation>

```
                ┌←────── , ─────┐
– <type id> ──┴─ <record id> ─┴──────────────────────────────────────┤
```

Explanation

A COMSRECORD type invocation must follow a COMSRECORD type declaration. The type invocation declares a COMSRECORD that has the format associated with the type id.

The type id construct is a user-defined name that is associated with the specific COMSRECORD format.

The record id construct identifies the individual COMSRECORD.

COMSRECORD Structures and Types

The following list describes the three valid COMSRECORD formats.

- INPUTHEADER

  A COMSRECORD with a format type of INPUTHEADER has the same structure and type as the normal COMS input header record described earlier in this section. Refer to "Declaring an Input or Output Header" in this section for the structure, field names, and field types of a COMS input header record. Consult the *COMS Programming Guide* for information about the use and meaning of the input header fields.

- OUTPUTHEADER

  A COMSRECORD with a format type of OUTPUTHEADER has the same structure and type as the normal COMS output header record described earlier in this section. Refer to "Declaring an Input or Output Header" in this section for the structure, field names, and field types of a COMS output header record. Consult the *COMS Programming Guide* for information about the use and meaning of the output header fields.

- X25

  The structure and type of a COMSRECORD with a format type of X25 is described in the following pages.

Structure and Type of an X.25 COMSRECORD

Table 3–3 describes the predefined fields for a COMSRECORD, in an X.25 format, that are available in an ALGOL program. The fields are listed as they appear in the structure. The listing displays the ALGOL name, data type, and description of each field.

Table 3–3. X.25 COMSRECORD Structure and Type

Field Name	Data Type	Brief Description
CLASS	Integer	The CLASS field describes the class or type of the record. The record must contain a CLASS field. The initial value for this field is X25. X25 is the only possible value for use with the X.25 MCS.
VERSION	Integer	The VERSION field contains the record version number. All records must contain a VERSION field. If changes occur in the future to the structure of a record, this field will be incremented. The initial value for this field is X25PIRVERSION.
FUNCTION	Integer	The FUNCTION field contains a description of the packet type of the record. All records must contain a FUNCTION field.

Table 3–3. X.25 COMSRECORD Structure and Type

Field Name	Data Type	Brief Description
COMMUNICATIONNUMBER	Integer	The COMMUNICATIONNUMBER field contains the communication number assigned to the connection by the X.25 MCS. The possible values for this field are in the range 0 to (2**39) - 1.
QBIT	Boolean	The QBIT field contains the qualifier bit. When set, this field qualifies a data packet and corresponds to the qualifier bit in the X.25 network packet.
DBIT	Boolean	Use of this field is not currently supported. The DBIT field, when set, requests acknowledgment from the remote DTE and corresponds to a D-bit description of an X.25 level 3 packet.
DATAIDENTIFIER	Integer	Use of this field is not currently supported. The DATAIDENTIFIER field is used to identify the data message being sent or the data message being acknowledged when the DBIT field has been set to TRUE. The possible values for this field are in the range 0 to 65535.
ORIGINATOR	Integer	The ORIGINATOR field is used in conjunction with the CAUSE and DIAGNOSTIC fields. The ORIGINATOR field describes the originator of a message that is received by the application program. The possible values for this field are NETWORKORIGINATED, SYSTEMORIGINATED, and APPLICATIONORIGINATED.
CAUSE	Integer	The CAUSE field describes the reason the record was sent. It corresponds to the Cause field in an X.25 level 3 packet when the ORIGINATOR field contains the value NETWORKORIGINATED. The possible values for this field are in the range 0 to 255.

Table 3–3. X.25 COMSRECORD Structure and Type

Field Name	Data Type	Brief Description
DIAGNOSTIC	Integer	The DIAGNOSTIC field describes the diagnostic information sent with the record. It corresponds to the Diagnostic field in an X.25 level 3 packet when the ORIGINATOR field contains the value NETWORKORIGINATED. The possible values for this field are in the range 0 to 255.
ALREADYACCEPTED	Boolean	The ALREADYACCEPTED field is meaningful only with the X.25 MCS on a BNA Version 2 platform. This field is valid only with the INCOMINGCALL function. The X.25 MCS sets this field to TRUE on an incoming call if the connection has already been accepted by a CP 2000.
WAITFORCHANNEL	Boolean	The WAITFORCHANNEL field is valid only with the CALLREQUEST function. • On a BNA Version 1 platform, if a logical channel is not currently available, a TRUE value in this field instructs the X.25 MCS to hold the call to the remote DTE until a channel is available to make the connection. • On a BNA Version 2 platform, a TRUE value in this field instructs the X.25 MCS to initiate or wait for a dialogue with the CP 2000.
TRUNCATED	Boolean	The TRUNCATED field is valid only with the DCEDATA function. A TRUE value in this field indicates that the data message is truncated.
REMOTEADDRESSLENGTH	Integer	The REMOTEADDRESSLENGTH field contains the length of the REMOTEADDRESS field in hex digits. The maximum value for this field is 40. However, the X.25 MCS limits this field to 15 hex digits on a BNA Version 1 platform or a BNA Version 2 platform.

Table 3–3. X.25 COMSRECORD Structure and Type

Field Name	Data Type	Brief Description
REMOTEADDRESS	Hexadecimal	The REMOTEADDRESS field contains the address of the remote DTE endpoint in hex digits. The maximum value for this field is 40 hex digits. However, the X.25 MCS limits this field to 15 hex digits on a BNA Version 1 platform or a BNA Version 2 platform.
LOCALSUBADDRESSLENGTH	Integer	The LOCALSUBADDRESSLENGTH field contains the length of the LOCALSUBADDRESS in hex digits. The maximum value for this field is 14. However, the X.25 MCS limits this field to 10 hex digits on a BNA Version 1 platform or a BNA Version 2 platform.
LOCALSUBADDRESS	Hexadecimal	The LOCALSUBADDRESS field contains the local endpoint identification address in hex digits. The maximum value for this field is 14 hex digits. However the X.25 MCS limits this field to 10 hex digits on a BNA Version 1 platform or a BNA Version 2 platform. The data in this field must be left-justified, binary-coded decimal (BCD) characters.
FACILITIESLENGTH	Integer	The FACILITIESLENGTH field contains the length of the FACILITIES field specified in octets. This field corresponds to the X.25 level 3 Facility Length field. The maximum value for this field is 109.
FACILITIES	EBCDIC	The FACILITIES field contains untranslated information. It does not contain message data. This field corresponds to the X.25 level 3 facility field. The maximum value for this field is 109 octets. Information in the FACILITIES field is passed unchanged by the X.25 MCS directly to and from the X.25 network. Therefore, the application program must format the FACILITIES field exactly according to the CCITT standards in use by the X.25 network.

Table 3–3. X.25 COMSRECORD Structure and Type

Field Name	Data Type	Brief Description
ENSEMBLELENGTH	Integer	The ENSEMBLELENGTH field contains the length of the ENSEMBLE field specified in octets. The maximum value for this field is 17.
ENSEMBLE	EBCDIC	The ENSEMBLE field identifies the ensemble through which the specified message is routed. The same remote DTE address can be reached through different ensembles. The maximum value for this field is 17 octets. This field is used for load balancing and corresponds to a preferred station in the UK and US formats of X.25 records.
PHONENUMBERLENGTH	Integer	The PHONENUMBERLENGTH field contains the length of the PHONENUMBER field specified in hex digits. The maximum value for this field is 30. However, the X.25 MCS limits this field to 17 hex digits on a BNA Version 2 platform and ignores this field on a BNA Version 1 platform.
PHONENUMBER	Hexadecimal	The PHONENUMBER field contains the complete phone number, in hex digits, that a CP 2000 must call to establish a connection. This field is meaningful only on a BNA Version 2 platform for the CALLREQUEST function. It is ignored for all other functions.
DATALENGTH	Integer	The DATALENGTH field contains the length of the DATA field (specified in octets). For call user data, the maximum value for this field is 128. For message data, there is no maximum value.
DATA	EBCDIC	The DATA field contains data. This field corresponds to the data following an X.25 level 3 Data Packet header or untranslated message in the X.25 level 3 User Data field. This field is the only variable-length field in a PIR.

Using Records in COMS

The following pages describe techniques used to work with records in a COMS application program and considerations that affect the way the records are used. The information includes

- Accessing individual fields within a record
- Binding considerations for COMS

Accessing Header Fields

<input or output headers>

```
                    ┌←───────────────────────────┐
— <record ID> — . ──┴─┬─ <field ID> ──────────┬──┘───────────────────────┤
                      └─ <subscripted field ID> ┘
```

<subscripted field ID>

```
— <field ID> — [ — <subscript> — ] ──────────────────────────────────┤
```

Explanation

Input headers, output headers, and COMSRECORDS are defined in ALGOL as record structures whose fields have predefined names and purposes. The individual fields can be accessed through fully qualified record syntax.

When referencing fields in a record, each field must be uniquely identified. The field is qualified by the record identifier, the field identifier, and as needed, by a subscript field identifier.

The record ID construct is the user-declared name of the input header, output header or COMSRECORD.

Both the field and subscripted field identifiers are defined by COMS. The field ID construct identifies the COMS name for the field. If the field is subscripted, use the subscripted field ID. Subscripting is used to access a field in an embedded packed record with a header.

When a field within a record is passed as a parameter in a procedure call, the value of the field, rather than a reference to it, is passed. If you want to modify a field through a procedure call, pass the record itself (input header, output header, or COMSRECORD) rather than the field.

Additional information relating to the fields of input or output headers is included under “Input Header Structure and Type,” “Output Header Structure and Type,” and “COMSRECORD Structures and Types” in this section.

Examples

The example below accesses the subscripted field FIELDS.TRANSPARENT in the record MYHEADER.

```
MYHEADER.FIELDS.TRANSPARENT
```

In the following example, the input header named MYIN assigns the value 32 into the TEXTLENGTH field of the input header and the value of REQUESTDATA into word 7 of the conversation area field.

```
REAL REQUESTDATA;

INPUTHEADER MYIN (ARRAY CONVERSATION[0:8]);

MYIN.TEXTLENGTH := 32;

MYIN.CONVERSATION[6] := REQUESTDATA;
```

Binding Considerations for COMS

The ALGOL interface to COMS contains three types of header records: input headers, output headers and COMSRECORDs. The following paragraphs detail some considerations that apply when you use the Binder program to bind procedures or programs that contain COMS header records.

- A header record variable can be bound to another header record variable or to a star-bounded REAL array. A header record can also be bound to any other record type that can be bound to a star-bounded REAL array.

  The Binder program does not check the record structures for compatibility when they are bound. Because no checking occurs, the Binder program binds header record variables to similarly defined header record variables.
- When you bind a subprogram that declares COMS input and output headers, declare the headers in the global part of the subprogram.
- Procedures that have declared formal parameters can be bound, but no type checking is performed when the procedures are bound. Ensure that the types of the formal and actual parameters are identical.
- When a variable is declared in a subprogram, the declaration of the variable determines what the subprogram can do with the variable and whether the variable is properly protected against write access.
  - If the subprogram declares the variable as a header record variable, the header record variable can be accessed through the described fields.
  - If the subprogram declares the variable as another type of record variable, the variable can be accessed through the field names of the record. The semantic rules for that type of record variable are enforced.
  - If the subprogram declares the variable as a REAL array, no field-oriented access can be used. Assignment to the variable is permitted.

Refer to the *Binder Programming Reference Manual* for more information.

COMS Statements

The COMS interface supports statements that pertain to the use of COMS features and statements that provide synchronized recovery for application programs that update Data Management System II (DMSII) and Semantic Information Manager (SIM) databases.

The COMS interface supports the following two database statements. These statements provide synchronized recovery for application programs that update Data Management System II (DMSII) databases, as detailed in the *COMS Programming Guide*.

BEGINTRANSACTION	ENDTRANSACTION

The ALGOL interface to COMS also supports the following statements:

DISABLE	RECEIVE
ENABLE	SEND
MESSAGECOUNT	

This section describes each of the above statements. The statements are presented in alphabetical order. For information regarding when and why you use these statements, consult the *COMS Programming Guide*.

Refer to Section 5, "Using DMSII Transaction Processing System (TPS) Extensions," for the TPS statements that work with COMS. These statements are

BEGINTRANSACTION	MIDTRANSACTION
ENDTRANSACTION	OPEN

Access to the functional Semantic Information Manager (SIM) environment is accomplished through the use of a COMS window. Refer to Section 7, "Using the Semantic Information Manager (SIM) Interface," for the database management statements that work with COMS. These statements are

ABORTTRANSACTION	ENDTRANSACTION
BEGINTRANSACTION	OPEN
CANCELTRPOINT	SAVETRPOINT
CLOSE	

Refer to Section 6, “Using the Screen Design Facility Plus (SDF Plus) Interface,” for an explanation of how to access SDF Plus from COMS. No extensions specific to COMS are required for SDF Plus.

Additional information relating to COMS statements is included in Section 4, “Using the Data Management System II (DMSII) Interface”; Section 5, “Using DMSII Transaction Processing System (TPS) Extensions”; Section 6, “Using the Screen Design Facility Plus (SDF Plus) Interface”; and Section 7, “Using the Semantic Information Manager (SIM) Interface.”

COMS BEGINTRANSACTION Statement

<begintransaction statement>

```
— BEGINTRANSACTION  — <inputheadername> —┬──────────────────┬──────────────→
                                         └─ <message area> ─┘

→─┬──────────────────────────────────────────┬─ <restart data set> ───────→
  ├─ ( — <transaction record variable> — ) ──┤
  ├─ AUDIT ──────────────────────────────────┤
  └─ NOAUDIT ────────────────────────────────┘

→─┬──────────────────────────┬─────────────────────────────────────────────┤
  └─ <exception handling> ───┘
```

Explanation

The COMS BEGINTRANSACTION statement places a program in transaction state. It enables a program interfacing with COMS to support synchronization of transactions and recovery. The statement is used in application programs that update a DMSII database. It provides synchronized recovery if an exception occurs while a program is in transaction state. (The SIM BEGINTRANSACTION statement is used for SIM databases.)

Note: *At any given time, a program can be in transaction state with only one database. For proper recovery, the name of the database in transaction state must be the name of the database noted in the COMS Utility.*

If the message area is specified, COMS stores restart information in the transaction trail.

COMS updates the STATUSVALUE field of the declared input header with the result of the BEGINTRANSACTION statement.

Consult the *COMS Programming Guide* for more information about the STATUSVALUE field, synchronized recovery and transaction trails, message areas, the restart data set, and handling a BEGINTRANSACTION exception.

Additional information regarding the COMS BEGINTRANSACTION statement is included under “Service Function Result Values” and “STATUSVALUE Field Values” in this section. Related information is also included under “DMSII BEGINTRANSACTION Statement” in Section 4 and “SIM BEGINTRANSACTION Statement” in Section 7.

Additional information regarding the inputheadername construct is included under “Declaring an Input and Output Header” in this section. Information on the message area construct is included under “Declaring an Input and Output Header” in this section. Information on transaction processing and the exception handling construct is included under “Database Status Word” in Section 4, “Using the Data Management System II (DMSII) Interface.”

Explanation

The construct inputheadername identifies the declared input header.

The message area construct identifies the declared variable reserved for the actual message.

The transaction record variable construct identifies a transaction record created through the Transaction Processing System (TPS).

If AUDIT is specified, the restart area is captured. If NOAUDIT is specified, the restart area is not captured. AUDIT is the default action.

The restart data set contains the restart records an application program can access to recover database information after a system failure.

An exception is returned if the BEGINTRANSACTION statement is encountered while the program is in transaction state. An ABORT exception frees all records that the program locked. Note that deadlock can occur during execution of a BEGINTRANSACTION statement.

Additional information is included under “Declaring a Message Area” in this section, and under “Exception Processing” in Section 4, “Using the Data Management System II (DMSII) Interface.”

Example

The following BEGINTRANSACTION statement is for the input header declared as MYHEADER. COMS stores restart information in the transaction trail because the message area, MSG, is specified. Since AUDIT is included, the restart area is trapped. The restart data set is RDS.

```
BEGINTRANSACTION MYHEADER MSG AUDIT RDS;
```

DISABLE Statement

```
— DISABLE  — ( — <inputheadername> ─┬──────────────┬─ , — <keyname> — ) ────┤
                                    └─ TERMINAL ───┘
```

<keyname>

```
─┬─ <''alpha string literal''>─┬───────────────────────────────────────┤
 └─ <EBCDIC array row> ────────┘
```

Explanation

The DISABLE statement logically disconnects the program from the station in the STATION field of the declared input header.

The DISABLE statement can be used as an integer-valued function. The returned integer is the same as the value COMS places in the STATUSVALUE field of the input header. For example, a returned value of 0 (zero) means the STATION field of the header contains a valid station designator and the disconnect was successful.

COMS updates the FUNCTIONSTATUS field of the input header. Consult the *COMS Programming Guide* for an explanation of the FUNCTIONSTATUS and STATUS fields.

The construct inputheadername identifies the input header.

The word “TERMINAL” specifies a disconnect from a station. If it is not specified, it is assumed.

The valid values for the construct keyname are: “DIAL”, “DONTCARE”, “RELEASE”, and “RETAIN”. They are detailed in the *COMS Programming Guide*. Note that these values are literals and require quotation marks. If blanks are entered or no keyname is specified, the default state of “DONTCARE” is assumed.

Additional information relating to DISABLE statement is included under “FUNCTIONSTATUS Field Values” and “STATUSVALUE Field Values” in this section.

Additional information relating to the inputheadername construct is included under “Declaring an Input and Output Header” in this section.

Consult Volume 1 for an explanation of alpha string literals.

Additional information is included under “ENABLE Statement” in this section.

Examples

The following DISABLE statement disconnects a previously enabled dial-out station.

```
DISABLE(MYINPUT TERMINAL, "DIAL");
```

In the following example, the program is disconnected from the station specified in the STATION field of the input header INCOMS. If the station is a CP 2000 station, the physical attachment is released.

```
DISABLE (INCOMS TERMINAL, "RELEASE");
```

The following example of the DISABLE statement shows using the default options. Even though the TERMINAL option is not specified, the disconnect is from the station in the STATION field of the input header THEINPUTHEADER. Since no keyname is given, the default state is “DONTCARE”. If the station is a CP 2000, the terminal gateway decides whether to retain or release the physical attachment.

```
DISABLE(THEINPUTHEADER);
```

ENABLE Statement

<enable statement>

```
 ─ ENABLE  ─ ( ─ <inputheadername>─┬─────────────┬─ , ─ <keyname> ─ ) ────┤
                                   └─ TERMINAL ─┘
```

Explanation

The ENABLE statement logically connects COMS and the destination specified in the Station Designator field of the declared input header.

The ENABLE statement can be used as an integer-valued function. The returned integer is the same as the value COMS places in the STATUSVALUE field of the input header. For example, a returned value of 0 (zero) means the ENABLE was successful.

The STATUSVALUE field of the input header contains the status of the connect.

Consult the *COMS Programming Guide* for an explanation of the fields of the headers.

The construct inputheadername identifies the input header.

If the word “TERMINAL” is not specified, the ENABLE statement initializes the program with COMS. If TERMINAL is specified, the ENABLE statement performs a dynamic attachment to a station.

The valid keynames depend on whether the TERMINAL syntax is used in the ENABLE statement. “BATCH” and “ONLINE” cannot be specified if the word “TERMINAL” appears in the statement.

The other valid keynames are: “DIAL,” “NOWAIT,” “WAIT,” “WAITDIALOUT,” and “NOBUSY.”

The “(HOSTNAME=<hostname>)” syntax can be used with the TERMINAL option for “WAIT,” “NOWAIT,” “WAITDIALOUT,” and “WAITNOBUSY” keynames. HOSTNAME is the name of the host of the station in the Destination field. The hostname string is not checked for accuracy by the compiler; it is used by COMS at run time to define a host.

Note: *The keyname values are literals and require quotation marks.*

Consult the *COMS Programming Guide* for information on keynames and on batch and interactive processing.

Additional information relating to the ENABLE statement is included under “STATUSVALUE Field Values” and the “DISABLE Statement” in this section.

Additional information relating to the inputheadername construct is included under “Declaring an Input and Output Header” in this section. Information on the keyname construct is included under “DISABLE Statement” in this section.

Examples

The following ENABLE statement informs COMS that it is dealing with an interactive program:

```
ENABLE(MYINPUT,"ONLINE");
```

In the following example, the conversation area field of the input header holds the telephone number, the TEXTLENGTH field holds the telephone number length, and the STATION field holds the station designator. The statement connects the program for data transfer to a dial-out station.

```
ENABLE(MYHEADER TERMINAL,"DIAL");
```

The following example shows the syntax when a hostname, shown here as MACHINE, is specified. The hostname is the name of the host of the station in the Destination field. The hostname string is not checked for accuracy by the compiler; it is used by COMS at run time to define a host.

```
ENABLE(MYHEADER TERMINAL,"WAIT (HOSTNAME = MACHINE)");
```

COMS ENDTRANSACTION Statement

<endtransaction statement>

```
— ENDTRANSACTION  — <outputheadername with send options>─┬─────────────┬──→
                                                        ├─ AUDIT ─────┤
                                                        └─ NOAUDIT ───┘

→— <restart data set>─┬────────┬─┬─────────────────────────┬────────────┤
                      └─ SYNC ─┘ └─ <exception handling> ──┘
```

<outputheadername with send options>

```
— <outputheadername>─┬──────────────────────────┬─┬───────────────────┬───┤
                     └─ [ — <send options> — ] ─┘ └─ <message area> ──┘
```

Explanation

The COMS ENDTRANSACTION statement takes a program out of transaction state. It is used only in application programs that update a DMSII database. (The SIM ENDTRANSACTION statement is used for SIM databases.)

Two of the basic tasks performed by the COMS ENDTRANSACTION statement are to

- Ensure that the information passed to COMS during the midtransaction phase is safely stored in the transaction trail.
- Perform a DMSII ENDTRANSACTION.

If the DMSII ENDTRANSACTION returns an exception, COMS resubmits the current transaction after synchronized recovery is complete.

COMS updates the STATUSVALUE field of the declared output header with the result of the ENDTRANSACTION statement.

Consult the *COMS Programming Guide* for more information on the STATUSVALUE field, synchronized recovery, the restart data set, and handling an ENDTRANSACTION exception.

The construct outputheadername identifies the output header.

The send options describe the carriage and message controls that can be used with a send operation.

The message area construct identifies the declared variable reserved for the actual message. If a message area is specified, COMS ensures that the message is sent before the DMSII ENDTRANSACTION is executed.

If AUDIT is specified, the restart area is captured. If NOAUDIT is specified, the restart area is not captured. AUDIT is the default action.

The restart data set contains the restart records an application program can access to recover database information after a system failure.

The word “SYNC” forces a syncpoint.

An exception is returned if an ENDTRANSACTION statement is attempted and the program is not in the transaction state. Records are freed in all cases. The transaction is not applied to the database.

Additional information relating to the COMS ENDTRANSACTION statement is included under “Exception Processing” in Section 4, “Using the Data Management System II (DMSII) Interface,” “Service Function Result Values,” “STATUSVALUE Field Values,” and “SEND Statement” in this section. Related information is also included under “DMSII ENDTRANSACTION Statement” in Section 4, “Using the Data Management System II (DMSII) Interface,” and under “SIM ENDTRANSACTION Statement” in Section 7, “Using the Semantic Information Manager (SIM) Interface.”

Additional information regarding the exception handling construct is included under “Exception Processing” in Section 4, “Using the Data Management System II (DMSII) Interface.” Information on the outputheadername construct is included under “Declaring an Input and Output Header” in this section. Information regarding the send options construct is included under “SEND Statement” in this section. Information on the message area construct is included under “RECEIVE Statement” in this section.

Example

In the following example, the output header is MYOUT. The send option instructs the system to skip two lines. Since a message area (MSG) is specified, a message will be sent during synchronized recovery. The restart area is captured in the restart data set RDS.

```
ENDTRANSACTION MYOUT [SKIP 2] MSG AUDIT RDS;
```

MESSAGECOUNT Statement

<messagecount statement>

```
— MESSAGECOUNT  — ( — <inputheadername> — ) ——————————————————————————————┤
```

Explanation

The MESSAGECOUNT statement returns the number of queued messages for the program. COMS places the number of messages into the MESSAGECOUNT field of the designated input header.

The MESSAGECOUNT statement can be used as an integer-valued function. The returned integer is the number of queued messages. The STATUSVALUE field of the input header is also updated. It contains the status of the MESSAGECOUNT request. A status value of 0 (zero) means the operation was successful.

Consult the *COMS Programming Guide* for more information about the MESSAGECOUNT and STATUSVALUE fields.

Additional information relating to the MESSAGECOUNT statement is included under "STATUSVALUE Field Values" in this section.

Additional information relating to the inputheadername construct is included under "Declaring an Input and Output Header" in this section.

The inputheadername construct identifies the input header.

Example

The number of messages associated with the input header MYINPUT is assigned to the variable COUNT and COMS puts the message count into the MESSAGECOUNT field of MYINPUT.

```
COUNT := MESSAGECOUNT(MYINPUT);
```

RECEIVE Statement

<receive statement>

```
— RECEIVE  — ( — <inputheadername>—————————————————— , — <message area>— ) —|
                                  └ [ — DONTWAIT  — ] ┘
```

<message area>

```
—┬— <EBCDIC array row> —┬——————————————————————————————————————————|
 └— <ADDS structure> ———┘
```

Explanation

The RECEIVE statement requests that a message be transferred from the program queue to the designated message area. Information about the message is provided in the specified input header.

The RECEIVE statement can also be used as an integer-valued function. The returned integer is the same as the value COMS places in the STATUSVALUE field of the input header. For example, a returned value of 0 (zero) means a message was received successfully.

Consult the *COMS Programming Guide* for an explanation of the fields of the input header.

Additional information relating to the inputheadername construct is included under "Declaring an Input and Output Header" in this section.

The construct inputheadername identifies the input header to receive the message.

The DONTWAIT option enables you to specify that a receive operation is not to wait for a message. If DONTWAIT is not specified, the receive operation waits for a message.

The message area construct identifies the variable into which the actual message is to be placed.

Additional information relating to the RECEIVE statement is included under "STATUSVALUE Field Values" in this section.

Example

In the following example, the first RECEIVE statement is a conditional receive operation. The variable COMSSTATUS, as well as the status value, is nonzero if no message is waiting or if some other exception occurs. The second receive operation waits forever or until a message comes in.

```
INTEGER COMSSTATUS;
INTEGER RECEIVECODE;

  COMSSTATUS := RECEIVE(MYINPUT [ DONTWAIT ],MSG);
  .
   .
   .
  IF RECEIVE(MYINPUT,MSG) > 0  THEN
   BEGIN
   RECEIVECODE := MYINPUT.STATUS;
   CASE RECEIVECODE OF
   BEGIN
   95:
   HANDLE_AGENDA_ERROR;
   ...
   ...
   ...
   ELSE:
   HANDLE_COMS_ERROR;
   END;
   END
  ELSE
   PROCESS_MESSAGE;
```

SEND Statement

<send statement>

```
 — SEND  — ( — <outputheadername>——————————————————————— , — <message length>————————→
                                   |                    |
                                   └ [ — <send options> — ] ┘

 →— , — <message area> — ) ———————————————————————————————————————————————————————————┤
```

<send options>

```
——┬————————————————————————————————┬┬——————————┬——————————————————————————————————————→
  |                                ||          |
  └ <message control indicator> ———┘├— BEFORE —┤
                                    |          |
                                    └— AFTER  ——┘

   ┌←——————————————————————— , ————————————————┐
   |                                            |
→——┴—┬—/1\——┬— SKIP  ——┬— <arithmetic expression> —┬——————————————————————————————————┤
     |      |          |                            |
     |      └— SPACE  —┘                            |
     |                                              |
     ├—/1\— PAGE  ——————————————————————————————————┤
     |                                              |
     ├—/1\— NOCR  ——————————————————————————————————┤
     |                                              |
     └—/1\— NOLF  ——————————————————————————————————┘
```

<message control indicator>

```
—┬— ESI ———————————————————————┬————————————————————————————————————————————————————————┤
 |                             |
 ├— EMI ———————————————————————┤
 |                             |
 ├— EGI ———————————————————————┤
 |                             |
 └— <arithmetic expression> ———┘
```

<message length>

```
—┬— <arithmetic expression> —┬——————————————————————————————————————————————————————————┤
 |                           |
 └— * ———————————————————————┘
```

Explanation

The SEND statement requests a message or portion of a message to be transferred from the specified message area to the program or station queue designated by either the DESTINATIONDESG or AGENDA field of the output header.

The SEND statement can be used as an integer-valued function. The returned integer is the same as the value COMS places in the STATUSVALUE field of the output header and represents the result of the transfer. For example, a returned value of 0 (zero) means the transfer was successful.

Delivery confirmation uses the CONFIRMFLAG and CONFIRMKEY fields of the output header. If the value of the CONFIRMFLAG is TRUE when the SEND statement is executed, the three bytes of the CONFIRMKEY field are used as the tag for delivery confirmation.

Consult the *COMS Programming Guide* for an explanation of the fields of the headers.

The outputheadername identifies the output header.

Additional information relating to the outputheadername construct is included under “Declaring an Input and Output Header” in this section.

The send options describe the message controls and carriage controls to be applied to the send operation.

A message control indicator is either the type or arithmetic value used to select a type of output for the message. The output can be nonsegmented or segmented. Segmented messages can be defined by changing the TEXTLENGTH field of the output header and using one of the three segmenting options. The TEXTLENGTH field is used by COMS to determine how much of the message area variable is to be used as the segment in the SEND statement. Unless the TEXTLENGTH field is set, COMS uses the entire message area.

The message control indicator types and their arithmetic equivalents are shown in the following table. The default is EMI (the value 2). For a detailed explanation of the indicators, consult the *COMS Programming Guide*.

Type	Value	Type of Indicator
ESI	1	End-of-Segment Indicator
EMI	2	End-of-Message Indicator (default)
EGI	3	End-of-Group Indicator

If multiple SEND statements are processed with the ESI control, and a SEND statement with the EMI control is processed in the middle of these, the SEND statement with the EMI control is sent immediately, while the other statements wait until one of the ESI output conditions is TRUE. This means that, in some cases, it can appear that the messages are not being sent in the correct order.

The results of the carriage control options can differ depending on the output device. If no carriage controls are specified, the default value of AFTER SPACE 1 is used. This default value sends the message and advances one line, consistent with ALGOL I/O.

The carriage control options, summarized in the following list, pertain to the output device.

- BEFORE and AFTER determine whether the carriage control action is performed before or after the message is sent to the output device. The BEFORE option causes a carriage control action and then sends the message. The AFTER option sends the message and then performs a carriage control action.
- SKIP causes the printer to skip to the channel specified by the value of the arithmetic expression.
- SPACE causes the printer to space the number of lines specified by the arithmetic expression.
- PAGE skips to the next page.
- NOCR suppresses the carriage return.
- NOLF suppresses line feed.

The message length construct gives the length, in bytes, of the data contained in the message area. If a value is specified in the message length construct, the TEXTLENGTH field of the output header is updated with that value.

Additional information relating to the SEND statement is included under “Service Function Result Values” and “STATUSVALUE Field Values” in this section.

Example

The following SEND statement sends the message specified by the EBCDIC array MSG, with a text length of 32 characters, and then uses the SKIP option to skip to channel 10 using the message control indicator EMI.

```
EBCDIC ARRAY MSG[0:32];

 IF SEND(MYOUT [EMI AFTER SKIP 10], 32, MSG) THEN
 BEGIN
 CASE SENDCODE OF
 BEGIN
 98:
 COMS_SECURITY_VIOLATION;
 ...
 ...
 ...
 ELSE:
 HANDLE_COMS_ERROR
 END;
 END
 ELSE
 RESUME_PROCESS;
```

Error Handling

When an error occurs during communication processing, the result of a COMS statement can be determined in two ways:

- The COMS statement can be used as a function.
- The value stored in the STATUSVALUE field of the header can be compared to the error codes for the particular statements.

All COMS statements can be used as functions. Each statement returns an integer value. Except for the MESSAGECOUNT statement, the integer value is the same as the status value COMS places in the STATUSVALUE field of the respective header. The MESSAGECOUNT statement returns the value COMS places in the MESSAGECOUNT field.

When you detach a station or program, or when you use the Modular Function Index (MFI), the status of the operation is reported in the FUNCTIONINDEX field of the input header. The value stored in this field can be used to check if the detachment was successful or if an error occurred.

STATUSVALUE Field Values

The values and meanings for the STATUSVALUE field of the input header and output header and for the status of a call are listed and detailed in an appendix of the *COMS Programming Guide*.

FUNCTIONSTATUS Field Values

COMS places values in the FUNCTIONSTATUS field of the input header when COMS performs a DISABLE statement or any MFI operation. These values are listed and detailed in the *COMS Programming Guide*. You can define these values in the program by using the DEFINE declaration, as shown in Volume 1 of this manual. For example,

```
DEFINE CONTROLMSG = -1#, GOOD_DELIVERY = -12# ;
```

Exception-Condition Statements and DMTERMINATE

If you must use exception-condition statements to close a database, use the DMTERMINATE statement for those exceptions not specifically handled by the program.

Additional information relating to the DMTERMINATE statement is included under “DMTERMINATE Statement” in Section 4, “Using the Data Management System II (DMSII) Interface.”

COMS Service Functions

COMS service functions are entry points that enable programs to obtain information on COMS entities and to translate designators and names that represent these entities.

The majority of the service functions, except STATION_TABLE_INITIALIZE, return an integer result. The integer definitions are described under “Service Function Result Values” in this section.

On input, all entity names must be terminated with a blank character.

To determine the length of a string returned by a service function, the program must test for a blank. The string is always terminated by a blank character.

The following pages briefly describe the service functions, detail their calling parameters, and define the values used to report the results of the call.

Consult the *COMS Programming Guide* for further information on the COMS service functions.

Additional information relating to COMS service functions is included under “Service Function Result Values,” “Designator Data Type,” “COMS Statements,” “Error Handling,” “Linking to COMS,” and “Designators for COMS Entities” in this section.

Functional Descriptions

The COMS service functions can be called by COMS application programs and by processing items. The service functions and a description of how to use input and output headers in conjunction with service functions are covered in the *COMS Programming Guide*. The service functions are explained briefly in Table 3–4.

Table 3–4. A Brief Explanation of COMS Service Functions

Service Function	Brief Explanation
CONVERT_TIMESTAMP	Converts value in a COMS TIMESTAMP field to the date or time as an EBCDIC array.
GET_DESIGNATOR_ARRAY_USING_DESIGNATOR	Gets a designator vector from a structure represented by a designator.
GET_DESIGNATOR_USING_DESIGNATOR	Gets a specific designator out of the structure represented by a designator.
GET_DESIGNATOR_USING_NAME	Converts a COMS entity name to a COMS designator.
GET_INTEGER_ARRAY_USING_DESIGNATOR	Gets a vector of integers from the structure represented by a designator.
GET_INTEGER_USING_DESIGNATOR	Gets a specific integer out of the structure represented by a designator.
GET_NAME_USING_DESIGNATOR	Converts a COMS designator to a COMS name for that designator.
GET_REAL_ARRAY	Gets a structure of data with no connection to any entity.
GET_STRING_USING_DESIGNATOR	Gets an EBCDIC string out of the structure represented by a designator.
STATION_TABLE_ADD	Adds a station designator to an existing station table.
STATION_TABLE_INITIALIZE	Initializes a station table so that station index values can be added using STATION_TABLE_ADD.
STATION_TABLE_SEARCH	Finds a station designator within a station table.
TEST_DESIGNATORS	Tests whether a designator is part of a structure represented by another designator.

Declaring COMS Service Functions

To declare the individual functions needed for an application, use the PROCEDURE declaration with the library entry point specification. The syntax for each service function is shown on the following pages.

When declaring the COMS library in a program that is to use one of the service functions, you must not use the name “COMSSUPPORT.” If you use “COMSSUPPORT” in your program, the compiler automatically generates a hidden library declaration of “COMSSUPPORT” when it encounters an INPUTHEADER or OUTPUTHEADER declaration. This declaration creates a conflict in your program. You can use a name like SERVICE_LIB.

Example 1: Use of FUNCTIONNAME to Access the Service Functions

```
LIBRARY SERVICE_LIB (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSUPPORT.";
```

Example 2: Use of TITLE

```
LIBRARY SERVICE_LIB (LIBACCESS = BYTITLE, TITLE = "SYSTEM/COMS ON PACK.");
```

Consult Volume 1 of this manual for a complete explanation of the PROCEDURE declaration, its syntax, and its constructs.

Consult the *COMS Programming Guide* for the valid designators for COMS entities and for the service function values. The guide contains detailed information regarding each service function.

Additional information relating to declarations of COMS service functions is included under “Linking to COMS” in this section.

CONVERT_TIMESTAMP

The following example declares a procedure to convert a COMS TIMESTAMP field to a date or time EBCDIC array.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE CONVERT_TIMESTAMP
 (ENTY_TIMESTAMP, ENTY_TYPE, ENTY_TIME);
      VALUE                 ENTY_TYPE;
      REAL                  ENTY_TIMESTAMP;
      INTEGER               ENTY_TYPE;
      EBCDIC ARRAY          ENTY_TIME[0];
      LIBRARY SERVICE_LIB;
```

ENTY_TIMESTAMP is the TIME (6) timestamp used as input in the conversion.

The ENTY_TYPE is the requested information. The only valid values are 72 for TIME and 71 for DATE. The time is returned in the form HHMMSS. The date is returned in the form MMDDYY.

ENTY_TIME is the array where the result from COMS is returned.

COMS provides a timestamp in the TIME(6) format for application programs using a direct-window interface. The TIME(6) intrinsic returns a unique 48-bit pattern for the time and date. The TIME(6) timestamp returns positive numbers for the years 1970 through 1986 and negative numbers for the years 1987 and beyond. This affects software that uses arithmetic compare operators, such as greater than, less than, or equal to, against the TIME(6) format timestamp. Consult Volume 1 for a definition and explanation of the TIME function.

GET_DESIGNATOR_ARRAY_USING_DESIGNATOR

The following example declares a procedure to retrieve a designator vector from the structure represented by the designator.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSUPPORT.");
INTEGER PROCEDURE GET_DESIGNATOR_ARRAY_USING_DESIGNATOR
 (ENTY_DESIGNATOR, ENTY_DESGTOTAL, ENTY_DESGVECTOR);
      INTEGER               ENTY_DESGTOTAL;
      REAL                  ENTY_DESIGNATOR;
      REAL ARRAY            ENTY_DESGVECTOR[0];
      LIBRARY SERVICE_LIB;
```

The ENTY_DESIGNATOR is the designator that represents the structure. The only valid entry is a station list designator.

ENTY_DESGTOTAL is the total number of designators returned in the vector.

The ENTY_DESGVECTOR is the vector in which the designators for the stations are returned.

GET_DESIGNATOR_USING_DESIGNATOR

The following example declares a procedure to retrieve a specific designator from the structure represented by the designator:

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE GET_DESIGNATOR_USING_DESIGNATOR
 (ENTY_DESIGNATOR, ENTY_TYPE, ENTY_DESGRES);
      VALUE       ENTY_TYPE;
      REAL        ENTY_DESIGNATOR,
                  ENTY_DESGRES;
      INTEGER     ENTY_TYPE;
      LIBRARY     SERVICE_LIB;
```

The ENTY_DESIGNATOR is the designator that represents the structure. All designators shown in “Designators for COMS Entities” can be used.

The ENTY_TYPE is the requested designator type. For example, DEVICE can be used only as an entry for a station designator.

Valid ALGOL values for the various structures are given in the following table.

If ENTY_DESIGNATOR represents . . .	The valid ALGOL value is . . .
Any designator	52 - INSTALLATION_DATA_LINK
Program	5 - SECURITY
Station	9 - DEVICE
	5 - SECURITY
User	5 - SECURITY
XATMI Call	9 - DEVICE
	22 - XATMI SERVICE

ENTY_DESGRES is the designator returned by COMS.

Additional information relating to COMS designators is included under “Designators for COMS Entities” in this section.

GET_DESIGNATOR_USING_NAME

The following example declares a procedure to convert a COMS entity name to a COMS designator.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = " COMSSUPPORT.");
INTEGER PROCEDURE GET_DESIGNATOR_USING_NAME
 (ENTY_NAME, ENTY_TYPE, ENTY_DESIGNATOR);
      VALUE              ENTY_TYPE;
      EBCDIC ARRAY       ENTY_NAME[0];
      REAL               ENTY_DESIGNATOR;
      INTEGER            ENTY_TYPE;
      LIBRARY SERVICE_LIB;
```

The ENTY_NAME contains the name of an entity, for example a window name, that is terminated with a blank character. If the entity is an agenda, a trancode, or installation data, and if the program calling the service function is running in another window or outside of COMS, the format of the entity name can be

```
<entity name> OF <window name>
```

For installation data, use the “ALL” entity when no window is specified and the window in which the program is running does not have an entity of the same name.

The ENTY_TYPE is the value for the requested name. For a list of valid entity types, refer to “Designators for COMS Entities” in this section.

The ENTY_DESIGNATOR is the returned designator.

To ensure the return of a valid designator when the entity is an agenda, trancode, or installation data

- Call the service function only from a direct-window program or processing item.
- Call the service function only after a direct-window program has executed an ENABLE statement.
- If you are calling from a processing item, do not enable the program to call the service function until it has executed a FREEZE statement.

Additional information relating to COMS designators is included under “Designators for COMS Entities” in this section.

GET_INTEGER_ARRAY_USING_DESIGNATOR

The following example declares a procedure to retrieve an array of integers from the structure represented by the designator.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE GET_INTEGER_ARRAY_USING_DESIGNATOR
 (ENTY_DESIGNATOR, ENTY_TYPE, ENTY_INTEGERTOTAL,
ENTY_INTEGERVECTOR);
      VALUE                  ENTY_TYPE;
      REAL                   ENTY_DESIGNATOR;
      INTEGER                ENTY_INTEGERTOTAL, ENTY_TYPE;
      INTEGER ARRAY          ENTY_INTEGERVECTOR[0];
      LIBRARY SERVICE_LIB;
```

The ENTY_DESIGNATOR is the designator that represents the structure. All designators shown in “Designators for COMS Entities” can be used.

The ENTY_TYPE describes which integer vector is requested. For example, INSTALLATION_INTEGER_ALL can be used as an entry for all designators. However, MIXNUMBERS is valid only if the designator represents a program.

Valid ALGOL values for the various structures are given in the following table.

If ENTY_DESIGNATOR represents . . .	**The valid ALGOL value is . . .**
Any designator	45 - INSTALLATION_INTEGER_ALL
Program	84 - MIXNUMBERS

The ENTY_INTEGERTOTAL is the number of integers returned in the vector. ENTY_INTEGERVECTOR is the vector itself.

Additional information relating to COMS designators is included under “Designators for COMS Entities” in this section.

GET_INTEGER_USING_DESIGNATOR

The following example declares a procedure to extract a specific integer from the structure represented by the designator.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE GET_INTEGER_USING_DESIGNATOR
 (ENTY_DESIGNATOR, ENTY_TYPE, ENTY_INTEGER);
      VALUE             ENTY_TYPE;
      REAL              ENTY_DESIGNATOR;
      INTEGER           ENTY_TYPE, ENTY_INTEGER;
      LIBRARY SERVICE_LIB;
```

The ENTY_DESIGNATOR is the designator representing the structure. All designators shown in “Designators for COMS Entities” can be used.

The ENTY_TYPE describes which integer is requested. For example, INSTALLATION_INTEGER_4 can be used as an entry for all designators. However, CURRENT_USER_COUNT is valid only if the designator represents a window.

Valid ALGOL values for the various structures are given in the following table.

If ENTY_DESIGNATOR represents . . .	The valid ALGOL value is . . .
Any designator	41 - INSTALLATION_INTEGER_1 42 - INSTALLATION_INTEGER_2 43 - INSTALLATION_INTEGER_3 44 - INSTALLATION_INTEGER_4
Program	61 - QUEUE_DEPTH 62 - MESSAGE_COUNT 63 - LAST_RESPONSE 64 - AGGREGATE_RESPONSE
Station	83 - LSN
Window	81 - MAXIMUM_USER_COUNT 82 - CURRENT_USER_COUNT
XATMI Buffer	88 - LENGTH

The ENTY_INTEGER is the result. Additional information relating to COMS designators is included under “Designators for COMS Entities” in this section.

GET_NAME_USING_DESIGNATOR

The following example declares a procedure to convert a COMS designator to a COMS name.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE GET_NAME_USING_DESIGNATOR
 (ENTY_DESIGNATOR, ENTY_NAME);
      REAL              ENTY_DESIGNATOR;
      EBCDIC ARRAY      ENTY_NAME [0];
      LIBRARY SERVICE_LIB;
```

The ENTY_DESIGNATOR is the supplied designator. All valid designators, as shown in “Designators for COMS Entities,” can be used, except for a SECURITY designator.

The ENTY_NAME is the returned name. It is a string of 1 to 255 characters terminated with a blank character.

Additional information relating to COMS designators is included under “Designators for COMS Entities” in this section.

GET_REAL_ARRAY

The following example declares a procedure to retrieve a structure of data that has no connection to any entity.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE GET_REAL_ARRAY
 (ENTY_TYPE, ENTY_REALTOTAL, ENTY_REALVECTOR);
     VALUE                 ENTY_TYPE;
     INTEGER               ENTY_TYPE, ENTY_REALTOTAL;
     REAL ARRAY            ENTY_REALVECTOR[0];
     LIBRARY SERVICE_LIB;
```

The ENTY_TYPE is the requested structure of data. The only valid value is 65.

ENTY_REALTOTAL is the total number of elements returned in the array.
ENTY_REALVECTOR is the array where the information is returned.

The service function returns a table. Refer to the *COMS Programming Guide* for further details.

GET_STRING_USING_DESIGNATOR

The following example declares a procedure to retrieve an EBCDIC string from the structure represented by the designator.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE GET_STRING_USING_DESIGNATOR
 (ENTY_DESIGNATOR, ENTY_TYPE, ENTY_STRINGTOTAL, ENTY_STRING);
      VALUE                 ENTY_TYPE;
      REAL                  ENTY_DESIGNATOR;
      INTEGER               ENTY_STRINGTOTAL, ENTY_TYPE;
      EBCDIC ARRAY          ENTY_STRING[0];
      LIBRARY SERVICE_LIB;
```

The ENTY_DESIGNATOR is the designator that represents the structure. All designators shown in “Designators for COMS Entities” can be used.

The ENTY_TYPE describes which string is requested.

Valid ALGOL values and ENTY-TYPE names for the various structures are given in the following table.

If ENTY_DESIGNATOR represents . . .	**The valid ALGOL value is . . .**
Any designator	46 - INSTALLATION_STRING_1 47 - INSTALLATION_STRING_2 48 - INSTALLATION_STRING_3 49 - INSTALLATION_STRING_4 50 - INSTALLATION_HEX_1 51 - INSTALLATION_HEX_2
Station designator	87 - HOSTNAME 95 - LANGUAGE 120 - CONVENTION
XATMI Buffer	89 - TYPE 90 - SUBTYPE

The ENTY_STRINGTOTAL is the number of valid characters in the string. ENTY_STRING is the returned string.

Additional information relating to COMS designators is included under “Designators for COMS Entities” in this section.

STATION_TABLE_ADD

The following example declares a procedure that adds a station designator to an existing table of station designators (sometimes called a station table). The procedure accepts the station table and a station designator. It returns a unique index into the station table.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMMSUPPORT.");


INTEGER PROCEDURE STATION_TABLE_ADD (STATION_HASH,
                                     STATION_DESIGNATOR);

      ARRAY STATION_HASH[0];
      REAL STATION_DESIGNATOR;
      LIBRARY SERVICE_LIB ;
```

STATION_HASH represents the station table. The station table is implemented as a hash table.

STATION_DESIGNATOR is the designator of the station that is added to the station table.

STATION_TABLE_INITIALIZE

The following example declares a procedure that initializes a table of station designators (sometimes called a station table). The procedure accepts a station table and a table modulus.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");


PROCEDURE STATION_TABLE_INITIALIZE (STATION_HASH, SHMOD);

      ARRAY STATION_HASH[0];
      INTEGER SHMOD;
      LIBRARY SERVICE_LIB;
```

STATION_HASH represents the station table. The station table is implemented as a hash table.

SHMOD is the table modulus. The modulus determines the density of the station table and the time required to access it.

- For fast access and lower table density, choose a value for the modulus that is twice the maximum number of entries in the station table.
- For slower access and greater table density, choose a value for the modulus that is one half of the maximum number of entries in the station table.

STATION_TABLE_SEARCH

The following example declares a procedure that finds a given station designator within a table of station designators (sometimes called a station table). The procedure accepts a station table and a station designator. It returns the index of the station designator within the station table. If the station designator is not found, the returned index is zero.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");


INTEGER PROCEDURE STATION_TABLE_SEARCH (STATION_HASH,
                                        STATION_DESIGNATOR);

      ARRAY STATION_HASH[0];
      REAL STATION_DESIGNATOR;
      LIBRARY SERVICE_LIB;
```

STATION_HASH represents the station table. The station table is implemented as a hash table.

STATION_DESIGNATOR is the designator of the desired station (the station that the procedure looks for in the station table).

TEST_DESIGNATORS

The following example declares a procedure to test if a designator is part of a structure represented by another designator.

```
LIBRARY SERVICE_LIB
 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "COMSSUPPORT.");
INTEGER PROCEDURE TEST_DESIGNATORS
 (ENTY_DESIGNATOR_1, ENTY_DESIGNATOR_2);
      REAL   ENTY_DESIGNATOR_1,
             ENTY_DESIGNATOR_2;
LIBRARY SERVICE_LIB;
```

ENTY_DESIGNATOR_1 and ENTY_DESIGNATOR_2 are both designators. The order in which they are passed does not affect the service function. However, only device, device list, security, and security category designators are valid. Device and device list designators can be used in combination. Security and security category designators can be used in combination. The valid designators are:

CATEGORY_LIST

DEVICE

DEVICE_LIST

INSTALLATION_DATA

INSTALLATION_DATA_LINK

INSTALLATION_HEX_1

INSTALLATION_HEX_2

INSTALLATION_INTEGER_ALL

INSTALLATION_INTEGER_1

INSTALLATION_INTEGER_2

INSTALLATION_INTEGER_3

INSTALLATION_INTEGER_4

INSTALLATION_STRING_1

INSTALLATION_STRING_2

INSTALLATION_STRING_3

INSTALLATION_STRING_4

SECURITY

SECURITY_CATEGORY

SECURITY_CATEGORY_LIST

The valid ALGOL values for these designators are listed in “Designators for COMS Entities” in this section.

Designators for COMS Entities

Each entity in the COMS configuration has an associated designator that can be used in service calls. Table 3–5 lists the most common entities, their ALGOL values, and the information a program can request. Table 3–6 lists the types for the installation data. Consult the *COMS Programming Guide* for information on passing these values to service functions and for a complete listing of values.

Each designator for agendas, trancodes, and installation data must uniquely identify a particular combination of a window and that entity. Each designator for a station must uniquely identify a particular combination of a window, a dialogue, and a station.

Because the layout of COMS designators can change with each software release, a program must not preserve any designators across executions. It is advisable not to use designators as keydata in a database.

Table 3–5. COMS Entities

Entity Type	Value	Type of Information
AGENDA	3	Name
		Installation data
AGGREGATE_RESPONSE	64	
CURRENT_USER_COUNT	82	
DATABASE	13	Name
		Installation data
DATE	71	
DEVICE	9	Name
		Installation data
DEVICE_LIST	11	Name
		Installation data
INSTALLATION_DATA	20	Name
		Installation data
LAST_RESPONSE	63	
LIBRARY	18	Name
		Installation data
LSN	83	
MAXIMUM_USER_COUNT	81	
MESSAGE_COUNT	62	
MIX_NUMBERS	84	

Table 3–5. COMS Entities

Entity Type	Value	Type of Information
PROCESSING_ITEM	14	Name
		Installation data
PROCESSING_ITEM_LIST	15	Name
		Installation data
PROGRAM	4	Name
		Installation data
		Security designator
		Current input queue depth
		Total number of input messages handled
		Response time for last transaction
		Response time aggregate
		Mixnumbers for active copies
SECURITY	5	
SECURITY_CATEGORY	8	Name
		Installation data
SECURITY_CATEGORY_LIST	19	Name
		Installation data
STATION	1	Name
		Installation data
		Logical station number
		Security designator
		Device designator
		Language
		Convention
STATION_LIST	10	Name
		Installation data
		Stations in list
STATISTICS	65	
TIME	72	
TRANCODE	16	

Table 3–5. COMS Entities

Entity Type	Value	Type of Information
QUEUE_DEPTH	61	
USERCODE	2	Name
		Installation data
WINDOW	12	Name
		Installation data
		Maximum number of users
		Current number of users
WINDOW_LIST	17	Name
		Installation data
XATMI SERVICE	22	Name
		Installation data

The types and values for installation data are shown in Table 3-6.

Table 3–6. Installation Data Values

Entity Type	Value
INSTALLATION_DATA	20
INSTALLATION_INTEGER_1	41
INSTALLATION_INTEGER_2	42
INSTALLATION_INTEGER_3	43
INSTALLATION_INTEGER_4	44
INSTALLATION_INTEGER_ALL	45
INSTALLATION_STRING_1	46
INSTALLATION_STRING_2	47
INSTALLATION_STRING_3	48
INSTALLATION_STRING_4	49
INSTALLATION_HEX_1	50
INSTALLATION_HEX_2	51
INSTALLATION_DATA_LINK	52

Service Function Types and Values

The types used by the generalized service functions are detailed in the *COMS Programming Guide*. Each type indicates which data item or items are being requested. When passing values to the service functions, use the DEFINE declaration as shown in Volume 1. For example,

```
DEFINE AGENDA = 3#;
```

Service Function Result Values

Table 3–7. Service Function Result Values

Value	Definition
0	The function successfully completed.
1	An invalid name or designator was supplied.
2	An invalid designator was supplied.
3	The supplied array was too short to hold the output.
4	The requested installation data was not present.

COMS Sample Program

The following sample program monitors a sailboat race and updates a DMSII database by using features of the COMS direct-window interface. The program illustrates the techniques used in writing transaction processors that enable synchronized recovery.

The program runs in a COMS environment that has been configured to include a DMSII database called SAILDB. The database contains three data sets.

- RACE_CALENDAR contains one record for every race.
- ENTRY contains one record for each boat entered in the race. A boat can have multiple records, depending on the number of races it enters.
- RDS is the restart data set.

An example of a program using COMS and a SIM database is included under “Example 4: Using COMS with a SIM Database” in Section 7, “Using the Semantic Information Manager (SIM) Interface.”

```
BEGIN
%          ONLINESAIL
   REAL
   COMS_STATUS;
   TYPE INPUTHEADER
   COMS_IN_TYPE (ARRAY CONVERSATION [0:59]);
 COMS_IN_TYPE
   COMS_IN;
 OUTPUTHEADER
   COMS_OUT;
   DATABASE
   SAILDB;
 DEFINE
  GOEOT = 99 #,
  TEXT_LEN= 113 #;
 EBCDIC ARRAY
   SCRATCH[0:255],
   MSG_TEXT[0 : TEXT_LEN-1];
   DEFINE          MSG_TCODE = MSG_TEXT[0] #,
 %               MSG_FILLER
 %               MSG_CREATE_RACE
                     MSG_CR_ID = INTEGER(MSG_TEXT[7],6) #,
                     MSG_CR_NAME = MSG_TEXT[13] #,
                     MSG_CR_DATE = MSG_TEXT[33] #,
                     MSG_CR_TIME = MSG_TEXT[39] #,
                     MSG_CR_LOCATION = MSG_TEXT[43] #,
                     MSG_CR_SPONSOR = MSG_TEXT[63] #,
%                   FILLER
%               MSG_ADD_ENTRY REDEFINES MSG_CREATE_RACE
                     MSG_AE_RACE_ID = INTEGER(MSG_TEXT[7],6) #,
                     MSG_AE_ID = MSG_TEXT[13] #,
                     MSG_AE_NAME = MSG_TEXT[19] #,
                     MSG_AE_RATING = INTEGER(MSG_TEXT[39],3)  #,
                          MSG_AE_OWNER = MSG_TEXT[42] #,
                          MSG_AE_CLUB = MSG_TEXT[62] #,
    %                     FILLER
    %                 MSG_DELETE_ENTRY REDEFINES MSG_CREATE_RACE
                          MSG_DE_RACE_ID = INTEGER(MSG_TEXT[7],6) #,
                          MSG_DE_ID = MSG_TEXT[13] #,
    %                        FILLER
                      MSG_STATUS = MSG_TEXT[83] #;
      BOOLEAN B;
      PROCEDURE SEND_MSG;
       %  Send the message back to the originating station.  Do
       %  not specify an output agenda.  Make sure to test
       %  the result of the SEND statement.
       BEGIN       COMS_OUT.DESTCOUNT := 1;
       COMS_OUT.DESTINATIONDESG := COMS_IN.STATION;
       COMS_OUT.STATUSVALUE := 0;
       COMS_STATUS := SEND(COMS_OUT, TEXT_LEN, MSG_TEXT);
       IF NOT(COMS_STATUS = 0 OR COMS_STATUS = 92) THEN
         DISPLAY(öOnline Program SEND Err: ö !! STRING8(COMS_STATUS,*));
END SEND_MSG;
      PROCEDURE CREATE_RACE;
       %  Enter a new race record into the database.  Since the
       %  transaction is done in online mode, save the restart
       %  data in the conversation area only.  If the program aborts
       %  at BEGINTRANSACTION or ENDTRANSACTION, go back to the
       %  RECEIVE statement.
       BEGIN
       CREATE RACE-CALENDAR;
       PUT RACE-CALENDAR (RACE-NAME     := MSG_CR_NAME);
       PUT RACE-CALENDAR (RACE-ID       := MSG_CR_ID);
       PUT RACE-CALENDAR (RACE-DATE     := MSG_CR_DATE);
       PUT RACE-CALENDAR (RACE-TIME     := MSG_CR_TIME);
       PUT RACE-CALENDAR (RACE-LOCATION := MSG_CR_LOCATION);
       PUT RACE-CALENDAR (RACE-SPONSOR  := MSG_CR_SPONSOR);
         BEGINTRANSACTION COMS_IN          NOAUDIT RDS : B;
         IF B THEN
          BEGIN
          IF REAL(B.DMERROR) NEQ ABORT THEN
             DMTERMINATE(B);
          END
       ELSE
          BEGIN
          STORE RACE-CALENDAR : B;
          IF B THEN
             REPLACE MSG_STATUS BY "Store Error", " " FOR 19
          ELSE             REPLACE MSG_STATUS BY "Race Added", " " FOR 20;
          ENDTRANSACTION COMS_OUT AUDIT RDS : B;
          IF B THEN
             BEGIN
             IF REAL(B.DMERROR) NEQ ABORT THEN
                DMTERMINATE(B);
             END
          ELSE
             SEND_MSG;
          END;
       END CREATE_RACE;
         PROCEDURE ADD_ENTRY;
       %  Enter a boat in a race.  The restart requirements are the
       %  same as those for creating a race.
       BEGIN
       FIND RACE-SET AT RACE-ID = MSG_AE_RACE_ID: B;
       IF B THEN
          IF REAL(B.DMERROR) = NOTFOUND THEN
             BEGIN
             REPLACE MSG_STATUS BY "Race does not exist", " " FOR 11;
            SEND_MSG;
             END
          ELSE
             DMTERMINATE(B)
       ELSE
          BEGIN
          CREATE ENTRY;
          PUT ENTRY (ENTRY-BOAT-NAME   := MSG_AE_NAME);
          PUT ENTRY (ENTRY-BOAT-ID     := MSG_AE_ID);
          PUT ENTRY (ENTRY-BOAT-RATING := MSG_AE_RATING);
          PUT ENTRY (ENTRY-BOAT-OWNER  := MSG_AE_OWNER);
          PUT ENTRY (ENTRY-AFF-Y-CLUB  := MSG_AE_CLUB);
          PUT ENTRY (ENTRY-RACE-ID     := MSG_AE_RACE_ID);
            BEGINTRANSACTION COMS_IN          NOAUDIT RDS : B;
            IF NOT B THEN
             BEGIN
             STORE ENTRY: B;
             IF B THEN
                REPLACE MSG_STATUS BY "Store Error", " " FOR 19
             ELSE
                REPLACE MSG_STATUS BY "Boat Added", " " FOR 20;
             ENDTRANSACTION COMS_OUT AUDIT RDS : B;
             END;
          IF B THEN
             BEGIN
               IF REAL(B.DMERROR) NEQ ABORT THEN
                  DMTERMINATE(B);
             END
          ELSE
             SEND_MSG;
          END;
END ADD_ENTRY;
    PROCEDURE DELETE_ENTRY;
       %  Delete a boat from a race.  The restart requirements are
       %  the same as those for adding an entry.
       BEGIN
       LOCK ENTRY-RACE-SET AT
               ENTRY-RACE-ID = MSG_DE_RACE_ID AND
               ENTRY-BOAT-ID = MSG_DE_ID          : B;
```

```
   IF B THEN
      IF REAL(B.DMERROR) = NOTFOUND THEN
         BEGIN
         REPLACE MSG_STATUS BY "Boat Entry Not Found", " " FOR 10;
         SEND_MSG;
         END
         ELSE
         DMTERMINATE(B)
   ELSE
      BEGIN
      BEGINTRANSACTION COMS_IN          NOAUDIT RDS : B;
      IF NOT B THEN
         BEGIN            DELETE ENTRY : B;
         IF B THEN
            REPLACE MSG_STATUS BY "Found But Not Deleted", " " FOR 9
          ELSE
            REPLACE MSG_STATUS BY "Boat Deleted", " " FOR 18;
         ENDTRANSACTION COMS_OUT AUDIT RDS : B;
         END;
      IF B THEN
         IF REAL(B.DMERROR) NEQ ABORT THEN
            DMTERMINATE(B);
      SEND_MSG;
      END;
   END DELETE_ENTRY;
PROCEDURE CHECK_COMS_INPUT_ERRORS;
   %  Check for COMS control messages.
   BEGIN
      CASE COMS_STATUS OF
        BEGIN
    93: REPLACE MSG_STATUS BY "MSG Causes Abort, Do Not Retry";
        SEND_MSG;
    20:
   100:
   101:
   102: REPLACE MSG_STATUS BY "Error in STA Attach/Detachment";
        SEND_MSG;
     0:
    92:
    99:
    ELSE:; % A good message, recovery message, or EOT notification.
          END;
   IF COMS_IN.FUNCTIONINDEX < 0 THEN
      BEGIN
      REPLACE MSG_STATUS BY "Negative Function Code", " " FOR 8;
      SEND_MSG;
      END;
   END CHECK_COMS_INPUT_ERRORS;
  PROCEDURE CLOSE_DOWN;
      %  Close the database.
   BEGIN
   CLOSE SAILDB;
```

```
        END CLOSE_DOWN;
       PROCEDURE PROCESS_TRANSACTION;
        %  Since the transaction type is based on the
        %  function index, make sure it is within
        %  range.
        BEGIN
        CASE COMS_IN.FUNCTIONINDEX OF
             BEGIN
        ELSE:BEGIN
             REPLACE MSG_STATUS BY
                     "Invalid Trans Code", " " FOR 12;
             SEND_MSG;
             END;
        1:   CREATE_RACE;
        2:   ADD_ENTRY;
        3:   DELETE_ENTRY;
             END;
        END PROCESS_TRANSACTION;
     PROCEDURE PROCESS_COMS_INPUT;
        %  Gets the next message from COMS.  If the status
        %  returned is an EOF_NOTICE, go to EOT, else make sure
        %  that it is a valid message before processing it.
        BEGIN
       REAL COMS_INPUT_STATUS;
        REPLACE MSG_TEXT BY " " FOR TEXT_LEN;
       COMS_INPUT_STATUS :=COMS_STATUS := RECEIVE(COMS_IN, MSG_TEXT);
        IF COMS_STATUS NEQ GOEOT THEN
           BEGIN
           CHECK_COMS_INPUT_ERRORS;
           IF (COMS_INPUT_STATUS = 0 OR COMS_INPUT_STATUS = 92) AND
             COMS_IN.FUNCTIONINDEX >= 0 THEN
             PROCESS_TRANSACTION;
           END;
           END PROCESS_COMS_INPUT;
%========================================================
%       OPEN UPDATE SAILDB: B;
   IF B THEN
       DMTERMINATE(B);
    ENABLE(COMS_IN,"ONLINE");
      CREATE RDS;
      DO
      PROCESS_COMS_INPUT
    UNTIL COMS_STATUS = GOEOT;
      CLOSE_DOWN;
    END.
```

Section 4 Using the Data Management System II (DMSII) Interface

An interface to the Data Management System II (DMSII) is provided in the BDMSALGOL language. BDMSALGOL is based on Unisys Extended ALGOL and contains extensions that enable a programmer to declare and use databases. The extensions to ALGOL that make up the BDMSALGOL language are described in this chapter. These extensions provide the following capabilities:

- Invoking a database
- Manipulating data through data management statements
- Using database items through a mapping syntax
- Processing exceptions

Programs written in the BDMSALGOL language must be compiled with the BDMSALGOL compiler. Typically, this compiler is titled “SYSTEM/BDMSALGOL”.

Refer to the *DMSII Application Program Interfaces Programming Guide* for a discussion of DMSII programming issues, such as audit and recovery. Consult the *DMSII Data and Structure Language (DASDL) Programming Reference Manual* for detailed information on DASDL.

DMSII and Semantic Information Manager (SIM) databases can be accessed and used in the same program. Each database must be invoked, manipulated, and processed with its own extensions. Use DMSII and BDMSALGOL extensions for DMSII databases. Use SIM extensions for SIM databases.

You can also use DMSII with other products described in this volume, such as Communications Management System (COMS), Advanced Data Dictionary System (ADDS), and Transaction Processing System (TPS).

Additional information relating to DMSII extensions is included in Section 3, “Using Communications Management System (COMS) Features,” Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions,” Section 5, “Using DMSII Transaction Processing System (TPS) Extensions,” and Section 7, “Using the Semantic Information Manager (SIM) Interface.”

Invoking a DMSII Database

Invoking a database involves both database declarations and database equations.

Declaring a Database

<database declaration>

```
DATABASE — <database reference>————————————————————————————————————┤
```

<database reference>

```
─┬──────────────────┬─┬───────────────────────────────┬─ <database name>→
 └─ <internal name> = ┘ └─ <logical database name> OF ─┘

→─┬────────────────────────────────────────────────────┬───────────────→
  └─ ( — TITLE = '' — <database title> — '' — )─────────┘

→─┬─────────────────────────────────────────┬─────────────────────────┤
  │      ┌←─────────── , ────────────┐      │
  └─ : ──┴─┬─ <data set reference> ─┬─┴──────┘
           └─ <set reference> ──────┘
```

<internal name>

```
 — <BDMS identifier>——————————————————————————————————————————————————┤
```

<logical database name>

```
 — <BDMS identifier>——————————————————————————————————————————————————┤
```

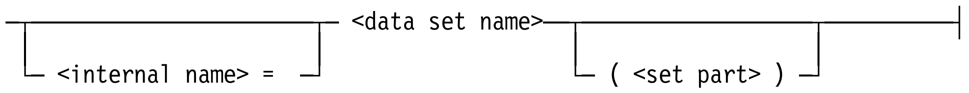
<database name>

```
 — <BDMS identifier>——————————————————————————————————————————————————┤
```

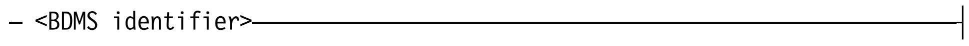
<database title>

A properly formed file title constant (as defined in the *Work Flow Language (WFL) Programming Reference Manual*) that has only one node; that is, a file title constant that does not contain any slashes (/).

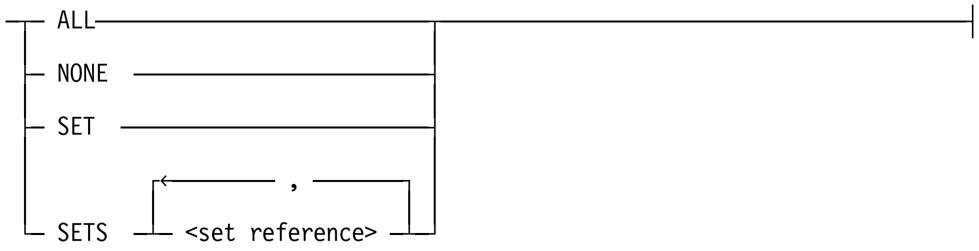
<data set reference>



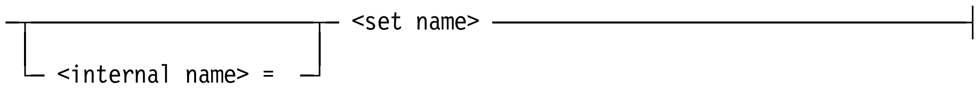
<data set name>



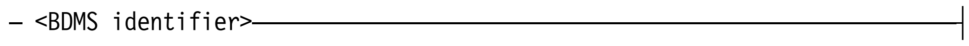
<set part>



<set reference>



<set name>



Explanation

Like all variables, a database must be declared in a BDMSALGOL program before it is referenced. However, a DATABASE declaration is unlike other declarations in that it is actually an invocation of a database that has already been fully described and declared in the Data and Structure Definition Language (DASDL).

Two different databases can be updated in the same program only if they are the same physical database.

If the compiler control options LIST and LISTDB are both TRUE, all invoked structures, together with the record formats, item and key descriptions, database titles, and other pertinent information, are written on the program listing. When database application programs are being developed, the LISTDB option should be used, and the resulting information should be studied carefully.

Additional information relating to the LIST and LISTDB options is included under "BDMSALGOL Compiler Control Options" in this section.

A DATABASE declaration declares a database and specifies which database or which parts of a database are to be invoked. If no data set reference parts and no set reference parts are specified in a DATABASE declaration, then all data sets and all sets for each data set are implicitly invoked.

The internal name construct assigns an internal name by which a database, data set, set, or subset is known within the program. When an internal name is specified, all subsequent references to the structure must use this internal name.

A database, data set, set, or subset can be invoked more than once; however, the external name (the name in the description file) can be used to reference only one invocation of a structure. Internal names must be used to provide unique names for all other invocations of a structure. The default internal name of a structure is its external name.

By using the internal names in the data set reference or the set reference constructs, multiple record areas or set paths can be established. Thus, several records of a single data set can be manipulated simultaneously.

The logical database name construct enables the program to reference a logical database. A program can invoke structures selectively from a logical database, or it can invoke the entire logical database. Selective invocations are specified in the same manner as for physical databases; however, the choice of structures is limited to those structures included in the logical database.

The database name form gives the external name of the database to be invoked.

The database title construct is an alphanumeric string. A usercode, if any, is the usercode of the control file. The single node of the title is the directory node under which the database files are stored. The family name, if any, is the family name of the control file. The default database title is the external name of the database plus the control file usercode and family name, if any, from the description file. When opening the database, the Master Control Program (MCP) builds the control file title from the database title specified in the declaration. See the *DMSII DASDL Programming Reference Manual* for a discussion of control files and description files.

This title equation is used only at run time, and cannot be used at compile time to specify the title of the database description file. The primary use of the database title construct is for modeling. See the *DMSII DASDL Programming Reference Manual* for a description of modeling.

The data set reference construct specifies a particular data set from the declared database. If a data set reference is used, only the specified structures are invoked. A data set reference must be used to invoke a disjoint data set.

The data set name construct gives the external name of the data set to be invoked.

The set part construct invokes specific sets from the data set declared in the data set reference that contains it. If the set part construct is omitted, all sets are implicitly invoked. If the set part construct is used, all sets (ALL), no sets (NONE), or only the specified sets are invoked.

The set reference construct establishes a set that is not implicitly associated with any particular record area. To load a record area using the set name specified in a set reference, the "<data set> VIA" form of the selection expression must be used.

The set name construct gives the external name of the set to be invoked.

Only disjoint structures can be explicitly invoked. When a master data set is invoked (either implicitly or explicitly), its embedded data set, sets, and subsets are always implicitly invoked. When a data set containing an embedded set associated with a disjoint data set is invoked, or a data set containing a link to another disjoint data set is invoked, then a path is established. However, the disjoint data set must be invoked if it is to be used.

Multiple invocations of a structure provide multiple record areas or set paths, or both, so that several records of a single data set can be manipulated simultaneously. Selecting only needed structures for UPDATE and INQUIRY provides better use of system resources.

If remaps are declared in DASDL, they are invoked in the same manner as conventional data sets.

Additional information relating to the BDMS identifier construct is included under "BDMS Identifier Construct" in this section.

Example: Simple Database

The following examples apply to the database DB described by the following DASDL description:

```
D DATA SET (
    K NUMBER (6);
    R NUMBER (5);
    );
S1 SET OF D KEY K;
S2 SET OF D KEY R;


DATABASE DB: D
```

This declaration establishes one current record area for the data set D, one path for the set S1 of data set D, and one path for the set S2 of data set D. The statements “FIND S1”, “MODIFY S1”, “FIND S2”, and “MODIFY S2” automatically load the data into the D record area.

```
DATABASE DB: D, X=D (NONE)
```

This declaration establishes two current record areas (D and X) and two paths (S1 and S2). The sets S1 and S2 are implicitly associated with the D record area. The set part NONE prevents a set from being associated with X. Thus, the statements “FIND S1” and “FIND S2” load the D record area. The statements “FIND X VIA S1” and “FIND X VIA S2” must be executed to load the X record area using a set.

```
DATABASE DB: D, X=D
```

This declaration shows how multiple current record areas and multiple current paths can be established. The statement “FIND S1 OF D” loads the D record area without disturbing the path S1 OF X, and the statement “FIND S1 OF X” loads the X record area without disturbing the path S1 OF D. Qualification of S1 is necessary to distinguish the paths.

```
DATABASE DB: D (SET S1), X=D (SET S1), Y=D (NONE)
```

This declaration shows how more current record areas than paths can be established. Three record areas (D, X, and Y) are established, but only two paths (S1 OF D and S1 OF X) are established. The program must execute the statement “FIND Y VIA S1 OF D”, “FIND Y VIA S1 OF X”, or “FIND Y” to load the Y record area.

```
DATABASE DB: X=D (SET S1), Y=D (SET T=S1)
```

This declaration explicitly associates a set with a given work area. The statement “FIND S1” loads the X record area, and the statement “FIND T” loads the Y record area. S1 and T both use the same key.

```
DATABASE DB: D, SY=S1
```

This declaration shows how a set reference can be used to establish a set that is not implicitly associated with any particular record area. The statement “FIND D VIA SY” must be executed to load a record area using the set S1.

Example: Invoking Disjoint Data Sets

The following example shows when a data set reference must be used to invoke disjoint data sets. The database DB is described by the following DASDL description:

```
F DATA SET (
   FI NUMBER (4);
   );
E DATA SET (
   EK NUMBER (8);
   );
D DATA SET (
   A NUMBER (6);
   SE SET OF E KEY EK;
   LINK REFERENCE TO F;
   );
```

If data set references are not specified to invoke E and F, as in the declaration

```
DATABASE DB: D
```

the paths are established by invoking the embedded set SE and the link item LINK. However, these paths cannot be used unless data set references for E and F are specified to establish record areas associated with these paths, as in the declaration

```
DATABASE DB: D,E,F
```

Example: Invoking a Logical Database

In this example, the database EXAMPLEDB, shown on the following page, is described by the DASDL description given below:

```
D1 DATA SET (
   A REAL;
   B NUMBER (5);
   C ALPHA (10);
   );
S1A SET OF D1 KEY IS A;
S1B SET OF D1 KEY IS (A,B,C);
D2 DATA SET (
   X FIELD (8);
   Y NUMBER (2);
   Z REAL;
   E DATA SET (
      V1 REAL;
      V2 ALPHA (2);
      );
   SE SET OF E KEY IS V1;
   );
S2A SET OF D2 KEY IS X;
S2B SET OF D2 KEY IS (X,Y,Z);
LDB1 DATABASE (D1(NONE), D2(SET S=S2A));
LDB2 DATABASE (D1(SET S1=S1B), D2(SET S2=S2B));
LDB3 DATABASE (D=D2);
```

The following BDMSALGOL program invokes the logical database LDB1 of EXAMPLEDB. Data sets D1 and D2 are available to the program; however, none of the sets associated with D1 are available. The only set associated with D2 that is available is set S2A, which appears as set S. The output produced by the LISTDB compiler control option is shown with the program.

```
$ SET LIST LISTDB
BEGIN
   DATABASE LDB1 OF EXAMPLEDB;
*DATABASE TITLE: EXAMPLEDB ON DISK
*01 D1: DATA SET (#2)
*      INVOKED SETS:
*      RECORD ITEMS:
*02       REAL A
*02       INTEGER B: NUMBER (5)
*02       STRING C: ALPHA (10)
*01 D2: DATA SET (#5)
*      INVOKED SETS:
*         S (#8, AUTOMATIC), KEY = X
*      RECORD ITEMS:
*02       REAL X: FIELD (8)
*02       INTEGER Y: NUMBER (2)
*02       REAL Z
*02       E: DATA SET (#6)
*            INVOKED SETS:
*               SE (#7, AUTOMATIC), KEY = V1
*            RECORD ITEMS:
*03            REAL V1
*03            STRING V2: ALPHA (2)
*DESCRIPTION TIMESTAMP: 06/09/82 @ 17:30:34
END.
```

Database Equation Operations

<database attribute assignment statement>

```
 — <string-valued database attribute> — := — <string expression> ——————————┤
```

<string-valued database attribute>

```
 — <internal name> — . — TITLE  ———————————————————————————————————————————┤
```

Explanation

The term “database equation” refers to three separate operations:

- Specification of database titles during compilation.
- Work Flow Language (WFL) database equation to override compiled-in titles. (For more information, refer to the *DMSII Application Program Interfaces Programming Guide* for the WFL syntax.)
- Run-time manipulation of database titles.

To take advantage of the reentrance capability of the Accessroutines, the user must be able to specify the title of a database at run time. Database equation enables the database title to be specified at run time and enables access to databases that are stored under other usercodes and on families that are not visible to a task. For further information about the Accessroutines, consult the *DMSII Application Program Interfaces Programming Guide.*

Database equation is operationally similar to file equation. WFL database equation overrides the specification of a database title in the DATABASE declaration, and run-time modification of a database title overrides both WFL database equation and the DATABASE declaration. However, database equation differs from file equation in that a run-time error results if a BDMSALGOL program attempts to assign a value to or examine the TITLE attribute of a database while it is open. For an explanation of the TITLE database attribute, refer to “DATABASE Declaration” in this section.

The string expression must evaluate to a string in the form of a database title.

The string-valued database attribute construct can be used anywhere a string expression is valid.

Database titles never end with a period, and a replace pointer-valued attribute statement is not valid for making assignments to database titles.

Note: *BDMSALGOL programs employing database equation must be compiled with a BDMSALGOL compiler with a release level later than Mark 3.2.*

Additional information relating to the <internal name> construct is included under “Declaring a Database” in this section.

Example

In this example, the first BDMS OPEN statement opens the database with the title LIVEDB, whose data and control files are stored under the user's directory. The second OPEN statement invokes the database TESTDB, whose files are stored on TESTPACK under the usercode UC.

```
BEGIN
STRING S;
DATABASE MYDB (TITLE="LIVEDB");
OPEN UPDATE MYDB;
 ...
CLOSE MYDB;
MYDB.TITLE := "(UC)TESTDB ON TESTPACK";
OPEN UPDATE MYDB;
 ...
CLOSE MYDB;
S := TAKE(MYDB.TITLE,5);
 ...
END.
```

BDMSALGOL Basic Language Constructs

The constructs described on the following pages are used within the DMSII "DATABASE" declaration and in DMSII data management statements and functions. The descriptions cover the following topics:

- The conventions for naming databases, data sets, sets, items, and so forth
- Input mapping and output mapping
- Selection expressions

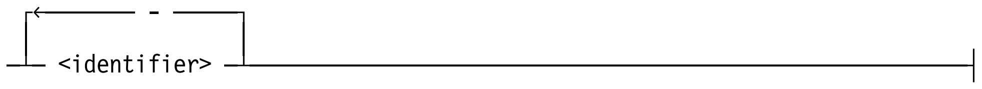
BDMS Naming and Qualification Conventions

Naming conventions in DASDL for databases and their components follow COBOL rules; that is, names can contain hyphens, and some item and structure names can require qualification. Although both of these conventions contradict normal ALGOL naming rules, they must be permitted in programs that declare and use databases.

BDMS Identifier Construct

The identifier of a database, data set, set, item, and so on is in the form of a BDMS identifier.

<BDMS identifier>



Explanation

The BDMS identifier construct must be fewer than 64 characters long.

Examples

If a database is described in DASDL by the following:

```
D-S DATA SET (
   A-1 NUMBER (5);
   A-2 NUMBER (10);
   );
```

then in a BDMSALGOL program, the data set D-S and the items A-1 and A-2 can be referenced as in the following examples:

```
INTEGER I;
GET D-S (I := A-1);
PUT D-S (A-2 := I);
```

Construct for Identifiers of Occurring Items

If an item is declared in the DASDL description to have an OCCURS clause, then its identifier must be subscripted to denote which of its occurrences is to be used.

<subscripted BDMS identifier>

```
                           ┌←────────── , ─────────┐
                           │                       │
— <BDMS identifier> — [ ───┴─ <arithmetic expression> ─┴─ ] ─────────────┤
```

Explanation

The leftmost arithmetic expression denotes the subscript of the outermost OCCURS clause that affects the item, the next arithmetic expression to the right denotes the subscript of the next outermost OCCURS clause, and so on.

Examples

If items A and B are described in DASDL as follows:

```
DS DATA SET (
   G GROUP (
      A ALPHA (10);
      B NUMBER (4) OCCURS 3 TIMES;
      )
   OCCURS 2 TIMES;
   );
```

there are two occurrences of A, denoted

```
A[1]          A[2]
```

and there are six occurrences of B, denoted

```
B[1,1]        B[2,1]
B[1,2]        B[2,2]
B[1,3]        B[2,3]
```

Qualification of Database Items

Database item names need not be unique within a database. Qualification is used to distinguish between database items with the same names.

<qualification>

```
 ┌←──────── OF ──────┐
 │                   │
─┴─ <BDMS identifier> ─┴──────────────────────────────────────────┤
```

Explanation

An item name can be qualified by the name of any structure that physically contains the item. Any number of qualification names desired can be used, provided that the result is unique. If improper or insufficient qualification is used, a syntax error is given.

A set name can be qualified by the name of the data set it spans.

A group name can be used to qualify an item it contains.

Qualification need not be used if the unqualified name is unique. Qualification must be used whenever there is ambiguity. A variable name can be declared with the same name as a database item in BDMSALGOL without requiring qualification of the item name.

Examples

If a database is described in DASDL as follows:

```
DS1 DATA SET (
   N NUMBER (4) OCCURS 4 TIMES;
   );
DS2 DATA SET (
   N NUMBER (4) OCCURS 4 TIMES;
   );
```

then the following BDMSALGOL statements indicate how qualification is used to distinguish between the two data items named N.

```
SET N OF DS1 TO NULL;
SET N OF DS2 TO NULL;

SET N(1) OF DS1 TO NULL;
SET N(1) OF DS2 TO NULL;
```

Referencing Database Items

The record area (user work area) is not directly accessible to a BDMSALGOL program. Instead, an explicit mapping between database data items and program variables must be specified whenever access to those items is desired.

Mappings specify the source and destination of data to be transferred into or out of a user work area. Mappings are of two kinds: input mappings and output mappings.

Example

If a database is described in DASDL by the following:

```
D1 DATA SET (
   A NUMBER (5);
   X NUMBER (5) OCCURS 3 TIMES;
   );
```

then the items of data set D1 can be referenced in the following ways:

```
INTEGER B,Y1,Y2,Y3;
% The following statement transfers the value of database item
%  A to the locally declared integer B.
GET D1 (B := A);

% The following statement transfers the value of locally
% declared integer B to the work area for D1.
PUT D1 (A := B);
% The following statement transfers the values of all three
% occurrences of X into Y1, Y2, and Y3.
GET D1 (Y1 := X[1];
        Y2 := X[2];
        Y3 := X[3]);

% The following statement transfers the values of locally
% declared integers Y1, Y2, and Y3 into the three occurrences
% of database item X.
PUT D1 (X[1] := Y1,
        X[2] := Y2,
        X[3] := Y3);
```

Input Mapping Used with Retrieval Statements

<input mapping>

```
 ┌←───────── ; ──────┐
 │┌←──────── , ──────┤
─┴┴─ <input assignment> ─┴──────────────────────────────────────┤
```

<input assignment>

```
─┬─ <arithmetic variable> ─ := ─┬─ <count item name> ─────────┬─┬────────────┤
 │                              ├─ <field item name> ─────────┤ │
 │                              ├─ <numeric item name> ───────┤ │
 │                              ├─ <population item name> ────┤ │
 │                              ├─ <real item name> ──────────┤ │
 │                              └─ <record type item name> ───┘ │
 ├─ <Boolean variable> ─ := ─ <Boolean item name> ──────────────┤
 └─ <pointer variable> ─ := ─┬─ <alpha item name> ───┬──────────┘
                             ├─ <group item name> ───┤
                             └─ <numeric item name> ─┘
```

<alpha item name>
<Boolean item name>
<count item name>
<field item name>
<group item name>
<numeric item name>
<population item name>
<real item name>
<record type item name>

```
─┬─<BDMS identifier>─────────────┬──────────────────────────────────┤
 └─ <subscripted BDMS identifier> ─┘
```

Explanation

Input mappings can be used with the retrieve statements DELETE, FIND, GET, BDMS LOCK, and MODIFY. Input mappings transfer the value of a DASDL-declared data item to a program variable. If the data item is an occurring item (that is, if the item is declared in DASDL with an OCCURS clause), it must be subscripted appropriately.Additional information relating to the BDMS identifier construct is included under "BDMS Identifier

Construct" in this section. Information related to the subscripted BDMS identifier construct is included under "Construct for Identifiers of Occurring Items" in this section.

An arithmetic variable can be an integer, real, or a double simple or subscripted variable. A Boolean variable can be a subscripted or Boolean simple variable. A pointer variable can be a pointer identifier or an element of a character array.

```
<arithmetic variable> := <field item name>
```

If the field item is defined to contain N bits, then N bits are stored right-justified in the arithmetic variable. All other bits are set to zero.

```
<arithmetic variable> := <numeric item name>
<arithmetic variable> := <real item name>
```

The numeric item or real item is converted into a binary value with a scale factor of zero (its true value). The value is stored in the arithmetic variable as in a normal arithmetic assignment; that is, it is converted to an integer or extended, if necessary. An error and termination results if it is not possible to convert the item to an integer, as in normal ALGOL arithmetic assignments.

```
<arithmetic variable> := <count item name>
<arithmetic variable> := <population item name>
<arithmetic variable> := <record type item name>
```

The value of the count item, population item, or record type item is placed in the arithmetic variable. Use of a count item, population item, or record type item enables read-only access to the particular field. Those items cannot be changed directly. They are accessed only through input mappings, and cannot be used in output mappings.

```
<Boolean variable> := <Boolean item name>
```

The Boolean variable is assigned the truth value (the value of bit 0) of the Boolean item. Bits 1 through 47 of the Boolean variable are set to zero.

```
<pointer variable> := <alpha item name>
<pointer variable> := <group item name>
```

If the alpha item or group item is defined to contain N EBCDIC characters, then N characters are transferred to the location pointed to by the pointer variable. A fault results if one of the following conditions is satisfied:

- The pointer is uninitialized.
- The pointer is not an EBCDIC (8-bit) pointer.
- Fewer than N character positions remain in the referenced array.

A group item is treated as if it were an alpha item; all subordinate data items are transferred without change.

```
<pointer variable> := <numeric item name>
```

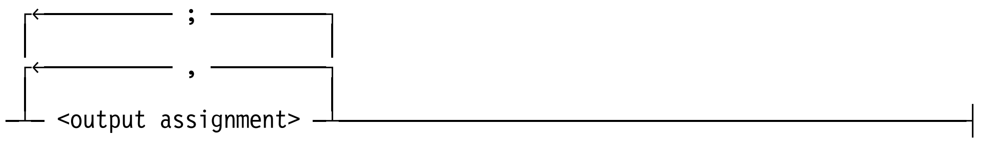
This assignment takes advantage of the fact that a numeric item is maintained as a hexadecimal string. If the numeric item is defined to contain N digits (including the sign digit, if specified), the N hexadecimal characters are transferred to the location pointed to by the pointer variable. A fault results if one of the following conditions is satisfied:

- The pointer is uninitialized.
- The pointer is not a hexadecimal (4-bit) pointer.
- Fewer than N hexadecimal character positions remain in the referenced array.

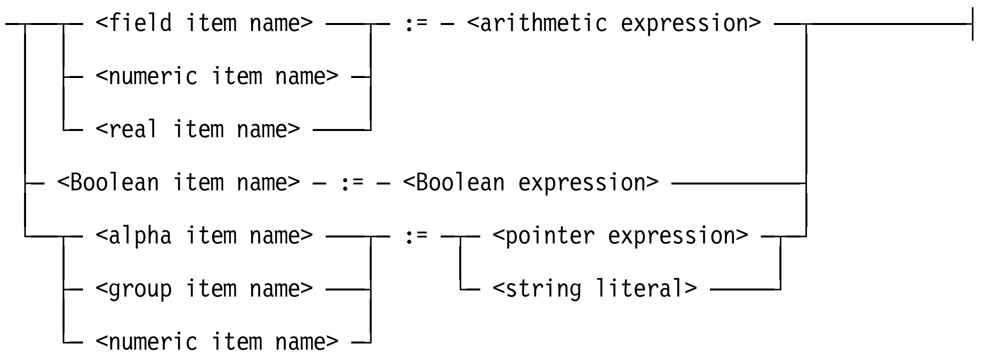
For more information concerning the arithmetic variable, Boolean variable, and pointer variable constructs, refer to Volume 1.

Output Mapping Used with Storage Statements

<output mapping>



<output assignment>



Explanation

Output mappings can be used with the storage statements PUT and STORE. Output mappings transfer the value of a program variable or expression to a DASDL-declared data item. If the data item is an occurring item (that is, if the item is declared in DASDL with an OCCURS clause), it must be subscripted appropriately.

An arithmetic expression used in an output mapping can be single precision or double precision.

```
<field item name> := <arithmetic expression>
```

If the field item is defined to contain N bits, then the N rightmost bits of the value of the arithmetic expression are assigned, unaltered, to the field item. Care should be taken if the arithmetic value is real or double precision (that is, not integer) because the value might be normalized, in which case the N rightmost bits would not contain the value.

```
<numeric item name> := <arithmetic expression>
<real item name> := <arithmetic expression>
```

The value of the arithmetic expression is scaled appropriately and assigned to the numeric item or real item. If the numeric item or real item is unsigned, the absolute value of the arithmetic expression is used.

```
<Boolean item name> := <Boolean expression>
```

The truth value (the value of bit 0) of the Boolean expression is assigned to the Boolean item. Bits 1 through 47 of the value of the Boolean expression are ignored.

```
<alpha item name> := <pointer expression>
<group item name> := <pointer expression>
```

If the alpha item or group item is defined to contain N EBCDIC characters, then N characters are transferred from the location pointed to by the pointer expression to the alpha or group item. A fault results if any of the following conditions is satisfied:

- The value of the pointer expression is an uninitialized pointer.
- The value of the pointer expression is not an EBCDIC (8-bit) pointer.
- Fewer than N character positions remain in the referenced array.

```
<numeric item name> := <pointer expression>
```

This mapping takes advantage of the fact that a numeric item is maintained as a hexadecimal string. If the numeric item is defined to contain N digits (including the sign digit, if specified), then N hexadecimal characters are transferred to the numeric item from the location pointed to by the pointer expression. The user is responsible for ensuring that the string is a valid representation of the item declared in DASDL; that is, the proper sign and numeric characters, in the proper format, must be used.

A fault results if any of the following conditions is true:

- The value of the pointer expression is an uninitialized pointer.
- The value of the pointer expression is not a hexadecimal (4-bit) pointer.
- Fewer than N hexadecimal character positions remain in the referenced array.

```
<alpha item name> := <string literal>
<group item name> := <string literal>
```

The string literal is transferred to the alpha item or group item. The string literal must be EBCDIC, or a syntax error results. If the string literal is shorter than the alpha item or group item, it is extended with blank fill characters on the right. If the string literal is longer than the alpha item or group item, the excess characters on the right are truncated.

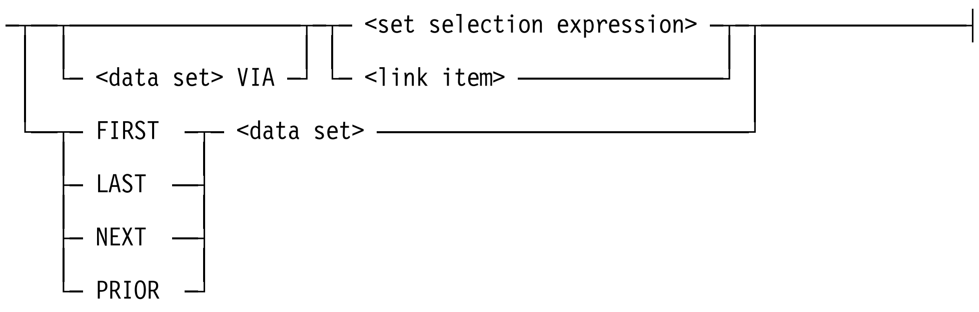
```
<numeric item name> := <string literal>
```

The string literal is transferred to the numeric item. The string literal must be a hexadecimal string and must contain the exact number of characters for the numeric item or a syntax error results. The user is responsible for ensuring that the string literal is a valid representation of the numeric item.

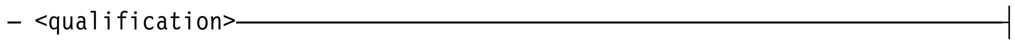
Additional information relating to the field item name, numeric item name, real item name, Boolean item name, alpha item name, and group item name constructs is included under “Input Mapping Used with Retrieval Statements” in this section.

Selecting a Record in a Data Set

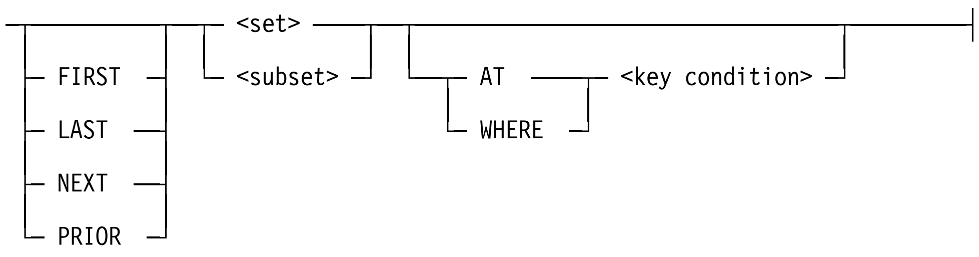
<selection expression>



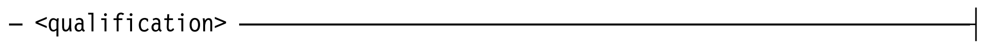
<data set>



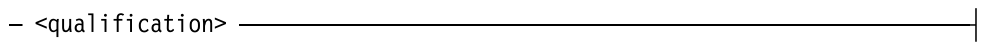
<set selection expression>



<set>



<subset>



<key condition>

```
  ┌←─────────────────┬─ AND ─┬──────────┐
  │                  └─ OR ──┘          │
──┴─┬─ <numeric relation> ─────────────┬┴─────────────────────────────┤
    ├─ <alphanumeric relation> ────────┤
    └──────────┬─ ( ─ <key condition> ─ ) ─┘
       └─ NOT ─┘
```

<numeric relation>

```
─┬─ <numeric item identifier> ─┬─ <relational operator> ──────────────→
 ├─ <field item identifier> ───┤
 └─ <real item identifier> ────┘

→─┬─ <arithmetic expression> ─┬──────────────────────────────────────┤
  └─ <pointer expression> ────┘
```

<numeric item identifier>
<field item identifier>
<real item identifier>

```
─ <BDMS identifier> ─────────────────────────────────────────────────┤
```

<alphanumeric relation>

```
─ <alpha item identifier> ─ <relational operator> ─┬─ <constant string expression>┤
                                                   └─ <pointer expression>─┘
```

<alpha item identifier>

```
─ <BDMS identifier> ─────────────────────────────────────────────────┤
```

<link item>

```
─ <qualification> ───────────────────────────────────────────────────┤
```

Explanation

A selection expression is used in DELETE, FIND, BDMS LOCK, and MODIFY statements to identify a particular record in a data set.

A set selection expression selects the record to which the set path refers. A NOTFOUND exception is returned if the record has been deleted or if the path does not refer to a valid current record.

The construct “data set VIA” identifies the record area and current path to be affected if the desired record is found. This option is used for link items and for sets that are not implicitly associated with the data set.

The link item form is used to specify a link item defined in the DASDL description. The record to which the link item refers is selected. An exception is returned if the link item is NULL.

The data set form is used to select the record to which the data set path refers. A NOTFOUND exception is returned if the record has been deleted or if the path does not refer to a valid current record.

The word “FIRST” selects the first record in the specified data set, set, or subset. If a key condition is also specified, the first record of the specified set or subset that satisfies the key condition is selected. FIRST is assumed by default.

The word “LAST” selects the last record in the specified data set, set, or subset. If a key condition is also specified, the last record of the specified set or subset that satisfies the key condition is selected.

The word “NEXT” selects the next record relative to either the set path (if a set or subset is specified) or the data set path (if a data set is specified). If a key condition is also specified, the next record (relative to the current path) of the specified set or subset that satisfies the key condition is selected.

The word “PRIOR” selects the prior record relative to either the set path (if a set or subset is specified) or the data set path (if a data set is specified). If a key condition is also specified, the prior record (relative to the current path) of the specified set or subset that satisfies the key condition is selected.

In a set selection expression, the set or subset construct selects the record to which the set or subset path refers. A NOTFOUND exception is returned if the record has been deleted or if the path does not refer to a valid current record.

The words “AT” or “WHERE” indicate that a key condition follows. AT and WHERE are synonyms.

A key condition specifies values used to locate specific records in a data set referenced by a particular set or subset. If the name of a data item specified in a key condition is not unique, the compiler provides implicit qualification through the set or subset of the set selection expression. Although not necessary, qualification of the item name by the name of the data set that contains the item is permitted; however, the compiler handles this qualification as documentation only.

The expressions that appear in a key condition cannot contain any transaction item references.

A numeric relation specifies a particular numeric, field, or real item and compares it to the value of an arithmetic expression or a pointer expression. The pointer expression must evaluate to a hexadecimal pointer.

An alphanumeric relation specifies a particular alpha item and compares it to the value of a constant string expression or a pointer expression. The pointer expression must evaluate to an EBCDIC pointer. The constant string expression must be an EBCDIC string.

Additional information relating to the BDMS identifier construct is included under "BDMS Identifier Construct" in this section. Information on the qualification construct is included under "Qualification of Database Items" in this section.

For more information concerning constant string expression and relational operators, refer to Volume 1.

Examples

These examples use the database described in DASDL by the following:

```
D DATA SET (
  A ALPHA (3);
  N NUMBER (5);
  );
S SET OF D KEY IS N, DATA A;


LOCK S WHERE N NEQ 10
```

This LOCK statement acts upon the first S where the value of N is not equal to 10.

```
FIND S AT A = "ABC" AND (N = 50 OR N = 90)
```

This statement locates the first S where A is equal to the string "ABC" and either N is equal to 50 or N is equal to 90.

BDMSALGOL Statements

The following data management statements enable a BDMSALGOL program to use and manipulate the data in a database.

ABORTTRANSACTION	GET
ASSIGN	INSERT
BEGINTRANSACTION	BDMS LOCK
CANCELTRPOINT	MODIFY
BDMS CLOSE	BDMS OPEN
CREATE	PUT
DELETE	RECREATE
DMTERMINATE	REMOVE
ENDTRANSACTION	SAVETRPOINT
FIND	SECURE
BDMS FREE	BDMS SET
GENERATE	STORE

Note that the BEGINTRANSACTION statement initiates a transaction which is concluded by an ENDTRANSACTION statement. A transaction is a series of changes to the database which are considered to be an indivisible logical change. A transaction is the basic unit effecting change in the DMSII database.

Transaction state is that period of execution time when the DMSII database can be updated. Every update program of an audited database must enter transaction state in order to perform any data record update statements. Transactions are applied but not actually committed until the ENDTRANSACTION statement is executed.

COMS and DMSII can be used together to provide a recoverable transaction system. Consult Section 3, “Using Communications Management System (COMS) Features” for more information and for the needed syntax.

ABORTTRANSACTION Statement

<aborttransaction statement>

```
— ABORTTRANSACTION  ——┬————————————————————┬— <restart data set> ————————→
                      │                    │
                      └— <COMS header ID> —┘

→—┬————————————————————————┬———————————————————————————————————————————————┤
  │                        │
  └— <exception handling> —┘
```

Explanation

The ABORTTRANSACTION statement backs out all updates that occurred during a transaction and takes a program out of the transaction state. The DMSII database is returned to the point before the BEGINTRANSACTION statement (which initiated the transaction) was executed.

The ABORTTRANSACTION statement is equivalent to performing a CANCELTRPOINT statement followed by an END TRANSACTION statement.

The COMS header ID construct identifies the COMS Output Header. If the system fails during transaction state, COMS resubmits the message when the program is reexecuted.

The restart data set construct identifies the data set containing the restart records that application programs can access to recover database information after a system failure.

Additional information relating to the <exception handling> construct is included under "Database Status Word" in this section.

Example

In this example, the ABORTTRANSACTION statement notifies DMSII of the abort, and assigns the result of the abort call to the variable DMSTATUS. All transactions are backed out to the BEGINTRANSACTION statement, and the program is taken out of transaction state.

```
BEGINTRANSACTION RSTDS;
  ...
  SAVETRPOINT (1);
  ...
ABORTTRANSACTION RSTDS : DMSTATUS;
```

ASSIGN Statement

<assign statement>

```
– ASSIGN ──┬─ <data set> ──┬─ TO  – <link item> ──┬────────────────────────┬──────────┤
           │               │                      │                        │
           ├─ NULL ────────┤                      └─ <exception handling> ─┘
           │               │
           └─ <link item> ─┘
```

Explanation

The ASSIGN statement establishes a link from one record in a data set to another record of the same or a different data set. It assigns either the value of the current record in a data set or the value in a link item to another link item. The value of the second link item, called the target link item, then enables the system to locate the record in the referenced data set.

The ASSIGN statement is effective immediately; therefore, the record containing the target link item does not need to be stored unless data items of this record have been modified.

The data set must be declared in DASDL as the object data set of the target link item. A value that points to the current record in the data set is assigned to that link item.

If the data set form is used, the current path of the specified data set must be valid, but the record need not be locked. If the data set path is not valid, an exception occurs.

If the word "NULL" is used, the relationship between records is severed by assigning a NULL value to the target link item. If that link item is already NULL, this option is ignored. A FIND, BDMS LOCK, or MODIFY statement on a NULL link item results in an exception.

If the ASSIGN statement specifies two link items, the value of the first link item is assigned to the target link item. The first link item must be declared in DASDL to have the same object data set as the target link item and be the same type of link (counted link, self-correcting link, symbolic link, unprotected link, or verified link). If the link items are counted links, the count item is automatically updated, even if the record that is referenced is locked by another program.

The current path of the data set containing the first link must be valid, but the record need not be locked. If the data set path is not valid, an exception occurs.

After the ASSIGN statement has executed, the target link item points to either the current record in the specified data set or to the record pointed to by the first link item.

The current path of the data set containing the target link item must be valid, and the record must be locked; otherwise, an exception occurs.

If the target link item references a disjoint data set, then that link item can point to any record in the data set. If the target link item references an embedded data set, then only certain records in the data set can be referenced. In this case, the record being referenced must be owned by the record containing the target link item or by an ancestor of the record containing this link item. (An ancestor is the owner of the record, the owner of the owner, and so forth.)

If an exception is returned, the ASSIGN statement is not completed, and a NULL value is assigned to the target link item.

Additional information relating to the data set and link item constructs is included under "Selecting a Record in a Data Set" in this section. Information on the exception handling construct is included under "Database Status Word" in this section.

Examples

If the database EXAMPLEDB is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   L IS IN E VERIFY ON N;
   );
S SET OF D KEY A;

E DATA SET (
   N NUMBER (3);
   R REAL;
   );
T SET OF E KEY N;
```

then the following BDMSALGOL program uses the ASSIGN statement to assign the value of the current record of data set E to link item L.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE EXAMPLEDB;
   EBCDIC ARRAY X[0:2];
   INTEGER Y;

   OPEN UPDATE EXAMPLEDB;
   WHILE NOT READ(CARD_FILE,<A3,I3>,X,Y) DO
      BEGIN
      FIND S AT A = X;
      FIND T AT N = Y;
      ASSIGN E TO L;
      END;
   CLOSE EXAMPLEDB;
END.
```

DMSII BEGINTRANSACTION Statement

<begintransaction statement>

```
— BEGINTRANSACTION ─┬──────────────────────────────────────┬──────────→
                    └─ <inputheadername> — <message area> ─┘

→─┬──────────────────────────────────────┬─ <restart data set> ───────→
  ├─ ( — <transaction record variable> — ) ─┤
  ├─ AUDIT ───────────────────────────────┤
  └─ NOAUDIT ─────────────────────────────┘

→─┬──────────────────────────┬─────────────────────────────────────────┤
  └─ <exception handling> ──┘
```

<restart data set>

```
—<qualification>──────────────────────────────────────────────────────┤
```

Explanation

The DMSII BEGINTRANSACTION statement places a program in transaction state. This statement can be used only with audited databases.

The BEGINTRANSACTION statement performs the following steps in order:

1. It captures the restart data set if AUDIT is specified.
2. It places a program in transaction state.

The transaction record variable construct identifies a transaction record created through the Transaction Processing System (TPS).

If the transaction record variable construct is used, it is the formal input transaction record variable, and NOAUDIT is the default action.

The word “AUDIT” causes the restart area to be captured. The path of the specified restart data set is not altered when the restart record is stored. AUDIT is the default action.

The word “NOAUDIT” causes the restart area to not be captured. The restart data set construct specifies the restart data set to be updated.

The restart data set construct identifies the data set containing the restart records that application programs can access to recover database information after a system failure.

An exception is returned if the BEGINTRANSACTION statement is attempted while the program is in transaction state. If any exception is returned, the program is not placed in transaction state. If an ABORT exception is returned, all records that the program had locked are freed.

Deadlock can occur during execution of a BEGINTRANSACTION statement.

Any attempt to modify an audited database when the program is not in transaction state results in a fault. The BDMSALGOL statements that modify databases are:

ASSIGN	INSERT
DELETE	REMOVE
GENERATE	STORE

Refer to the *DMSII Application Program Interfaces Programming Guide* for further details regarding audit and recovery. Refer to the COMS BEGINTRANSACTION statement when using DMSII and COMS and refer to the TPS BEGINTRANSACTION statement when using DMSII and TPS.

Additional information relating to the exception handling construct is included under "Database Status Word" in this section. Information on the inputheadername and message area constructs is included under "Declaring Input and Output Headers" and "RECEIVE Statement"in Section 3, "Using Communications Management System (COMS) Features." Related information is also included under "Passing Transaction Record Variables as Parameters" in Section 5, "Using DMSII Transaction Processing System (TPS) Extensions."

Additional information relating to the qualification construct is included under "Qualification of Database Items" in this section.

Additional information relating to DMSII transactions is included under "Declaring Transaction Record Variables" and "Transaction Processing Statements" in Section 5, "Using DMSII Transaction Processing System (TPS) Extensions."

Examples

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
    R RESTART DATA SET (
       P ALPHA (10);
       Q ALPHA (100);
       );
    D DATA SET (
       A ALPHA (3);
       N NUMBER (3);
       );
    S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the BEGINTRANSACTION statement can be used:

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY MY_A[0:2];
   INTEGER MY_N;

   OPEN UPDATE DBASE;
   MY_N := 1;
   WHILE MY_N < 100 DO
      BEGIN
      CREATE D;
      PUT D (N := MY_N);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R;
      MY_N := * + 1;
      END;
   WHILE NOT READ(CARD_FILE,<I3,A3>,MY_N,MY_A[0]) DO
      BEGIN
      LOCK S AT N = MY_N;
      BEGINTRANSACTION R;
      PUT D (A := MY_A[0]);
      STORE D;
      ENDTRANSACTION R;
      END;
   CLOSE DBASE;
END.
```

BDMS CANCELTRPOINT Statement

<canceltrpoint statement>

```
—CANCELTRPOINT  ——┬——————————————————————————————┬— <restart data set> —┤
                  │                              │
                  └— ( — <integer expression> — ) —┘
```

Explanation

The BDMS CANCELTRPOINT statement backs out all updates in a transaction to an intermediate save point (set through the SAVETRPOINT statement) or to the beginning of the transaction. The CANCELTRPOINT statement enables you to cancel all or part of the update assignments without having to terminate the transaction state. The program execution continues with the statement following the CANCELTRPOINT statement.

The inclusion of the integer expression construct causes DMSII to search for the corresponding SAVETRPOINT statement and cancel only those transactions lying between the two. If no corresponding SAVETRPOINT statement is found, or if the integer expression construct is omitted or is zero, then all update assignments performed during the current transaction state are discarded. However, the current transaction state is not terminated.

The restart data set construct identifies the data set containing the restart records that application programs can access to recover database information after a system failure.

Additional information on the BDMS SAVETRPOINT statement is included under “BDMS SAVETRPOINT Statement” in this section.

Example

In this example, there is an intermediate transaction point with an integer value of 1. If an error is detected, the CANCELTRPOINT statement backs out all updates accumulated after the SAVETRPOINT statement.

```
BEGINTRANSACTION R;
  ...
  SAVETRPOINT (1) R;
  ...
  IF ERROR ... THEN CANCELTRPOINT (1) R;
  ...
ENDTRANSACTION R;
```

BDMS CLOSE Statement

<BDMS close statement>

```
── CLOSE  ── <database identifier>──┬─────────────────────────┬──────────┤
                                    └─ <exception handling> ──┘
```

<database identifier>

```
── <BDMS identifier>─────────────────────────────────────────────────────┤
```

Explanation

The BDMS CLOSE statement closes a database when further access is no longer required and performs the following steps in order:

1. It closes the database.
2. It frees all locked records.

The database identifier specifies the database to be closed. If the database was declared to have an internal name, this internal name is the database identifier. If the database does not have an internal name but is a logical database, then the logical database name is the database identifier. For databases that do not have an internal name and are not logical databases, the database name is the database identifier.

An exception is returned if the CLOSE statement attempts to close a database that is not open. A database abort occurs if the CLOSE statement attempts to close a database that is in transaction state.

Use of the CLOSE statement is optional; the system closes any open database when a program terminates. A syncpoint in the audit file occurs when a database is successfully closed.

The CLOSE statement is the only BDMSALGOL statement in which the status word has meaning when no exception is indicated. Therefore, after a CLOSE statement, the status word should be examined by the program and appropriate action taken, whether or not an exception is returned. An ABORT exception can be obtained in this manner.

Additional information relating to the BDMS identifier construct is included under “BDMS Identifier Construct” in this section. Information on the exception handling construct is included under “Database Status Word” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
SS SUBSET OF D BIT VECTOR;
X SUBSET OF D BIT VECTOR;
Y SUBSET OF D BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program shows how to use the CLOSE statement to close DBASE.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];

   OPEN INQUIRY DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      GET D (MA[0] := A,MB := B);
      IF MB THEN
         GET D (MR := N)
      ELSE
         MR := 0;
      WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
            MN,MA[0],MB,MR);
      END;
   CLOSE DBASE;
END.
```

CREATE Statement

<create statement>

```
 ─ CREATE  ─ <data set>─┬──────────────────────────────────┬──────────→
                        └─ ( ─ <arithmetic expression> ─ ) ─┘

→─┬────────────────────────┬───────────────────────────────────────────┤
  └─ <exception handling> ─┘
```

Explanation

The CREATE statement initializes the user work area of a data set record and performs the following steps in order:

1. It frees the current record of the specified data set. (If the INDEPENDENTTRANS option is set in DASDL for the database and the program is in transaction state, the CREATE statement does not free the current record.)
2. It reads any specified expression to determine the format of the record to be created.
3. It initializes data items to one of the following values:
   a. The DASDL-declared INITIALVALUE, if present
   b. The DASDL-declared NULL, if present
   c. The default NULL

Note: *When creating partitioned data sets, you must establish the partition master record prior to execution of the CREATE command.*

The data set construct specifies the data set to be initialized. The current path of the data set is not changed until a subsequent STORE statement has completed successfully.

The arithmetic expression specifies the type of record to be created. This arithmetic expression is required when a variable-format record is created; otherwise, it must not appear.

An exception is returned if the arithmetic expression does not represent a valid record type.

Normally, the CREATE statement is eventually followed by a STORE statement, which places the newly created record into the data set. However, if a subsequent STORE operation is not desired, the CREATE statement can be nullified by a subsequent CREATE, DELETE, FIND, BDMS FREE, BDMS LOCK, MODIFY, or RECREATE statement.

The CREATE statement sets up only a record area. If the record contains embedded structures, the master record must be stored before entries can be created in the embedded structures. If only entries in the embedded structure are created (that is, if items in the master are not altered), the master need not be stored a second time.

Additional information relating to the CREATE statement is included under “Creating Transaction Record Formats” in Section 5, “Using DMSII Transaction Processing System (TPS) Extensions.”

Additional information relating to the data set construct is included under “Selecting a Record in a Data Set” in this section. Information on the exception handling construct is included under “Database Status Word” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program shows how a record of data set D can be created and stored.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY X[0:9];
   INTEGER Y,Z;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A10,I1,I3>,X[0],Y,Z) DO
      BEGIN
      CREATE D;
      PUT D (A := X[0]);
      IF Y = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := Z);
      STORE D;
      END;
   CLOSE DBASE;
END.
```

DMSII DELETE Statement

<delete statement>

```
— DELETE  — <selection expression> ——┬———————————————————————┬——————————→
                                     └— <exception handling> —┘

→—┬——————————————————————————┬————————————————————————————————————————————┤
  └— ( — <input mapping> — ) —┘
```

Explanation

The DMSII DELETE statement is identical to the FIND statement except that if a record is found, it is locked and then deleted. The DELETE statement performs the following steps in order:

1. It frees the current record, unless the selection expression is the name of the data set and the current record is locked. In that case, the locked status is not altered. (If the INDEPENDENTTRANS option is set in DASDL for the database and the program is in transaction state, the DELETE statement does not free the current record.)
2. It alters the current path to point to the record specified by the selection expression, and locks this record.
3. It transfers that record to the user work area.
4. It removes the record from all sets and automatic subsets, but not from manual subsets.
5. It removes the record from the data set.

If the record is found but cannot be deleted, an exception is returned and the DELETE statement terminates, leaving the current path pointing to the record specified by the selection expression.

If a set selection expression is used and the record is not found, then an exception is returned and the set path is changed and invalidated. It refers to a location between the last key less than the condition and the first key greater than the condition. A set selection expression using NEXT or PRIOR can be done from this point provided keys greater than and less than the condition exist. The current path of the data set, the current record, and the current paths of any other sets for that data set remain unchanged.

It is the responsibility of the programmer to ensure that no manual subset refers to the record being deleted.

The selection expression identifies the record to be deleted.

An exception is returned and the record is not deleted if the record has counted links pointing to it, or if the record contains a nonnull link or a nonempty embedded structure.

When the DELETE statement completes, the current paths still refer to the deleted record. Therefore, a FIND statement on the current record results in a NOTFOUND exception; however, FIND NEXT and FIND PRIOR statements are still appropriate.

Additional information relating to the selection expression construct is included under "Selecting a Record in a Data Set" in this section. Information on the exception handling construct is included under "Database Status Word" in this section. Information on the input mapping construct is included under "Input Mapping Used with Retrieval Statements" in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates the use of the DELETE statement to delete a record of the data set D where item N is equal to the value of X:

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      DELETE S AT N = X;
   CLOSE DBASE;
END.
```

DMTERMINATE Statement

<dmterminate statement>

```
— DMTERMINATE ——┬— <Boolean identifier> —┬——————————————————————————————┤
                ├— <integer identifier> —┤
                └— <real identifier> ————┘
```

Explanation

The DMTERMINATE statement aborts the current action. When an exception occurs that the program does not handle, the DMTERMINATE statement can be called to produce the same results as if the exception-handling syntax had not been specified in the statement; that is, the DMTERMINATE statement causes the program to terminate with a fault.

For more information concerning the Boolean identifier, integer identifier, and real identifier, refer to Volume 1.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program shows an example of the use of the DMTERMINATE statement.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   BOOLEAN RSLT;
   REAL RRSLT = RSLT;
   INTEGER X;

   OPEN UPDATE DBASE;
   FIND FIRST D :RSLT;
   IF RSLT THEN
      BEGIN
      DISPLAY("D IS EMPTY DATA SET");
      DMTERMINATE(RSLT);
      END
   ELSE
      WHILE NOT READ(CARD_FILE,<I3>,X) DO
         BEGIN
         DELETE S AT N = X :RSLT;
         IF RRSLT.DMERROR = NOTFOUND THEN
            DMTERMINATE(RSLT);
         END;
   CLOSE DBASE;
END.
```

DMSII ENDTRANSACTION Statement

<endtransaction statement>

```
— ENDTRANSACTION ──┬──────────────────────────────────┬─ <restart data set──────→
                   ├─ ( — <endtransaction parameters> — ) ─┤
                   ├─ AUDIT ──────────────────────────┤
                   └─ NOAUDIT ────────────────────────┘

→─┬────────┬─┬─────────────────────────┬──────────────────────────────────┤
  └─ SYNC ─┘ └─ <exception handling> ──┘
```

<endtransaction parameters>

```
— <transaction record variable ID> — , — <saveoutput procedure identifier> ┤
```

<saveoutput procedure identifier>

```
— <procedure identifier> ──────────────────────────────────────────────────┤
```

Explanation

The DMSII ENDTRANSACTION statement takes a program out of transaction state. This statement can be used only with audited databases. The ENDTRANSACTION statement performs the following steps in order:

1. It captures the restart area if AUDIT is specified.
2. It forces a syncpoint if the SYNC option is specified.
3. It implicitly frees all records of the database that the program has locked.

If the endtransaction parameters form is used, the transaction record variable ID construct is the formal input transaction record variable. The saveoutput procedure identifier is the name of the SAVERESPONSETR formal procedure. For more information about the SAVERESPONSETR procedure, refer to the *DMSII Transaction Processing System (TPS) Programming Guide.*

The word “AUDIT” causes the restart area to be captured. The path of the restart data set is not altered when the restart record is stored.

The word “NOAUDIT” causes the restart area to not be captured. NOAUDIT is the default action.

The restart data set construct identifies the data set containing the restart records that application programs can access to recover database information after a system failure.

The word “SYNC” forces a syncpoint.

An exception is returned if an ENDTRANSACTION statement is attempted and the program is not in transaction state.

Records are freed in all cases. If an exception occurs, the transaction is not applied to the database.

Refer to the *DMSII Application Program Interfaces Programming Guide* for information regarding audit and recovery. Refer to the COMS ENDTRANSACTION statement when using COMS and DMSII and refer to the TPS ENDTRANSACTION statement when using TPS and DMSII.

Additional information relating to DMSII transactions is included under “Declaring Transaction Record Variables” and “Transaction Processing Statements” in Section 5, “Using DMSII Transaction Processing System (TPS) Extensions.”

Additional information relating to the <exception handling> construct is included under “Database Status Word” in this section.

For information concerning transaction records, consult Section 5, “Using DMSII Transaction Processing System (TPS) Extensions.” For more information concerning procedure identifiers, refer to Volume 1.

Examples

Assume a database named DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (3);
   N NUMBER (3);
   );
S SET OF D KEY N;
```

The following BDMSALGOL program demonstrates how the ENDTRANSACTION statement can be used with this database.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY MY_A[0:2];
   INTEGER MY_N;

   OPEN UPDATE DBASE;
   MY_N := 1;
   WHILE MY_N < 100 DO
      BEGIN
      CREATE D;
      PUT D (N := MY_N);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R;
      MY_N := * + 1;
      END;
   WHILE NOT READ(CARD_FILE,<I3,A3>,MY_N,MY_A[0]) DO
      BEGIN
      LOCK S AT N = MY_N;
      BEGINTRANSACTION R;
      PUT D (A := MY_A[0]);
      STORE D;
      ENDTRANSACTION R;
      END;
   CLOSE DBASE;
END.
```

FIND Statement

<find statement>

```
──┬─ FIND ──┬─ <selection expression> ─┬───────┬┬─────────────────────────┬─→
  │         └─ <database identifier> ──┘       │└─ <exception handling> ─┘
  └─ FIND KEY OF ─ <set selection expression> ─┘

→─┬─────────────────────────────┬─────────────────────────────────────────────┤
  └─ ( ─ <input mapping> ─ ) ───┘
```

Explanation

The FIND statement transfers a record to the user work area associated with a data set or global data and performs the following steps in order:

1. It frees a locked record in the data set if a data set is specified in the FIND statement, or frees a locked record in the associated data set if a set is specified in the FIND statement. (If the INDEPENDENTTRANS option is set in DASDL for the database and the program is in transaction state, the FIND statement does not free the locked record.)
2. It alters the current path to point to the record specified by the selection expression or database name.
3. It transfers that record to the user work area.

The FIND statement does not prevent reads by other transactions before an update transaction is complete.

The selection expression form is used to specify the record to be transferred to the user work area.

The database identifier form is used to specify the global data record to be transferred to the user work area associated with the global data. If no global data was described in DASDL for the database, a syntax error occurs.

If the invoked database contains a remap of the global data, the name of the logical database, not the name of the global data remap, is used to LOCK the global data record.

The form “FIND KEY OF set selection expression” moves the key and any associated data (as specified in DASDL) from the key entry to the user work area. A physical read is not performed on the data set; consequently, all items in the record area that do not appear in the key entry retain whatever value they had before the FIND statement. The current path of the data set is not affected.

If an exception is returned, the record is not freed. If a set selection expression is used and the record is not found, then an exception is returned and the set path is changed and invalidated. It refers to a location between the last key less than the condition and the first key greater than the condition. A set selection expression using NEXT or PRIOR can be done from this point provided keys greater than and less than the condition exist. The current path of the data set, the current record, and the current paths of any other sets for that data set remain unchanged.

To access data items, input mapping is required.

Additional information relating to the selection expression and set selection expression constructs is included under “Selecting a Record in a Data Set” in this section. Information on the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the exception handling construct is included under “Database Status Word” in this section. Information on the input mapping construct is included under “Input Mapping Used with Retrieval Statements” in this section.

Additional information relating to the input mapping construct is included under “Input Mapping Used with Retrieval Statements” in this section.

Examples

```
FIND FIRST EMP AT DEPT-NO = 1019 :RSLT;
IF RSLT THEN
   POP-EMPS[1019] := 0;

FIND EMP AT EMP-NO = SSN :RSLT;
IF RSLT THEN
   ERR_OUT(INV_EMP_NO_ERR);

FIND NEXT EMP :RSLT;
IF RSLT THEN
   GO NO_MORE_EMP;

FIND FIRST OVR-65 AT DEPT-NO = 1019 :RSLT;
IF RSLT THEN
   POP-OVR-65[1019] := 0;
```

BDMS FREE Statement

<BDMS free statement>

```
 ─ FREE ──┬─ <data set> ────────────────────┬─┬──────────────────────────┬──┤
          ├─ <database identifier> ─────────┤ └─ <exception handling> ─┘
          └─ STRUCTURE  ─ <data set name> ──┘
```

Explanation

The BDMS FREE statement unlocks the current record or structure.

Normally, a FREE statement can be executed after any operation. However, the FREE statement is ignored if the current record or structure is already free, if no current record or structure is present, or if the INDEPENDENTTRANS option is set in DASDL for the database and the program is in transaction state.

The FREE statement can be used to unlock a record or structure that the user anticipates cannot be implicitly freed for a relatively long time. A FREE statement executed on a record or structure enables other programs to lock the record.

The data set form is used to specify the data set whose current record is to be unlocked. The data set path and current record area remain unchanged.

The database identifier form is used to specify the global data record to be unlocked. The data set path and current record area remain unchanged.

The STRUCTURE data set name construct frees all records in the structure.

If an exception is returned, the state of the database remains unchanged.

The FREE statement is optional in many situations because DELETE, FIND, BDMS LOCK, and MODIFY statements can free a record before they execute. FIND, LOCK, and MODIFY statements that use sets or subsets can free the locked record or structure only if a new record or structure is successfully retrieved. Otherwise, the previously locked record or structure remains locked. In general, an implicit FREE statement is performed, if necessary, during any operation that establishes a new data set path.

Additional information relating to the data set construct is included under “Selecting a Record in a Data Set” in this section. Information on the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the exception handling construct is included under “Database Status Word” in this section.

Examples

If the database DBASE is described in DASDL as follows:

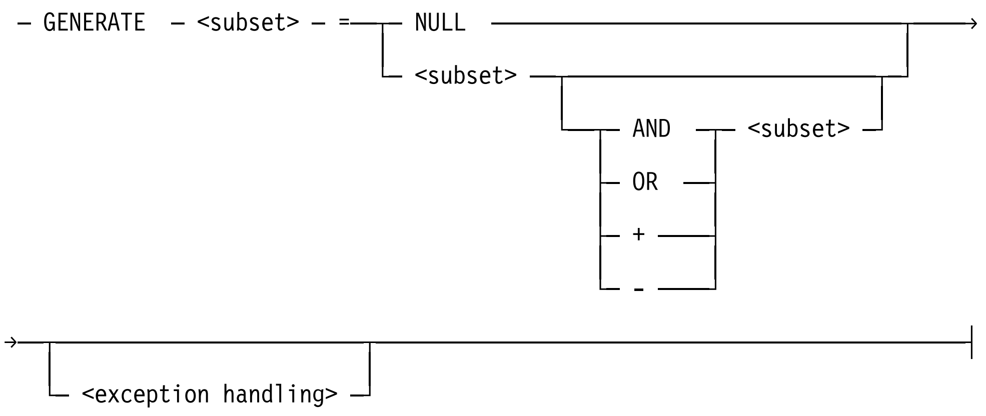
```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates the use of the FREE statement to unlock the current record of data set D.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      BEGIN
      LOCK S AT N = X;
      IF DMTEST(A ISNT NULL) THEN
         DELETE D
      ELSE
         FREE D;
      END;
   CLOSE DBASE;
END.
```

GENERATE Statement

<generate statement>



Explanation

The GENERATE statement creates an entire subset in one operation. All subsets must be disjoint bit vectors. The GENERATE statement performs the following steps in order:

1. It deletes all the records from the subset to be generated if it is not already empty.
2. It assigns a null value, the records in another subset, or a combination of the records in two other subsets to the subset that is generated.

The subset to the left of the equal sign (=) is the name of the subset to be generated. This subset must be a manual subset, which must be a disjoint bit vector.

The word “NULL” assigns a null value to the generated subset.

If subset follows the equal sign, it is the name of the subset whose records are to be assigned to the generated subset. This subset must be of the same data set as the generated subset, and it must be a disjoint bit vector.

If to the right of the equal sign there are two subsets joined by the operation AND, OR, +, or -, then these two subsets are to be combined in the specified manner. The result is then assigned to the generated subset. The two subsets must be of the same data set, and must be disjoint bit vectors.

The operator “AND” specifies that the intersection of the two subsets is to be assigned to the generated subset. The intersection is defined to be all the records in the first subset that are also in the second subset.

The operator “OR” specifies that the union of the two subsets is to be assigned to the generated subset. The union is defined to be all the records that are in either the first subset or the second subset.

The operator "+" specifies that the exclusive OR of the two subsets is to be assigned to the generated subset. The exclusive OR consists of the records in either the first subset or the second subset, but not the records that appear in both subsets.

The operator "-" specifies that the subset difference of the two subsets is to be assigned to the generated subset. The subset difference is defined to be the records in the first subset that are not in the second subset.

Additional information relating to the subset construct is included under "Selecting a Record in a Data Set" in this section. Information on the exception handling construct is included under "Database Status Word" in this section.

Examples

If the database DBASE is described in DASDL as the following:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D WHERE (N GEQ 21 AND NOT B) BIT VECTOR;
Y SUBSET OF D WHERE (R LSS 1000) BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program shows how the GENERATE statement can be used to assign all the records that are in both X and Y to subset Z.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY S[0:2];
   INTEGER T,U,V;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A3,I1,I3,I4>,S,T,U,V) DO
      BEGIN
      CREATE D;
      PUT D (A := S);
      IF T = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := U);
      PUT D (R := V);
      STORE D;
      END;
   GENERATE Z = X AND Y;
   CLOSE DBASE;
END.
```

GET Statement

<get statement>

```
– GET  ──┬─ <data set> ──────────┬─ ( – <input mapping> – ) ──────────┤
         └─ <database identifier> ─┘
```

Explanation

The GET statement is used to transfer information from the user work area associated with a data set or global data record into program variables or arrays.

The GET statement does not access the database; it assumes that prior database operations have loaded the proper record or data items into the user work area.

The data set construct is used to transfer information from the user work area associated with this data set into a program variable or array.

The database identifier is used to transfer information from the user work area associated with the global data record into a program variable or array.

No exceptions are associated with the GET statement. However, if the database containing the referenced data set or global data record has not been opened at the time execution of the GET statement is attempted, the program terminates with a fault.

Additional information relating to the data set construct is included under “Selecting a Record in a Data Set” in this section. Information on the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the input mapping construct is included under “Input Mapping Used with Retrieval Statements” in this section.

Examples

Assume a database named DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
S SET OF D KEY N;
```

The following BDMSALGOL program demonstrates how the GET statement can be used to assign current values of data items to program variables and arrays.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];

   OPEN INQUIRY DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      GET D (MA[0] := A,MB := B);
      IF MB THEN
         GET D (MR := R)
      ELSE
         MR := 0;
      WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
            MN,MA[0],MB,MR);
      END;
   CLOSE DBASE;
END.
```

DMSII INSERT Statement

<insert statement>

```
 — INSERT  — <data set> — INTO  — <subset>──┬──────────────────────────┬────┤
                                            │                          │
                                            └─ <exception handling> ───┘
```

Explanation

The DMSII INSERT statement places a record into a manual subset and performs the following steps in order:

1. If INDEPENDENTTRANS is set in DASDL, the current record of the specified data set will be locked unconditionally.
2. Inserts the current record of the specified data set into the specified subset.
3. Alters the set path for the specified subset to point to the inserted record.

The data set construct specifies the data set whose current record is inserted into the subset specified by subset. The path of the specified data set must be the object data set of the specified subset.

The subset must be a manual subset, and it must be a subset of the specified data set.

The path of the specified data set must refer to a valid record; if not, an exception is returned. Other reasons an exception is returned are:

- If duplicates are not allowed for the specified subset and the record to be inserted has a key identical to that of a record currently in that subset.
- If the specified subset is embedded in a data set that does not have a valid current record.
- If “LOCK TO MODIFY DETAILS” was specified in DASDL and the current record is not locked.
- If the locking procedure in Step 1 results in a deadlock situation.

Additional information relating to the subset construct is included under “Selecting a Record in a Data Set” in this section. Information on the exception handling construct is included under “Database Status Word” in this section. Information on the data set construct is included under “Selecting a Record in a Data Set” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program shows how the INSERT statement can be used to place the current record of data set D into subset X.

```
BEGIN
   DATABASE DBASE;
   BOOLEAN RSLT;
   INTEGER MN;

   OPEN UPDATE DBASE;
   SET D TO BEGINNING;
   FIND NEXT D :RSLT;
   WHILE NOT RSLT DO
      BEGIN
      GET D (MN := N);
      IF MN > 10 THEN
         INSERT D INTO X;
      FIND NEXT D :RSLT;
      END;
   CLOSE DBASE;
END.
```

BDMS LOCK Statement

<BDMS lock statement>

```
──┬─ LOCK ───┬──┬─ <selection expression> ──────┬──┬────────────────────────┬──→
  └─ MODIFY ─┘  ├─ <database identifier> ───────┤  └─ <exception handling> ─┘
                └─ STRUCTURE ─ <data set name> ─┘

→──┬──────────────────────────┬─────────────────────────────────────────────┤
   └─ ( ─ <input mapping> ─ ) ─┘
```

Explanation

The BDMS LOCK statement is similar to the FIND statement, except that if a record or structure is found, it is locked against a concurrent modification by another user. The LOCK statement provides an exclusive lock and can designate either a structure lock or a record lock. The program owning an exclusive lock prevents all other programs from successfully executing a SECURE or LOCK statement. However, other programs can successfully execute a FIND statement. Use the SECURE statement to allow other programs to secure the record or structure.

The words “LOCK” and “MODIFY” are synonyms.

If the record or structure to be locked has already been locked by another program, the system performs a contention analysis. In this case, the present program waits until the record or structure is unlocked. However, if a wait would result in a deadlock, all records or structures locked by the program with the lowest priority involved in the deadlock are unlocked, and the operation in that program terminates with a DEADLOCK exception.

A DEADLOCK exception also occurs if the program waits on a LOCK statement longer than the period specified by the MAXWAIT task attribute.

Consult the *DMSII Application Program Interfaces Programming Guide* for more information on the DEADLOCK exception. For information about task attributes, consult the *Task Attributes Programming Reference Manual.*

The LOCK statement performs the following steps in order:

1. If the LOCK statement specifies a data set, then a locked record or structure in the data set is freed. If the LOCK statement specifies a set, then a locked record or structure in the associated data set is freed. (If the INDEPENDENTTRANS option is set in DASDL for the database and the program is in transaction state, the statement does not free the locked record or structure.)
2. It alters the current path to point to the record or structure specified by the selection expression or database identifier.
3. It locks the specified record or structure and then transfers that record to the user work area.

Implicit structure locks are freed after execution of the ENDTRANSACTION statement.

The selection expression is used to specify the record to be locked.

The database identifier is used to specify the global data record to be locked. If the invoked database contains a remap of the global data, the name of the logical database, not the name of the global data remap, is used to LOCK the global data record.

The STRUCTURE data set name construct locks all records in the structure. This is an explicit structure lock; therefore, the records are not freed after the execution of the ENDTRANSACTION statement. Explicit structure locks are freed with the FREE STRUCTURE statement or by closing the database.

If an exception is returned, the record is not freed.

If a LOCK statement using a set selection expression returns an exception, the current path of the specified set is invalidated. However, the current path of the data set, the current record, and the current paths of any other sets for that data set remain unaltered.

To access data items, the input mapping construct must appear.

Because no other user can lock a record or structure once it is locked, a record or structure must be freed when it is no longer required to be locked. A record or structure can be freed explicitly by a BDMS FREE statement or implicitly by a subsequent CREATE, DELETE, FIND, BDMS LOCK, or RECREATE statement on the same data set.

Additional information relating to locked records and structures is included under "SECURE Statement" in this section.

Additional information relating to the selection expression construct is included under `Selecting a Record in a Data Set" in this section. Information on the database identifier construct is included under "BDMS CLOSE Statement" in this section. Information on the exception handling construct is included under "Database Status Word" in this section. Information on the input mapping construct is included under "Input Mapping Used with Retrieval Statements" in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates the use of the LOCK statement to lock records of subset X.

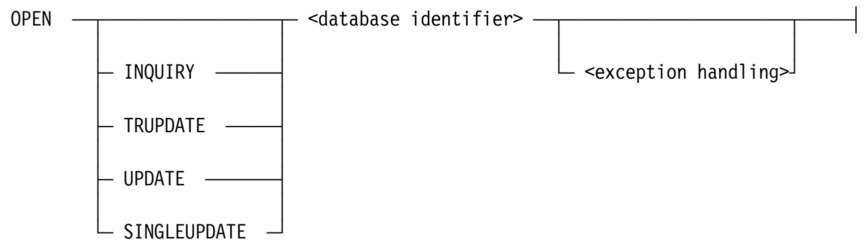
```
BEGIN
   DATABASE DBASE;
   BOOLEAN RSLT;
   INTEGER MN;

   OPEN UPDATE DBASE;
   SET X TO BEGINNING;
   LOCK NEXT X :RSLT;
   WHILE NOT RSLT DO
      BEGIN
      GET D (MN := N);
      IF MN <= 10 THEN
         BEGIN
         REMOVE D FROM X;
         DELETE D;
         END
      ELSE
         BEGIN
         PUT D (B := TRUE);
         STORE D;
         END;
      LOCK NEXT X :RSLT;
      END;
   CLOSE DBASE;
END.
```

DMSII MODIFY Statement

The DMSII MODIFY statement is described under the BDMS LOCK statement in this section. They are synonyms.

BDMS OPEN Statement

<BDMS open statement>



Explanation

The BDMS OPEN statement opens a database for subsequent access and specifies the access mode. The OPEN statement performs the following steps in order:

1. It opens an existing database. Appropriate “NO FILE” messages are displayed if files required for invoked structures are not present in the system directory.
2. It performs an implicit CREATE statement on the restart data set.

The word “INQUIRY” enforces read-only access to the database. This option is specified when no update operations are to be performed on the database. An exception is returned if the following BDMSALGOL statements are used when the database has been opened with the INQUIRY option:

ASSIGN	GENERATE
BEGINTRANSACTION	INSERT
DELETE	REMOVE
ENDTRANSACTION	STORE

The data management system does not open any audit files if the “OPEN INQUIRY” form has been used by all programs accessing the database.

The word “UPDATE” enables the program to modify the database being opened. The UPDATE option must be specified in order to use the BDMSALGOL statements listed above under the INQUIRY option. UPDATE is the default option.

The word “TRUPDATE” must be specified in order to use the MIDTRANSACTION statement or the transaction record variable ID form of the BEGINTRANSACTION or ENDTRANSACTION statements. Refer to Section 5, “Using the DMSII Transaction Processing System (TPS) Extensions,” for more information on the MIDTRANSACTION statement.

The word “SINGLEUPDATE” enables only one user to modify the database being opened. The SINGLEUPDATE option can use the BDMSALGOL statements listed under the INQUIRY option.

The database identifier specifies the database to be opened.

If an exception is returned, the state of the database remains unchanged. An exception is returned if the database is already open.

An OPEN statement must be executed before the first access of the database; otherwise, the program terminates with a fault.

Additional information relating to the BDMS OPEN statement is included under “BDMS OPEN Statement with TPS” and “Transaction Processing Statements” in Section 5, “Using DMSII Transaction Processing System (TPS) Extensions”.

Additional information relating to the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the exception handling construct is included under “Database Status Word” in this section.

Examples

Assume a database named DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
SS SUBSET OF D BIT VECTOR;
X SUBSET OF D BIT VECTOR;
Y SUBSET OF D BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

The following BDMSALGOL program demonstrates the use of the OPEN statement with the INQUIRY option to open database DBASE and perform read-only actions on the database.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];

   OPEN INQUIRY DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      GET D (MA[0] := A,MB := B);
      IF MB THEN
         GET D (MR := N)
      ELSE
         MR := 0;
      WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
            MN,MA[0],MB,MR);
      END;
   CLOSE DBASE;
END.
```

The following BDMSALGOL program demonstrates the use of the OPEN statement with the UPDATE option to open database DBASE and perform update actions on the database.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      BEGIN
      LOCK S AT N = X;
      IF DMTEST(A ISNT NULL) THEN
         BEGIN
         BEGINTRANSACTION R;
         DELETE D;
         ENDTRANSACTION R;
         END
      ELSE
         FREE D;
      END;
   CLOSE DBASE;
END.
```

PUT Statement

<put statement>

```
— PUT  ——┬— <data set> ——————————————┬— ( — <output mapping> — ) ——————————┤
         └— <database identifier> ——┘
```

Explanation

The PUT statement transfers information from program expressions into the user work area associated with a data set or global data record.

The PUT statement does not update the database; a subsequent STORE statement must be executed to place the data in the user work area into the database.

Any number of PUT statements can be used to update items before a STORE statement is executed.

The data set form is used to transfer information associated with this data set into the user work area.

The database identifier form is used to transfer information associated with the global data record into the user work area.

Output mappings transfer the value of a program variable or expression to a DASDL-declared data item. If the data item is an occurring item, it must be subscripted appropriately.

No exceptions are associated with the PUT statement. However, if the database containing the specified data set or the specified database has not been opened, the program terminates with a fault.

Additional information relating to the data set construct is included under “Selecting a Record in a Data Set” in this section. Information on the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the output mapping construct is included under “Output Mapping Used with Storage Statements” in this section.

Examples

Assume a database named DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D BIT VECTOR;
```

The following BDMSALGOL program demonstrates how the PUT statement can be used to assign values to data items.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY S[0:2];
   INTEGER T,U,V;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A3,I1,I3,I4>,S[0],T,U,V) DO
      BEGIN
      CREATE D;
      PUT D (A := S);
      IF T = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := U,R := V);
      STORE D;
      END;
   CLOSE DBASE;
END.
```

RECREATE Statement

<recreate statement>

```
— RECREATE  — <data set> ——┬——————————————————————————————————┬——————→
                           └— ( — <arithmetic expression> — ) —┘

→—┬—————————————————————————┬——————————————————————————————————————————┤
  └— <exception handling> —┘
```

Explanation

The RECREATE statement partially initializes the user work area. All data items remain unaltered; however, control items such as links, sets, counts, and data sets are unconditionally set to NULL.

For variable-format records, the record type supplied must be the same as that supplied in the CREATE statement that created the record. If not, the subsequent STORE statement results in a DATAERROR subcategory 4.

The RECREATE statement performs the following steps in order:

1. It frees the current record of the specified data set.
2. It reads any specified arithmetic expression to determine the format of the record to be created.
3. It unconditionally sets links, sets, counts, and data sets to NULL.

The data set construct specifies the data set to be initialized.

The arithmetic expression specifies a value indicating the type of record to be created. This arithmetic expression is required when a variable-format record is created; otherwise, it must not appear.

An exception is returned if the arithmetic expression does not represent a valid record type.

Additional information relating to the data set construct is included under “Selecting a Record in a Data Set” in this section. Information on the exception handling construct is included under “Database Status Word” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
SS SUBSET OF D BIT VECTOR;
X SUBSET OF D BIT VECTOR;
Y SUBSET OF D BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates how the RECREATE statement can be used to partially initialize a record of data set D.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY X[0:9];
   INTEGER Y,Z;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A10,I1,I3>,X[0],Y,Z) DO
      BEGIN
      CREATE D;
      PUT D (A := X[0]);
      IF Y = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := Z);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R;
      RECREATE D;
      PUT D (N := Z+1);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R;
      END;
   CLOSE DBASE;
END.
```

REMOVE Statement

<remove statement>

```
 ─ REMOVE ──┬─ CURRENT ────┬─ FROM ─ <subset> ──┬──────────────────────────┬──┤
            └─ <data set> ─┘                    └─ <exception handling> ──┘
```

Explanation

The REMOVE statement is similar to the FIND statement, except that if a record is found, it is locked and then removed from the specified subset.

The REMOVE statement performs the following steps in order:

1. Frees the current record. (If the INDEPENDENTTRANS option is set in DASDL for the database and the program is in transaction state, the REMOVE statement does not free the current record.)
2. Alters the current path to point to the record specified by CURRENT or the data set.
3. If INDEPENDTTRANS is set in DASDL, it will lock the previously found record and will then remove the record from the specified subset.

If an exception occurs after step 2, the current path is invalid. If an exception occurs after step 3, the operation terminates, leaving the current path pointing to the record specified by CURRENT or by the data set.

The word “CURRENT” removes the current record from the specified subset. If this option is specified, the subset must have a valid current record; if it does not have a valid current record, an exception is returned.

The data set construct is used to find and remove from the specified subset the record referenced by the current path. An exception is returned if the record is not in the subset.

The subset construct specifies the subset from which a record is to be deleted. The subset must be a manual subset of the specified data set.

If the subset is embedded in a data set, the data set must have a current record defined and that record must be locked; if not, an exception is returned. Exceptions are also returned

- If CURRENT is specified and the specified subset does not have a valid current record.
- If a data set is specified and the record is not in the subset.
- If the specified subset is embedded in a data set, and the data set does not have a current record defined and locked.
- If the locking procedure in Step 3 results in a deadlock situation.

After the REMOVE statement is executed, the current paths still refer to the deleted record. Therefore, a subsequent FIND statement on the current record results in a NOTFOUND exception. However, the FIND NEXT and FIND PRIOR forms of the FIND statement give valid results.

Additional information relating to the data set construct is included under “Selecting a Record in a Data Set” in this section. Information on the subset construct is included under “Selecting a Record in a Data Set” in this section. Information on the exception handling construct is included under “Database Status Word” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
SS SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates the use of the REMOVE statement to lock and remove the record of data set D that is referenced by the current path from the subset SS.

```
BEGIN
   DATABASE DBASE;
   BOOLEAN RSLT;
   INTEGER MN;

   OPEN UPDATE DBASE;
   SET SS TO BEGINNING;
   FIND NEXT SS :RSLT;
   WHILE NOT RSLT DO
      BEGIN
      GET D (MN := N);
      IF MN < 10 THEN
         REMOVE D FROM SS;
      FIND NEXT SS :RSLT;
      END;
   CLOSE DBASE;
END.
```

BDMS SAVETRPOINT Statement

<savetrpoint statement>

```
 — SAVETRPOINT  — ( — <integer expression> — ) — <restart data set> ———————┤
```

Explanation

The BDMS SAVETRPOINT statement creates an intermediate transaction point. You use the SAVETRPOINT in conjunction with the CANCELTRPOINT statement. All updates occurring between a SAVETRPOINT statement and a CANCELTRPOINT statement can be backed out if an error condition is encountered that disrupts the integrity of the updates.

The integer expression construct marks the intermediate transaction point. It must have the same value as the integer expression construct of the corresponding CANCELTRPOINT statement.

The restart data set construct identifies the data set containing the restart records that application programs can access to recover database information after a system failure.

Additional information relating to the BDMS CANCELTRPOINT statement is included under “BDMS CANCELTRPOINT Statement” in this section.

Example

In this example, an intermediate transaction point has an integer value of 1. If an error is detected, the CANCELTRPOINT statement backs out all updates accumulated after the SAVETRPOINT statement.

```
BEGINTRANSACTION R;
  ...
  SAVETRPOINT (1) R;
  ...
  IF ERROR ... THEN CANCELTRPOINT (1) R;
  ...
ENDTRANSACTION R;
```

SECURE Statement

<secure statement>

```
 — SECURE ——┬— <selection expression> ——————————┬——┬———————————————————————┬—→
            ├— <database identifier> ————————————┤  └— <exception handling> —┘
            └— STRUCTURE  — <data set name> —————┘

→——┬——————————————————————————————┬————————————————————————————————————————————┤
   └— ( — <input mapping> — ) —————┘
```

Explanation

The SECURE statement is similar to the FIND statement, except that if a record or structure is found, it is locked against a concurrent modification by another user. The SECURE statement provides a shared lock and allows other programs to execute a SECURE statement or a FIND statement successfully. However, other programs cannot execute a LOCK statement successfully.

If the record or structure to be locked has already been locked by another program, the system performs a contention analysis. In this case, the present program waits until the record or structure is unlocked. However, if a wait would result in a deadlock, all records or structures locked by the program with the lowest priority involved in the deadlock are unlocked, and the operation in that program terminates with a DEADLOCK exception.

A DEADLOCK exception also occurs if the program waits on a SECURE statement longer than the period specified by the MAXWAIT task attribute.

Consult the *DMSII Application Program Interfaces Programming Guide* for more information on the DEADLOCK exception. For information about task attributes, consult the *Task Attributes Programming Reference Manual.*

The SECURE statement performs the following steps in order:

1. If the SECURE statement specifies a data set, then a locked record or structure in the data set is freed. If the SECURE statement specifies a set, then a locked record or structure in the associated data set is freed. (If the INDEPENDENTTRANS option is set in DASDL for the database and the program is in transaction state, the statement does not free the locked record or structure.)
2. It alters the current path to point to the record or structure specified by the selection expression or database identifier.
3. It locks the specified record or structure and then transfers that record to the user work area.

Implicit structure locks are freed after execution of the ENDTRANSACTION statement.

The selection expression is used to specify the record to be locked.

The database identifier is used to specify the global data record to be locked. If the invoked database contains a remap of the global data, the name of the logical database, not the name of the global data remap, is used to lock the global data record.

The STRUCTURE <data set name> construct locks all records in the structure. This is an explicit structure lock; therefore, the records are not freed after execution of the ENDTRANSACTION statement. Explicit structure locks are freed with the FREE STRUCTURE statement or by closing the database.

If an exception is returned, the record is not freed.

If a SECURE statement using a set selection expression returns an exception, the current path of the specified set is invalidated. However, the current path of the data set, the current record, and the current paths of any other sets for that data set remain unaltered.

To access data items, the input mapping construct must appear.

Because no other user can lock a record or structure once it is locked, a record or structure must be freed when it is no longer required to be locked. A record or structure can be freed explicitly by a BDMS FREE statement or implicitly by a subsequent CREATE, DELETE, FIND, BDMS LOCK, or RECREATE statement on the same data set.

Additional information relating to locked records and structures is included under “BDMS LOCK Statement” in this section.

Additional information relating to the selection expression construct is included under `Selecting a Record in a Data Set” in this section. Information on the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the exception handling construct is included under “Database Status Word” in this section. Information on the input mapping construct is included under “Input Mapping Used with Retrieval Statements” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates the use of the SECURE statement to lock records of subset X.

```
BEGIN
   DATABASE DBASE;
   BOOLEAN RSLT;
   INTEGER MN;

   OPEN UPDATE DBASE;
   SET X TO BEGINNING;
   SECURE NEXT X :RSLT;
   WHILE NOT RSLT DO
      BEGIN
      GET D (MN := N);
      IF MN <= 10 THEN
         BEGIN
         REMOVE D FROM X;
         DELETE D;
         END
      ELSE
         BEGIN
         PUT D (B := TRUE);
         STORE D;
         END;
      SECURE NEXT X :RSLT;
      END;
   CLOSE DBASE;
END.
```

BDMS SET Statement

<BDMS set statement>

```
─ SET ─┬──┬─┬─ <set> ────┬─ TO ─┬─ <data set> ─┬─┬─<exception handling>─┬───┤
       │  │ └─ <subset> ─┘      ├─ BEGINNING  ─┤ │                      │
       │  │                     └─ ENDING  ────┘ │                      │
       │  └─ <data set> ─ TO ─┬─ BEGINNING ─┬────┘                      │
       │                      └─ ENDING  ───┘                           │
       └─ <item> ─ TO ─ NULL ───────────────────────────────────────────┘
```

<item>

```
  ─ <qualification>──────────────────────────────────────────────────────┤
```

Explanation

The BDMS SET statement alters the current path or changes the value of an item in the current record. Only the record area is affected. The data set is not affected until a subsequent STORE statement is executed.

The SET statement performs the following steps in order:

1. It frees the current path of the data set, set, or subset.
2. It performs one of the following actions:
   a. Alters the current path of the data set, set, or subset to point to the beginning or the ending of the indicated structure
   b. Alters the set or subset path to point to the current path of another data set
   c. Assigns a NULL value to a particular item

The constructs data set, set, or subset following the word “SET” specify the data set, set, or subset, respectively, whose path is altered.

If “TO data set” is specified, the current path of the set or subset is altered to point to the current record of the specified data set.

If “TO BEGINNING” is specified, the current path of the set, subset, or data set is altered to point to the beginning of the set, subset, or data set, respectively.

If “TO ENDING” is specified, the current path of the set, subset, or data set is altered to point to the ending of the set, subset, or data set, respectively.

The item construct specifies an item of the current record that is assigned a NULL value. The item cannot be a link item. NULL can be the DASDL-declared NULL value or the system default NULL value. Consult the *DMSII DASDL Programming Reference Manual* for more information.

After a SET TO BEGINNING form of the SET statement, the FIND NEXT and FIND FIRST forms of the FIND statement are equivalent; similarly, after a SET TO ENDING, a FIND PRIOR and FIND LAST are equivalent.

Additional information relating to the data set, set, and subset constructs is included under “Selecting a Record in a Data Set” in this section. Information on the qualification construct is included under “Qualification of Database Items” in this section. Information on the exception handling construct is included under “Database Status Word” in this section.

Examples

Assume a database named DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (20);
   B BOOLEAN;
   N NUMBER (2);
   R REAL;
   );
S SET OF D KEY (N);
SS SUBSET OF D WHERE (N = 3);
```

The following BDMSALGOL program demonstrates different ways to use the SET statement.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB,RSLT;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];
   LABEL CLOSE_DATABASE;

   OPEN INQUIRY DBASE;
   SET SS TO BEGINNING :RSLT;
   IF RSLT THEN
      BEGIN
      WRITE(PRINT_FILE,<"** NO ENTRIES IN SS. **">);
      GO CLOSE_DATABASE;
      END;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      SET SS TO D :RSLT;
      IF RSLT THEN
         WRITE(PRINT_FILE,<I3," NOT IN SS.">,MN)
      ELSE
         BEGIN
         GET D(MA[0] := A,MB := B);
         IF MB THEN
            GET D (MR := R)
         ELSE
            MR := 0;
         WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
               MN,MA[0],MB,MR);
         END;
      END;

CLOSE_DATABASE:
   CLOSE DBASE;
END.
```

STORE Statement

<store statement>

```
 ─ STORE ──┬─ <data set> ─────────────────┬─┬──────────────────────────┬─────────→
           │                              │ │                          │
           └─ <database identifier> ──────┘ └─ <exception handling> ──┘

→─┬──────────────────────────────┬────────────────────────────────────────────┤
  │                              │
  └─ ( ─ <output mapping> ─ ) ───┘
```

Explanation

The STORE statement places a new or modified record into a data set or a global record area. The data from the user work area for the data set or global record is inserted into the data set or global record area.

The STORE statement performs the following actions after a CREATE or RECREATE statement:

1. Check the data in the user work area for validity if a VERIFY condition is specified in the DASDL.
2. Test the record for validity for insertion in each set in the data set (for example, tests whether or not duplicates are permitted).
3. Evaluate the WHERE condition for each automatic subset.
4. Insert the record into all sets and automatic subsets if all conditions are satisfied.
5. Lock the new record.
6. Alter the data set path to point to the new record.

After a BDMS LOCK or MODIFY statement, the STORE statement performs the following actions:

1. Check the data in the user work area for validity if a VERIFY condition is specified in the DASDL.
2. Reevaluate the conditions if items involved in the insertion conditions have changed. If the condition yields FALSE, the record is removed from each automatic subset that contains the record. If the condition yields TRUE, the record is inserted into each automatic subset that does not contain the record.
3. Delete and reinsert the record in the proper position if a key used in the ordering of a set or automatic subset is modified so that the record must be moved within that set or automatic subset.
4. Store the record in a manual subset, but performs no reordering on that subset. The user is responsible for maintaining manual subsets. A subsequent reference to the record using that subset produces undefined results.

If the data set form is used, the data in the user work area for the data set is returned to the specified data set.

If the database identifier form is used, the data in the user work area for the global data is returned to the global data record area. The global data record must be locked before a STORE statement references it; otherwise, the STORE statement is terminated with an exception.

An exception is returned and the record is not stored if the record does not meet any of the validity conditions.

An exception is returned if the data set path is valid and the current record is not locked, or if the global data record is not locked.

Additional information relating to the data set construct is included under “Selecting a Record in a Data Set” in this section. Information on the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the exception handling construct is included under “Database Status Word” in this section. Information on the output mapping construct is included under “Output Mapping Used with Storage Statements” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (3);
   N NUMBER (3);
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the STORE statement can be used to place a record into the data set D.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY MY_A[0:2];
   INTEGER MY_N;

   OPEN UPDATE DBASE;
   MY_N := 1;
   WHILE MY_N < 100 DO
      BEGIN
      CREATE D;
      PUT D (N := MY_N);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R;
      MY_N := *+1;
      END;
   WHILE NOT READ(CARD_FILE,<I3,A3>,MY_N,MY_A[0]) DO
      BEGIN
      LOCK S AT N = MY_N;
      BEGINTRANSACTION R;
      PUT D (A := MY_A[0]);
      STORE D;
      ENDTRANSACTION R;
      END;
   CLOSE DBASE;
END.
```

BDMSALGOL Functions

There are two data management functions available in the BDMSALGOL language: DMTEST and STRUCTURENUMBER. These functions are described in this section.

DMTEST Function

<dmtest function>

```
— DMTEST  — (——┬— <alpha item> ————┬——┬— EQL  ——┬— NULL— ) ————————————————┤
               ├— <link item> ————┤  ├— = ——————┤
               ├— <numeric item> —┤  ├— IS  ————┤
               └— <real item> ————┘  ├— NEQ  ———┤
                                     ├— ^= ——————┤
                                     └— ISNT  ——┘
```

<alpha item>
<numeric item>
<real item>

```
  — <qualification>———————————————————————————————————————————————————————┤
```

Explanation

The DMTEST function determines whether an item is null. The function returns a Boolean value of TRUE or FALSE. It is TRUE if the value of the relationship expressed between the parentheses is TRUE; otherwise, it is FALSE. No status value is associated with the DMTEST function.

The alpha item construct specifies an alpha item declared in the DASDL. The alpha item contains a NULL value after a “SET item TO NULL” form of the BDMS SET statement, where item is the alpha item.

The numeric item construct specifies a numeric item declared in the DASDL. The numeric item contains a NULL value after a “SET item TO NULL” form of the BDMS SET statement, where item is the numeric item.

The real item construct specifies a real item declared in the DASDL. The real item contains a NULL value after a “SET item TO NULL” form of the BDMS SET statement, where item is the real item.

The link item construct specifies a link item declared in the DASDL. The link item contains a NULL value if either of the following is TRUE:

1. The link item does not point to a record.
2. No current record is present for the data set that contains the link item. This condition occurs following a BDMS OPEN statement, following the SET TO BEGINNING and SET TO ENDING forms of the BDMS SET statement, or when the record containing the link item has been deleted.
3. The use of a DMVERB against the structure where the link item points will cause a VERSIONERROR.

The link item contains a nonnull value if the link item points to a record, even if that record has been deleted.

The word “NULL” represents the DASDL-defined NULL value.

Additional information relating to the link item construct is included under “Selecting a Record in a Data Set” in this section. Information on the qualification construct is included under “Qualification of Database Items” in this section.

Examples

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
   S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the DMTEST function can be used to determine whether or not the alpha item A is NULL:

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      BEGIN
      LOCK S AT N = X;
      IF DMTEST(A ISNT NULL) THEN
         DELETE D
      ELSE
         FREE D;
      END;
   CLOSE DBASE;
END.
```

STRUCTURENUMBER Function

<structurenumber function>

```
— STRUCTURENUMBER  — (—┬— <database identifier> —┬— ) ——————————————————┤
                       ├— <data set> ————————————┤
                       ├— <set> —————————————————┤
                       └— <subset> ——————————————┘
```

Explanation

The STRUCTURENUMBER function allows the programmer to determine programmatically the structure number of a data set, set, subset, or of global data. This function can be used to analyze the result of exception conditions.

This capability is most useful when several sets span a data set and the previous operation against the data set yielded an exception. The program can determine which structure caused the exception from the corresponding structure number.

If the database identifier construct is used, the STRUCTURENUMBER function returns the structure number of the global data. Otherwise, the function returns the structure number of the data set, set, or subset specified by its respective construct.

When a partitioned structure is declared in DASDL, it is assigned one or more structure numbers, depending on unsigned integer in the “OPEN PARTITIONS = unsigned integer” form of the DASDL OPEN data set option. For example, if “OPEN PARTITIONS = 3” is specified, three structure numbers are assigned to the partitioned structure. Refer to the *DMSII DASDL Programming Reference Manual* for further information.

The STRUCTURENUMBER function returns the smallest structure number assigned to the structure; however, DMSTRUCTURE, the value in the exception status word, can evaluate to any of these values; that is, it does not necessarily evaluate to the same structure number every time.

Additional information relating to the database identifier construct is included under “BDMS CLOSE Statement” in this section. Information on the data set, set, and subset constructs is included under “Selecting a Record in a Data Set” in this section.

Example

```
REAL ERRORWORD;
IF STRUCTURENUMBER(D) = ERRORWORD.DMSTRUCTURE THEN
   REPLACE EA BY "D FAULT";
```

Exception Processing

When executing BDMSALGOL statements, any one of several exception conditions, which prevent the operation from being performed as specified, can be encountered. These conditions result if the operation encounters a fault or does not produce the expected action. For example, execution of the statement

```
FIND S AT NAME = "JONES"
```

would result in an exception if there is no entry in S that has a value of “JONES” for the key item. If the operation terminates normally, no exception occurs.

A database status word is returned to the BDMSALGOL program at the conclusion of each BDMSALGOL statement. The value of this word indicates whether or not an exception has occurred and specifies the nature of the exception.

Database Status Word

<exception handling>

```
 — : — <exception  variable> ————————————————————————————————————————————┤
```

<exception variable>

```
——┬— <Boolean variable> —┬————————————————————————————————————————————┤
  └— <real variable> ————┘
```

Explanation

In a BDMSALGOL statement, the user must specify the name of a real variable or Boolean variable in which the value of the database status word is stored at the completion of the BDMSALGOL statement. If no such variable is specified, the status value cannot be interrogated.

The exception handling construct is used in the syntax of the BDMSALGOL statements to denote those statements where a program variable can be designated to receive the value of the database status word.

A Boolean variable is a Boolean simple variable or an element of a Boolean array. A real variable is a real simple variable or an element of a real array.

For more information regarding Boolean and real variables, refer to Volume 1.

Example

```
REAL ERRORWORD;
OPEN UPDATE DBASE :ERRORWORD;
```

Exception Handling

<exception value>

```
 — <exception variable> — .─┬─ DMERROR ─────┬──────────────────────────┤
                            ├─ DMERRORTYPE ─┤
                            └─ DMSTRUCTURE ─┘
```

Explanation

If the database status word is treated as a Boolean quantity, its value is TRUE if the operation containing it results in an exception; otherwise, it is FALSE.

If an exception results from a database operation, but the value of the database status word is not assigned to an exception variable in the program, the program is terminated. If the value is assigned to an exception variable, no other indication of the exception is given. The BDMSALGOL program is responsible for determining the nature of the exception and responding appropriately. Failure to do so can cause unpredictable results.

To determine the nature of an exception, the database status word is interrogated by specifying a period (.) and an attribute name following the exception variable. The attribute names are recognized by the BDMSALGOL compiler as representations of the appropriate fields of the database status word.

The values that can be stored in the database status word are noted and explained in the *DMSII Application Program Interfaces Programming Guide*.

The DMERROR attribute yields a numeric value identifying a major category. Mnemonic names are also available to represent these numeric values. Either the category number or the category mnemonic can be used to test for a particular category.

The DMERRORTYPE attribute yields a numeric value identifying the subcategory of the major category.

The DMSTRUCTURE attribute yields a numeric value identifying the structure number of the structure involved in the exception. The structure numbers of all invoked structures are shown in the program listing if the program was compiled with the compiler control options LIST and LISTDB equal to TRUE.

Examples

The following example illustrates one way of handling exceptions in a BDMSALGOL program:

```
REAL ERRORWORD;
OPEN UPDATE DBASE :ERRORWORD;
IF BOOLEAN(ERRORWORD) THEN
   IF ERRORWORD.DMERROR = OPENERROR THEN
      IF ERRORWORD.DMERRORTYPE = 1 THEN
         DISPLAY("I/O ERROR ON ACCESSROUTINES CODE FILE");
```

If the exception variable is a Boolean variable, the preceding example is changed as follows:

```
BOOLEAN ERRORWORD;
OPEN UPDATE DBASE :ERRORWORD;
IF ERRORWORD THEN
   IF REAL(ERRORWORD).DMERROR = OPENERROR THEN
      IF REAL(ERRORWORD).DMERRORTYPE = 1 THEN
         DISPLAY("I/O ERROR ON ACCESSROUTINES CODE FILE");
```

BDMSALGOL Compiler Control Options

<datadictinfo option>

```
 — DATADICTINFO ———————————————————————————————————————————————————————————|
```

<listdb option>

```
 — LISTDB —————————————————————————————————————————————————————————————————|
```

<nodmdefines option>

```
 — NODMDEFINES ————————————————————————————————————————————————————————————|
```

Explanation

The above compiler control options are available in the BDMSALGOL language in addition to the options available in the ALGOL language. For information on the compiler control options available in ALGOL, refer to Volume 1.

(Type: Boolean, Default value: FALSE) If the DATADICTINFO option is TRUE, information about the usage of database structures and items is placed into the object code file. This information shows which database structures and items were invoked by the program and whether they were read or written. This option cannot be assigned a value after the appearance of the first syntactical item in the program.

(Type: Boolean, Default value: FALSE) If both the LIST option and the LISTDB option are TRUE, the printer listing contains information about the invoked databases, structures, and items, including the declared database titles. If the LIST option is TRUE but the LISTDB option is FALSE, the printer listing does not contain this information. The value of LISTDB is ignored if the LIST option is FALSE.

(Type: Boolean, Default value: FALSE) If the NODMDEFINES option is TRUE, no defines are expanded in BDMSALGOL constructs.

When the NODMDEFINES option is FALSE, defines in BDMSALGOL constructs are expanded, including defines in the following situations:

- A database item has the same identifier as a define.
- An alphanumeric string that is part of a database item identifier (between two hyphens, before the first hyphen, or after the last hyphen) is the same as the identifier of a define.

Binding and SEPCOMP of Databases

Programs that declare and use databases can use the Binder program and the separate compilation (SEPCOMP) facility of the compiler.

Binding

Programs that declare and reference databases can be bound together by the Binder program. The following example shows a BDMSALGOL host program that

- Declares a database
- Declares an external procedure
- Declares a separate procedure that is to be bound to the host
- Declares the database in its global part

The DASDL description of the database TESTDB is as follows:

```
DS DATA SET (
   NAME GROUP (
      LAST ALPHA (10);
      FIRST ALPHA (10);
      );
   AGE NUMBER (2);
   SEX ALPHA (1);
   SSNO ALPHA (9);
   );
NAMESET SET OF DS KEY (LAST, FIRST);
```

The following program, compiled with the name SEP/HOST, is the BDMSALGOL host program:

```
BEGIN
   DATABASE TESTDB;
   PROCEDURE P; EXTERNAL;
   OPEN UPDATE TESTDB;
   P;
   CLOSE TESTDB;
END.
```

The following separate procedure, P, compiled with the name SEP/P, is to be bound to the external procedure P of the host. Note how the database TESTDB is declared in the global part.

```
[DATABASE TESTDB;]
PROCEDURE P;
   BEGIN
   BOOLEAN EXCEPTIONWORD;
   EXCEPTIONWORD := FALSE;
   SET NAMESET TO BEGINNING;
   WHILE NOT EXCEPTIONWORD DO
      BEGIN
      FIND NEXT NAMESET AT LAST = "SMITH"
         AND FIRST = "JOHN" :EXCEPTIONWORD;
      % Other statements
      END;
   END;
```

The separate procedure P in SEP/P can be bound to the host SEP/HOST using the following Work Flow Language (WFL) job. The resulting bound code file is named GLOBDB.

```
?BEGIN JOB BIND/GLOB;
 BIND GLOBDB WITH BINDER LIBRARY;
 BINDER DATA
 HOST IS SEP/HOST;
 BIND P FROM SEP/P;
?END JOB.
```

Note: *The description of the database in each subprogram that is called upon must exactly match the invocation of the database in the host program to which you want the subprogram to be bound. The host file, and all the subprograms that access the database in these environments, must be compiled using the same database description file. If you fail to adhere to these requirements, you receive syntax errors when you attempt to combine the various components by using Binder.*

SEPCOMP

Programs that declare and use databases can also make use of the SEPCOMP facility of the compiler, as shown in the following example.

The DASDL description of the database TESTDB is as follows:

```
DS DATA SET (
   NAME GROUP (
      LAST ALPHA (10);
      FIRST ALPHA (10);
      );
   AGE NUMBER (2);
   SEX ALPHA (1);
   SSNO ALPHA (9);
   );
NAMESET SET OF DS KEY (LAST, FIRST);
```

Because the MAKEHOST compiler control option is TRUE, the following program, compiled as MY/HOST, can be used as a host program for SEPCOMP:

```
$ SET MAKEHOST
BEGIN                                                       1
   DATABASE TESTDB;                                         2
   PROCEDURE P;                                             3
      BEGIN                                                 4
      BOOLEAN EXCEPTIONWORD;                                5
      EXCEPTIONWORD := FALSE;                               6
      SET NAMESET TO ENDING;                                7
      WHILE NOT EXCEPTIONWORD DO                            8
         BEGIN                                              9
         FIND NEXT NAMESET AT LAST = "SMITH"               10
            AND FIRST = "JOHN": EXCEPTIONWORD;             11
         % Other statements                                12
         END;                                              13
      END;                                                 14
   OPEN UPDATE TESTDB;                                     15
   P;                                                      16
   CLOSE TESTDB;                                           17
END.                                                       18
```

The following source input invokes the SEPCOMP facility to change the record of the host MY/HOST with sequence number 7, recompile the procedure P, and bind the new P to the host:

```
$ SET SEPCOMP "MY/HOST"          % Patch follows
      SET NAMESET TO BEGINNING;                             7
```

Section 5
Using DMSII Transaction Processing System (TPS) Extensions

The Transaction Processing System (TPS) provides Data Management System II (DMSII) users the software means to process a high volume of transactions. TPS separates into modules the various functions needed to perform database processing. TPS also supplies a library of transaction processing procedures. By using TPS, the DMSII user can

- Minimize program coding and maintenance.
- Eliminate much of the complexity that characterizes programming for database processing.
- Centrally define all transactions to be performed against a database.
- Rely on comprehensive recovery capabilities.

Basically, there are two types of programs you can write for TPS:

- The application program, which can call Transaction Library points to invoke library procedures.
- The Update Library, which is a collection of transaction-processing routines that provide an interface between the Transaction Library and a DMSII database.

Consult the *DMSII Transaction Processing System (TPS) Programming Guide* for a thorough discussion of TPS, its modules and libraries, and its associated Transaction Formatting Language (TFL). Pertinent information about the DMSII and BDMSALGOL interface can be found in this volume.

The TPS program interface consists of extensions that provide access to a transaction base. You can

- Invoke a transaction base.
- Create transaction records.
- Use transaction records to pass variables as parameters and to assign (or copy) the contents of a variable to another transaction record variable.
- Access transaction record items.
- Inquire about transaction record control items.
- Use transaction record compile-time functions to access certain properties of transaction record formats.
- Use Transaction Library entry points to invoke library procedures.
- Use the Update Library to perform data management of the database with transaction processing statements.

Sample ALGOL programs at the end of this section demonstrate the uses of the TPS interface.

The ALGOL compiler enforces all restrictions on the use of transaction record variables noted in this section and, when appropriate, issues syntax errors.

Additional information relating to DMSII transactions is included in Section 4, “Using the Data Management System II (DMSII) Interface.”

Using the Transaction Formatting Language (TFL)

Transaction Formatting Language (TFL) is a symbolic language used to define information related to transaction processing. The symbolic descriptions of transaction record structures are collectively referred to as a transaction base. Consult the *DMSII TPS Programming Guide* for a complete description of TFL.

Table 5–1 shows what type must be declared for each TFL item in ALGOL application programs that access a transaction base. In the listing, name is the declared item name. For ALPHA and FIELD TFL items, “n” is the length. For all other items, “n” is an unsigned integer, “Sn” is a signed integer, and “m” is a decimal.

Table 5–1. TFL Item Interpretations

TFL Item	ALGOL Type
<name> ALPHA(n)	STRING <name>
<name> NUMBER(n)	INTEGER <name>
<name> NUMBER(Sn)	INTEGER <name>
<name> NUMBER(n,m)	REAL <name>
<name> NUMBER(Sn,m)	REAL <name>
<name> REAL	REAL <name>
<name> REAL(n)	INTEGER <name>
<name> REAL(Sn)	INTEGER <name>
<name> REAL(n,m)	REAL <name>
<name> REAL(Sn,m)	REAL <name>
<name> BOOLEAN	BOOLEAN <name>
<name> FIELD(n)	REAL <name>
<name> GROUP	STRING <name>

Declaring a Transaction Base

```
— TRANSACTION BASE — <base spec> ——┬—————————————————————┬— ; ——————————┤
                                   └ : — <format list> ——┘
```

<base spec>

```
—┬————————————————————————┬—┬—————————————————————————┬— <base name> ——┤
 └ <internal base ID> — = ┘ └ <subbase name> — OF ————┘
```

<format list>

```
 ┌←——————————————————————— , ———————————————┐
—┴ <format spec> ——┬—————————————————————————┬┴—————————————————————————┤
                   └ ( — <subformat list> — )┘
```

<format spec>

```
—┬———————————————————————————┬— <format name> ——————————————————————————┤
 └ <internal format ID> — = —┘
```

<subformat list>

```
—┬— ALL ——————————————┬—————————————————————————————————————————————————┤
 ├— NONE —————————————┤
 │  ┌←———————— , ————┐│
 └——┴ <subformat spec> ┴┘
```

<subformat spec>

```
—┬——————————————————————————————┬— <subformat name> ———————————————————┤
 └ <internal subformat ID> — = —┘
```

Explanation

Before making any references to formats or items defined within a transaction base, a user-written program must declare that transaction base. In the declaration, you can

- Specify only the transaction base and, by default, invoke all structures in the transaction base.
- Optionally specify a list of transaction record formats and subformats to invoke only those structures of the transaction base.

Any program that invokes the Transaction Library should not be a library itself.

The program can also specify alternate internal names for the transaction base and for any of the formats or subformats declared. If alternate internal names are used for the base name, subbase name, format name, or subformat name, the program must reference these internal identifiers rather than the TFL source identifiers.

If a subbase has been defined for the transaction base, the program can also invoke the subbase. When a subbase is invoked, only the transaction record formats and subformats defined within that subbase are accessible to the program. As in transaction base invocation, the program can specify a list of transaction record formats and subformats, possibly using internal names that can be invoked from the defined subbase.

The syntax “TRANSACTION BASE base spec” specifies the name of the transaction base or subbase to be invoked. Optionally, a list of transaction record formats and subformats can be invoked. If the list is not included, all transaction record formats and subformats are invoked for the designated transaction base or subbase. If the list is included, only the indicated transaction record formats and subformats are invoked.

The different forms of the base spec construct specify either the transaction base or subbase. The syntax “internal base ID=base name” is used to invoke a transaction base with the designated internal name. The syntax “subbase name OF base name” designates the name of a transaction subbase to be invoked.

The format list is a list of transaction record format and subformat names including, possibly, internal names. If only format name is specified in the format spec syntax, by default all subformats of that format are invoked.

When a subformat list is used in a format spec construct, it specifies the name of the transaction record format being invoked. If the “internal format ID=format name” syntax is used, it specifies its internal name, an indication of the transaction record subformats to be invoked, or both.

A subformat list indicates the specific subformats of a transaction record format. If no subformat list is included for a particular format name, ALL is assumed.

If a transaction base with a list of formats has been invoked, specifying ALL invokes all the subformats of that format. If a transaction subbase has been invoked, specifying ALL invokes only those subformats specified for this format in the TFL subbase declaration.

If only a format name is listed in the TFL subbase declaration, then by default TFL includes all subformats of the format in the subbase declaration.

If NONE is specified for a particular transaction format, then no subformats are invoked.

If a list of subformat specs is specified, only those subformats on the list are invoked. If a transaction subbase is invoked, then subformat specs can include only those subformats defined within the transaction subbase for a particular format.

Examples

In the following example, the transaction base BANKACCT is invoked. Since no format list is invoked, all transaction record formats and subformats are also invoked.

```
TRANSACTION BASE BANKACCT;
```

As seen in the following example, the transaction base MANUFACT is equated to the internal name MNF and invoked. All transaction record formats and subformats are invoked.

```
TRANSACTION BASE MNF = MANUFACT;
```

In the following example a transaction base with the internal base identifier DOC1 is equated to DOC and invoked. The format list includes several formats with subformat lists.

IFMT1, IFMT6, IFMT3, IFMT4, and IFMT5 are internal format identifiers that are each equated to a format name.

The ALL option specifies that all the subformats of IFMT3 are invoked. Because neither NONE nor a specific subformat is noted, any subformats of FMT0, IFMT1, and IFMT6 are also invoked. (The default is ALL.)

The NONE option specifies that none of the subformats of IFMT4 are invoked. Only the subformats S1 and IS3 are invoked for IFMT5. The internal subformat identifier IS3 is equated to the subformat S3.

```
TRANSACTION BASE DOC1 = DOC :
   FMT0,
   IFMT1 = FMT1,
   IFMT6 = FMT6,
   IFMT3 = FMT3 (ALL),
   IFMT4 = FMT4 (NONE),
   IFMT5 = FMT5 (S1,IS3 = S3);
```

Creating Transaction Records

A transaction record is an array row that can contain the transaction data of one of several transaction formats declared in the TFL source. A transaction record variable names one of these array rows. A transaction record variable can contain the transaction data of one of several transaction formats and can make the transaction record, in effect, a structured variable.

A transaction record variable can be associated with only one transaction base or transaction subbase. A transaction record variable can contain only formats and subformats that have been invoked from its associated transaction base or transaction subbase. The size of the array row is large enough to accommodate the largest of all the formats invoked for it.

The following information explains how transaction record variables are declared and how transaction records are created.

Declaring Transaction Record Variables

<transaction record declaration>

```
──┬──────┬─ TRANSACTION RECORD ── ( ── <base ID> ── ) ──────────→
  └ LONG ┘
  ┌←─────────────── , ──────────┐
→─┴─ <transaction variable ID> ─┴─ ; ──────────────────────────┤
```

<transaction record array declaration>

```
 ─ TRANSACTION RECORD ARRAY ─ ( ─ <base ID> ─ ) ────────────────────→

  ┌←─────────────────────────── , ──────────────────────────────┐
→─┴─ <transaction array ID list> ─ [ ─ <bound pair list> ─ ] ───┴─ ; ────┤
```

<transaction array ID list>

```
  ┌←──────────── , ──────────┐
──┴─ <transaction array ID> ─┴──────────────────────────────────────┤
```

<bound pair list>

```
  ┌←─────────────────────────────── , ──────────────────────────────┐
──┴─ <arithmetic expression> ─ : ─ <arithmetic expression> ─────────┴────────┤
```

Explanation

The transaction record can be declared as a one-dimensional or a two-dimensional array. Use the transaction record declaration syntax to declare a one-dimensional array. Use the transaction record array syntax to declare a two-dimensional array.

The transaction record declaration syntax is used with one-dimensional arrays. The transaction record array declaration syntax is used with two-dimensional arrays.

The option LONG suppresses the segmentation of transaction records. Ordinarily, transaction records larger than 1024 words are segmented into 512-word entities. (This segmentation is standard for all ALGOL arrays declared to have more than 1024 elements.)

The base ID construct is the name, or internal name, of a transaction base or transaction subbase. Specifying a base ID in the declaration of a transaction record or transaction record array associates a transaction base or transaction subbase with the particular record or records.

The transaction variable ID construct identifies the name of a transaction record variable. The transaction array ID construct identifies the name of an array of transaction record variables. Each fully subscripted element of a transaction array ID is a transaction record.

Additional information relating to transaction record variables is included under “Inquiring About Transaction Record Control Items,” “Passing Transaction Record Variables as Parameters,” and “Accessing Transaction Record Items ” in this section.

The bound pair list construct gives the lower and upper bounds of all subscripts taken in order from left to right.

Refer to Volume 1 for information about arithmetic expressions and bound pair lists.

Examples

In the following example, the transaction variables TRIN, TROUT, LASTINPUT, and LASTRESPONSE are associated with the transaction base BANKACCT.

```
TRANSACTION RECORD (BANKACCT)
   TRIN,
   TROUT,
  LASTINPUT,
   LASTRESPONSE;
```

In the next example, the LONG option suppresses segmentation for the transaction records in the transaction base DOC.

```
LONG TRANSACTION RECORD (DOC);
```

In the following example, the transaction base DOC is an array. TRARRAY1, TRARRAY2, and TRARRAY3 are transaction array identifiers. The lower and upper bounds of TRARRAY2 are 0 and 9, respectively. The lower and upper bounds of TRARRAY3 are 0 and 0, respectively.

```
TRANSACTION RECORD ARRAY (DOC)
   TRARRAY1,
   TRARRAY2 [0:9],
   TRARRAY3 [0:0];
```

Creating Transaction Record Formats

<create statement>

```
── CREATE ── <transaction record> ── . ── <format ID> ────────────────→
→─┬───────────────────┬─ ; ────────────────────────────────────────────┤
  └─ . ── <subformat ID> ┘
```

<transaction record>

```
─┬─ <transaction record variable ID> ─────────────────────┬───────────┤
 └─ <transaction record array ID> ─ [ ─ <subscript list> ─ ] ─┘
```

<subscript list>

```
   ┌←────── , ─────┐
 ──┴─ <subscript> ─┴──────────────────────────────────────────────────┤
```

Explanation

The contents of a transaction record variable are undefined until the variable is initialized to a particular format by a CREATE statement. A CREATE statement assigns the initial values of all items in the transaction record format (and transaction subformat) to the record variable and initializes the transaction record control items.

When a format is created, only those items in the common part are assigned initial values. When a subformat is created, the common part items as well as the subformat part items are assigned initial values. The record variable continues to contain the given format until it is reinitialized by a subsequent CREATE statement. It is never cleared by the system.

Once a transaction record variable has been created in a particular transaction format and, optionally, subformat, the items defined within the format and subformat can be accessed and manipulated. If a transaction record is created in a particular transaction record format, the record contains only the data items associated with that transaction record format. If a transaction record is created in a particular transaction record format and subformat, then the record contains the data items associated with the format and the data items associated with the subformat.

The transaction record construct is the name of the transaction record variable to be initialized. If a transaction record array element is referenced, it must be fully subscripted.

Additional information relating to transaction record formats is included under “Declaring Transaction Record Variables” and “Requirements for Data Item Qualification” in this section. Related information is also available under “CREATE Statement” in Section 4, “Using the Data Management System II (DMSII) Interface.”

The format ID and subformat ID constructs are, respectively, the names of the format and subformat (if given) whose data item's initial values are assigned to the record variable.

The subscript list construct gives one or more subscripts that are required to qualify the referenced item. In this syntax, it is a transaction record array ID. The subscript form is defined as any legitimate ALGOL arithmetic expression. Refer to Volume 1 for further details on subscripting.

Examples

In this example, the transaction record variable TRIN is initialized. The data items of the format ACCT are assigned to TRIN.

```
CREATE TRIN.ACCT;
```

Below, the transaction record variable TRRECORD is initialized. The data items of the format ACCT and the subformat MAY are assigned to TRRECORD.

```
CREATE TRRECORD.ACCT.MAY;
```

As seen in the following example, the transaction record array TRARRAY1 is initialized. It has a subscript of 7. The data items of the format ACCT are assigned to TRARRAY1.

```
CREATE TRARRAY1[7].ACCT;
```

Using Transaction Records

The compiler enforces certain restrictions on the use of transaction record variables. Transaction record variables can be used only as shown below.

- To create transaction records and use compile-time and run-time functions
- To store data in transaction records
- To obtain data from transaction records
- To pass transaction records as parameters in procedures

Transaction record variables cannot be used

- In lists
- In input or output statements
- In assignment statements except as described in “Assigning Transaction Record Variables”

Additional information relating to transaction records is included under “Assigning Transaction Record Variables” in this section.

Passing Transaction Record Variables as Parameters

In transaction processing, most of the work is carried out by Transaction Library procedures. Transaction records are passed to these procedures as parameters. Transaction records cannot be passed to intrinsics or to external procedures initiated through a CALL or PROCESS statement.

The formal and actual parameters must refer to the same transaction base, but they need not specify the same list of transaction record formats. If a procedure is given a transaction record in a format it has not invoked, the procedure is limited as to what it can do with that record.

The transaction base or subbase must be declared before specifying the syntax for a formal transaction record variable.

The compiler checks that all the uses of a particular transaction record variable within a code file are compatible. When the variable is passed as a parameter to a separately compiled code file (such as the Transaction Library), parameter checking code ensures that the following attributes of the variable are those that are expected:

- Transaction record format level
- Transaction record control item length
- Transaction base creation date-time stamp

The contents of the variable need not be inspected to make this check. If any of these three attribute values do not match, the error message “MISMATCH AT PARAMETER NUMBER <number>, TRANSACTION RECORD <attribute>s DIFFER” is issued when an attempt is made to call a separately compiled code file.

Additional information relating to transaction record variables is included under “Declaring Transaction Record Variables” and “Using Transaction Library Entry Points” in this section.

Assigning Transaction Record Variables

```
— <transaction record> — := — <transaction record> ————————————————————————┤
```

Explanation

The contents of a transaction record variable can be assigned (that is, copied) to another transaction record variable, provided that both variables represent the same transaction base. Both the control and data portions of the transaction record are transferred when an assignment is performed.

In DMALGOL, the implementation language for the Transaction Library, an ARRAY reference variable can be assigned to a transaction record variable. This construct is not permitted in user-written programs. Consult the *DMALGOL Programming Reference Manual* for more information on DMALGOL.

The construct transaction-record-1 is the name of the transaction record variable that receives its contents from another transaction record variable.

The construct transaction-record-2 is the name of the transaction record variable whose contents are being assigned or copied to another transaction record variable.

Additional information relating to transaction record variables is included under “Inquiring About Transaction Record Control Items” in this section.

Example

The contents of the transaction record TRRECORD are assigned to the transaction record TRRECEIVE.

```
  TRRECEIVE := TRRECORD;
```

Accessing Transaction Record Items

<item reference>

```
 – <transaction record>──┬────────────────────┬┬──────────────────────────┬─ . →
                         │                    ││                          │
                         └─ . – <format ID> ──┘└─ . – <subformat ID> ─────┘

 →– <item name> ─┬────────────────────────────┬──────────────────────────────┤
                 │                            │
                 └─ [ – <subscript list> – ] ─┘
```

<item name>

```
─┬─ <group item name> ───┬──────────────────────────────────────────────┤
 ├─ <alpha item name> ───┤
 ├─ <Boolean item name> ─┤
 ├─ <numeric item name> ─┤
 ├─ <real item name> ────┤
 └─ <field item name> ───┘
```

Explanation

A transaction record can contain only a transaction that has a format and subformat declared for it in the TFL source. Data items in the declared format and subformat of that transaction can be referenced.

Transaction record data items are considered normal data items and can be referenced in the same manner as normal data items.

The construct transaction record is used to name a transaction record variable. If a transaction record array element is used, it must be fully subscripted.

The format ID and subformat ID constructs are normally optional. However, they can be required for qualification.

The item name construct specifies an item within the transaction record format or subformat presently occupying the record variable. The item name must be fully subscripted if it is an element of an occurring item.

An item name can be used either as the left part of an assignment or REPLACE statement or as a primary in an expression. The type of the item must be consistent with the context in which it is used.

Data items of transaction record formats or subformats that are occurring items, items embedded within one or more occurring groups, or items that occur and are embedded within occurring groups must be subscripted. The subscripts within a subscript list construct are listed from left to right, from the outer most occurring GROUP to the innermost occurring GROUP or occurring items.

Additional information relating to transaction record items is included under "Using the Transaction Formatting Language (TFL)" and "Requirements for Data Item Qualification" in this section.

Examples

In the example, the item GR is within the transaction record format TRONE. The content of the transaction record MANUFACT is assigned to the item GR.

```
TRONE.GR := MANUFACT;
```

The next example contains a subscript construct. The item ST is qualified by the format GENLED and the subformat JONO. ST is an occurring item within the transaction record format TRTWO. The content of AX is copied to the item.

```
TRTWO.GENLED.JONO.ST[9] := AX;
```

Requirements for Data Item Qualification

A data item is qualified in order to make it unique or to differentiate it from other similar items. Use qualification to assure that the items referenced are the desired data items.

The amount of qualification required to access a data item of a particular transaction record format or subformat varies. In every case, however, the transaction record variable containing the desired data item must be referenced.

Shown below are the varying requirements and syntaxes for qualification. The following tokens are used in the syntaxes.

Token	Name
DATAITEMNAME	Data item name
FORMATNAME	Format name
SUBFORMATNAME	Subformat name
TRANREC	Transaction record

Data Item Qualification

If the name of the desired data item is unique with respect to data items of other invoked

Example

```
TRANREC.DATAITEMNAME
```

Format Name and Data Item Name Qualification

If the name of the desired data item is not unique with respect to data items of other invoked formats, but is unique to the format that contains it, specify both the format and the data item name.

Example

```
TRANREC.FORMATNAME.DATAITEMNAME
```

Subformat Name and Data Item Name Qualification

Specify both the subformat name and the data item name whenever any of of the following are true:

- The name of the desired data item is not unique with respect to the common portion of another invoked format.
- The name of the desired data item is contained within a subformat.
- Another data item within a different subformat of the same format has the same name as the desired data item.

Also, if the desired data item is contained within a subformat whose name is unique to all invoked formats and subformats, and the desired data item is not unique with respect to a subformat of another format, then both the subformat name and the data item name are needed.

Example

```
TRANREC.SUBFORMATNAME.DATAITEMNAME
```

Format Name, Subformat Name, and Data Item Name Qualification

When all the following statements are true, specify the format name, subformat name, and data item name for qualification.

- The desired item is not unique with respect to a subformat of another invoked format.
- The item is not unique with respect to the format that contains it.
- The name of the subformat that contains the desired item is not unique with respect to all invoked formats and subformats.

Example

```
TRANREC.FORMATNAME.SUBFORMATNAME.DATAITEMNAME
```

Inquiring About Transaction Record Control Items

```
— <transaction record ID> ─┬──────────────────┬─ . — <record control item> ┤
                           └─ [<subscript>] ──┘
```

Explanation

Control items are system-defined items contained within every transaction record. These items are maintained by the TPS and are read-only in all BDMSALGOL programs. The initial values of these control items are assigned when a transaction record is created. These items are defined only after a transaction record has been created using the TPS CREATE statement.

The transaction record ID construct is a transaction record variable. The variable must be fully subscripted if a transaction record array element is used.

A subscript is an ALGOL arithmetic expression that identifies a particular transaction record variable within an array of transaction record variables.

The record control item construct identifies the specific control item. The valid items are described in the *DMSII TPS Programming Guide*.

Additional information relating to transaction record control items is included under "Creating Transaction Record Formats," "Assigning Transaction Record Variables," and "Declaring Transaction Record Variables" in this section.

Example

In the following example, the record control item TRCONTROLSIZE is used to specify the size, in bytes of the control portion of the transaction record TRIN. The content is assigned to the variable STOREBYTES.

```
STOREBYTES := TRIN.TRCONTROLSIZE;
```

Using Transaction Compile-Time Functions

<transaction compile-time functions>

```
— <transaction compile-time function name> — ( ——————————————————————————→

→— <transaction compile-time function argument> — ) —————————————————————|
```

<transaction compile-time function argument>

```
─┬─────────────────┬─ <format ID> ─┬─────────────────────┬───────────────→
 └─ <base ID> — . ─┘               └─ . — <subformat ID> ─┘

→─┬──────────────────────────────┬───────────────────────────────────────|
  └─ . — <transaction item ID> ─┘
```

Explanation

Transaction compile-time functions provide access to certain properties of transaction record formats that are constant at compile time. These compile-time constructs are particularly useful when coding an Update Library.

The transaction compile-time function names are identified and described in the *DMSII TPS Programming Guide.*

The constructs base ID, format ID, subformat ID, and transaction item ID are all components of the transaction compile-time function argument. The base ID is the name of a transaction base that has been invoked within the program. The format ID specifies the name of a transaction format that has been invoked within the program. A subformat ID is the name of a transaction subformat that has been invoked within the program. The transaction item ID is the name of a data item contained within an invoked transaction format or subformat.

The table below identifies the possible arguments for each of the compile-time functions available in ALGOL. Not all arguments apply to all functions. For example, the base ID construct needs to be referenced only when transaction base qualification is required.

Additional information relating to compile time constructs is discussed in “Using Update Libraries” located in this section.

Function	Arguments
TRBITS	<format ID>.<transaction item ID> <format ID>.<subformat ID>.<transaction item ID>
TRBYTES	<format ID>.<transaction item ID> <format ID>.<subformat ID>.<transaction item ID>
TRDATASIZE	<format ID> <format ID>.<subformat ID>
TRDIGITS	<format ID>.<transaction item ID> <format ID>.<subformat ID>.<transaction item ID>
TRFORMAT	<format ID>
TROCCURS	<format ID>.<transaction item ID> <format ID>.<subformat ID>.<transaction item ID>
TRSUBFORMAT	<format ID>.<subformat ID>

Examples

TRBITS will return, in bits, the size of the transaction item REQUESTCASE. The subformat is REMOTEREQUEST and the format is ACCT.

```
TRBITS(ACCT.REMOTEREQUEST.REQUESTCASE)
```

TROCCURS will return the maximum number of occurrences of the transaction item REQUESTCASE. ACCT is the format and REMOTEREQUEST is the subformat.

```
TROCCURS(ACCT.REMOTEREQUEST.REQUESTCASE)
```

TRSUBFORMAT will return the numeric valued assigned to the subformat REMOTEREQUEST. The format is ACCT.

```
TRSUBFORMAT(ACCT.REMOTEREQUEST)
```

Using Transaction Library Entry Points

The Transaction Library is a collection of procedures that are accessed by user-written programs to process or tank transactions and read them back from transaction journal files. The procedures are accessed through a set of entry points supplied by the Transaction Library.

The Transaction Library is tailored for a particular transaction during compilation. The library performs functions such as

- Calling the Update Library to process a transaction against the data base
- Saving transaction records in transaction journal files
- Automatically reprocessing transactions backed out by DMSII recovery

The external entry points to the Transaction Library are called by user- written programs. Calling these entry points is the only method of invoking them. If the Library detects an exception condition, the entry point returns a nonzero result as the value of the procedure. The value can be examined to determine the cause of the exception.

The TPS application program should not be a library itself whose entry points invoke the Transaction Library's entry points.

The Transaction Library recovery mechanism requires that each program that submits a transaction record for processing must have its own private library. The first program that invokes an entry point which in turn invokes the OPENTRBASE Transaction Library entry point becomes the only TPS user recognized by the Transaction Library.

The following alphabetical listing briefly describes the purpose of each entry point. The syntax used to declare the entry point is shown. Consult the *DMSII TPS Programming Guide* for a detailed explanation of the entry points and parameters.

CREATETRUSER

Creates and identifies a new transaction for the currently open journal.

```
INTEGER PROCEDURE CREATETRUSER(IDSTRING,IDNUM);
      STRING IDSTRING;
      STRING IDNUM;
```

CLOSETRBASE

Ends the use of TPS by the calling program.

```
INTEGER PROCEDURE CLOSETRBASE;
```

HANDLESTATISTICS

Enables the user to print out all TPS statistics and reset the statistics while the transaction base is open.

```
INTEGER PROCEDURE HANDLESTATISTICS(STATOPTION);
     VALUE STATOPTION;
     INTEGER STATOPTION;
```

LOGOFFTRUSER

Deactivates a transaction user.

```
INTEGER PROCEDURE LOGOFFTRUSER(IDNUM);
     INTEGER IDNUM;
```

LOGONTRUSER

Makes a transaction user active.

```
INTEGER PROCEDURE LOGONTRUSER(IDSTRING, IDNUM);
 STRING IDSTRING; INTEGER IDNUM;
```

OPENTRBASE

Initiates transaction processing and opens a specified transaction journal for subsequent use. OPENTRBASE must be the first Transaction Library entry point called.

```
INTEGER PROCEDURE OPENTRBASE(USEROPTION, TIMEOUT);
     INTEGER USEROPTION, TIMEOUT;
```

PROCESSTRFROMTANK

Similar to PROCESSTRANSACTION except a transaction user number other than that of the input transaction is used to restart programs. It is used primarily for processing transactions from a tank file.

```
INTEGER PROCEDURE PROCESSTRFROMTANK(IDNUM, TRIN, RESTARTNUM,
  RESTARTTR);
     INTEGER IDNUM, RESTARTNUM;
     TRANSACTION RECORD (TRBASE) TRIN, RESTARTTR;
PROCESSTRNORESTART
```

Sends an input transaction record to the user's Update Library for processing against the database. No restart transaction record is passed. Use PROCESSTRNORESTART to process transactions against the database if the program does not require the use of a restart transaction record.

```
INTEGER PROCEDURE PROCESSTRNORESTART(IDNUM, TRIN, TROUT);
     INTEGER IDNUM;
     TRANSACTION RECORD (TRBASE) TRIN, TROUT;
```

PROCESSTRANSACTION

Sends an input transaction record to the user's Update Library for processing against the database. A restart transaction record is passed.

```
INTEGER PROCEDURE PROCESSTRANSACTION(IDNUM, TRIN, TROUT,
  RESTARTTR);
     INTEGER IDNUM;
     TRANSACTION RECORD (TRBASE) TRIN, TROUT, RESTARTTR;
```

PURGETRUSER

Purges or deletes a transaction user previously created by CREATETRUSER. After PURGETRUSER is called, the transaction user is no longer known to the currently open journal. Information about that user's transactions is discarded.

```
INTEGER PROCEDURE PURGETRUSER(IDNUM);
     INTEGER IDNUM;
```

READTRANSACTION

Reads the next transaction record in sequence from the transaction journal and returns the record in the parameter TRREC. The READTRANSACTION entry point can be called only after the entry point SEEKTRANSACTION has opened and positioned the current record pointer within a specific journal data file.

```
INTEGER PROCEDURE READTRANSACTION(TRREC);
     TRANSACTION RECORD (TRBASE) TRREC;
```

RETURNLASTADDRESS

Returns the address of the last transaction to be either tanked or processed by a transaction user.

```
INTEGER PROCEDURE RETURNLASTADDRESS(FILENUM, BLOCKNUM, OFFSET,
  IDNUM);
     REAL FILENUM, BLOCKNUM, OFFSET;
     INTEGER IDNUM.
```

RETURNLASTRESPONSE

Returns the last saved response transaction record for the user.

```
INTEGER PROCEDURE RETURNLASTRESPONSE(IDNUM, TRREC);
     INTEGER IDNUM;
     TRANSACTION RECORD (TRBASE) TRREC;
```

Note: *For reliable program restarting, the response record returned from RETURNLASTRESPONSE should be used in conjunction with the restart or input transaction record returned from the entry point RETURNRESTARTINFO.*

RETURNSTARTINFO

Helps restart a user-written program.

```
INTEGER PROCEDURE RETURNRESTARTINFO(IDNUM, TRREC);
     INTEGER IDNUM;
     TRANSACTION RECORD (TRBASE) TRREC;
```

SEEKTRANSACTION

Positions a current record pointer at a particular address within a journal data file.

```
INTEGER PROCEDURE SEEKTRANSACTION(TRFILE, TRBLOCK, TROFFSET);
     INTEGER TRFILE, TRBLOCK, TROFFSET;
```

SWITCHTRFILE

Forces a file switch on the current data file of the journal. The current file is closed, the file number associated with the current file is incremented by 1, and the next file in sequence is created. The next write to the journal occurs on the new file.

If SWITCHTRFILE is not called, the Transaction Library creates the next journal data file in sequence when the current file becomes full.

```
INTEGER PROCEDURE SWITCHTRFILE;
```

TANKTRANSACTION

Tanks an input transaction record and restart transaction record.

```
INTEGER PROCEDURE TANKTRANSACTION(IDNUM, TRIN, RESTARTTR);
     INTEGER IDNUM;
     TRANSACTION RECORD (TRBASE) TRIN, RESTARTTR;
```

TANKTRNORESTART

Tanks an input transaction record only. It performs the same function as TANKTRANSACTION except that no restart transaction record is passed and subsequently audited in the tank journal. For TANKTRNORESTART, only the input transaction TRIN is saved in the tank journal.

```
INTEGER PROCEDURE TANKTRNORESTART(IDNUM,TRIN);
     INTEGER IDNUM;
     TRANSACTION RECORD (TRBASE) TRIN;
```

TRUSERIDSTRING

Returns, in the parameter IDSTRING, the user identification string that corresponds to the value of the input parameter IDNUM.

```
INTEGER PROCEDURE TRUSERIDSTRING(IDSTRING, IDNUM);
     INTEGER IDNUM;
     STRING IDSTRING;
```

Using Update Libraries

The Update Library is a collection of user-written transaction processing routines that serve as an interface between the Transaction Library and a DMSII database.

The Update Library is the only user-written module within TPS that contains the database declaration and all the code that performs data management statements against the database.

To ensure effective interaction between the Update and Transaction Libraries, follow the conventions regarding database consistency and reproducing transactions when programming the Update Library. The Update Library conventions and ACCESSDATABASE entry point are briefly explained here. For a full explanation, refer to the *DMSII TPS Programming Guide.*

Additional information relating to the transaction library is included under “Using Transaction Library Entry Points” in this section.

ACCESSDATABASE Entry Point

The Update Library must provide one entry point that makes it accessible to the Transaction Library. For ALGOL Update Libraries, the procedure entry point must be named ACCESSDATABASE.

The ACCESSDATABASE entry point accepts the following parameters, listed in the order in which they must be declared:

1. A function flag indicating which basic function the Update Library should perform. This value is input to the Update Library from the Transaction Library.
2. An input transaction record containing input data for one of the transaction update routines.
3. An output transaction record containing the data output from a transaction update routine also known as the “response transaction record.”
4. A Transaction Library procedure named SAVEINPUTTR that is passed as a formal parameter to the Update Library and used in the MIDTRANSACTION statement.

A Transaction Library procedure named SAVERESPONSETR that is passed as a formal parameter to the Update Library. This procedure is used in the TPS ENDTRANSACTION statement.

Methods of Structuring the Update Library

There are three approaches to structuring the Update Library:

- Invoking the entire database using a single update library
- Invoking part of the database using a single update library
- Invoking multiple parts of the database using multiple update libraries

Whatever approach is used to implement the Update Library, the library must provide the external entry point ACCESSDATABASE and must be compiled as

```
<base name>/CODE/UPDATELIB
```

so that the Transaction Library can find it.

Synchronizing TPS and DMSII recovery is an important consideration in deciding which approach to use. Refer to the synchronization statements in this section for more information.

Information relating to the synchronization statements is included under “TPS BEGINTRANSACTION Statement,” “TPS ENDTRANSACTION Statement,” “MIDTRANSACTION Statement,” “BDMS OPEN Statement with TPS,” and “Transaction Processing Statements” in this section.

Example: Update Library Skeleton Program

An example of the correct structure for an Update Library is shown in a skeleton program on the following pages. The example uses multiple libraries to provide the code that actually processes the transaction records.

```
$ SHARING = PRIVATE
BEGIN  % Transaction Update Library

    LIBRARY DBSUBONE ( TITLE = "TRBASE/UPDATELIB/SUBONE ");

     PROCEDURE ACCESSSUBBASEONE(FUNCTIONFLAG,INQ,TRIN,TROUT,
                             SAVEINPUT, SAVERESPONSE );
                VALUE FUNCTIONFLAG, INQ;
                REAL FUNCTIONFLAG, INQ;
                TRANSACTION RECORD (TRB) TRIN, TROUT;
                PROCEDURE SAVEINPUTTR(); FORMAL;
                PROCEDURE SAVERESPONSETR(); FORMAL;
                LIBRARY DBSUBONE;

    LIBRARY DBSUBTWO ( TITLE = "TRBASE/UPDATELIB/SUBTWO ");
      PROCEDURE ACCESSSUBBASETWO(FUNCTIONFLAG,INQ,TRIN,TROUT,
                             SAVEINPUT,SAVERESPONSETR );
                VALUE FUNCTIONFLAG, INQ;
                REAL FUNCTIONFLAG, INQ;
                TRANSACTION RECORD (TRB) TRIN, TROUT;
                PROCEDURE SAVEINPUTTR(); FORMAL;
                PROCEDURE SAVERESPONSETR(); FORMAL;
                LIBRARY DBSUBONE;

    DEFINE UPDATEV = 1 #,
    FORCEABORTV = 2 #;

    % Global variables
  REAL LASTSUBBASE, OPENTYPE;
  ....

  PROCEDURE FORCEABORT;
  BEGIN
    CASE LASTSUBBASE OF
      BEGIN
        (2):
             ACCESSSUBBASEONE(FORCEABORTV,SAVEFUNCTIONFLAG,TRIN,TROUT,
                              SAVEINPUT,SAVERESPONSE);

        (3):
             ACCESSSUBBASETWO(FORCEABORTV,SAVEFUNCTIONFLAG,TRIN,TROUT,
                              SAVEINPUT,SAVERESPONSE);


             ....
    END OF CASE;
         ...
END;

  PROCEDURE UPDATE(TRIN,TROUT,SAVEINPUTTR,SAVERESPONSETR);
  TRANSACTION RECORD (TRB) TRIN, TROUT;
  PROCEDURE SAVEINPUTTR(); FORMAL;
  PROCEDURE SAVERESPONSETR(); FORMAL;
  BEGIN
    ....
    LASTSUBBASE := TRIN.TRSUBBASE;
    CASE TRIN.TRSUBBASE OF
      BEGIN
        (2):
             ACCESSSUBBASEONE(UPDATEV,SAVEFUNCTIONFLAG,TRIN,TROUT,
                              SAVEINPUT,SAVERESPONSE);
             % Invokes library DBSUBONE
        (3):
             ACCESSSUBBASETWO(UPDATEV,SAVEFUNCTIONFLAG,TRIN,TROUT,
                              SAVEINPUT,SAVERESPONSE);
             % Invokes library DBSUBTWO
    END OF CASES;
    ...
  END;

    PROCEDURE ACCESSDATABASE(FUNCTIONFLAG,TRIN,TROUT,SAVEINPUT,
                           SAVERESPONSE);
        VALUE FUNCTIONFLAG;
        REAL FUNCTIONFLAG;
        TRANSACTION RECORD (TRB) TRIN, TROUT;
        PROCEDURE SAVEINPUT(); FORMAL;
        PROCEDURE SAVERESPONSE(); FORMAL;
  % External entrypoint
  BEGIN

      CASE FUNCTIONFLAG OF
      BEGIN
```

```
           1: % Open update
              OPENTYPE := FUNCTIONFLAG;
           2: % Open inquiry
              OPENTYPE := FUNCTIONFLAG;
           3: % Update
              UPDATE(TRIN, TROUT, SAVEINPUT, SAVERESPONSE);
           4: % Force abort
              FORCEABORT;
           5: % Close database
              % Let BLOCKEXIT Do It;
         END;
     END ACCESSDATABASE;
*********************************************************************

       EXPORT
       ACCESSDATABASE;

       FREEZE(TEMPORARY);

   END OF LIBRARY.
```

Transaction Processing Statements

Generally, DMSII program interface statements are used for programming the Update Library in TPS. The following extensions and statements are required for the Update Library to synchronize TPS recovery with DMSII recovery.

- The MIDTRANSACTION statement
- Optional extensions to the BEGINTRANSACTION and ENDTRANSACTION statements
- The TRUPDATE option for the BDMS OPEN statement

These extensions are detailed in alphabetical order on the following pages. Examples of their use are in the sample programs at the end of this section. Consult the *DMSII TPS Programming Guide* for further information on using the statements. Refer to the *DMSII Application Program Interfaces Programming Guide* for information on exception handling.

Note that the TPS syntax of these statements is uniquely designed for TPS. DMSII applications that do not use TPS can continue to use the DMSII syntax that existed prior to the implementation of TPS. However, user-written code in the Update Library must use the syntax as it is defined here.

Additional information relating to the syntax of the TPS statements is included under "TPS BEGINTRANSACTION Statement," "TPS ENDTRANSACTION Statement," "MIDTRANSACTION Statement," "BDMS OPEN Statement with TPS," and "Sample User-Written Applications" in this section.

TPS BEGINTRANSACTION Statement

```
— BEGINTRANSACTION ┬────────────────────────────────────────┬─ ( ──────────→
                   │                                        │
                   └ <inputheadername> ┬───────────────────┬┘
                                       │                   │
                                       └ <message area> ───┘


→— <transaction record variable> — ) — <restart data set> ┬─────────────────────┬┤
                                                         │                     │
                                                         └ <exception handling>┘
```

Explanation

The TPS BEGINTRANSACTION statement places a program in transaction state. This statement can be used only with audited databases. Any attempt to modify an audited database when the program is not in transaction results in a fault.

The database must be opened with the TRUPDATE form of the BDMS OPEN statement.

If a BEGINTRANSACTION statement is attempted while the program is in transaction state, an exception is returned. The program is not placed in transaction state. If an ABORT exception is returned, all records that the program has locked are freed.

Deadlock can occur during execution of a BEGINTRANSACTION statement.

The transaction record variable construct is the formal input transaction record variable.

The restart data set name is detailed in the *DMSII TPS Programming Guide.*

Exception handling is detailed in the *DMSII Application Program Interfaces Programming Guide.*

Additional information relating to the TPS BEGINTRANSACTION statement is included under “Transaction Processing Statements,” “Declaring Transaction Record Variables," “BDMS OPEN Statement with TPS,” Related information is also available under “DMSII BEGINTRANSACTION Statement” in Section 4, “Using the Data Management System II (DMSII) Interface.”

Additional information regarding the transaction record variable construct is included under “Passing Transaction Record Variables as Parameters” in this section. Information on the inputheadername and message area constructs is included under “Declaring Input and Output Headers,” and “RECEIVE Statement” respectively in Section 3, “Using Communications Management System (COMS) Features.” Information on the exception handling construct is included under “Database Status Word” in Section 4, “Using the Data Management System II (DMSII) Interface.”

Example

In the following BEGINTRANSACTION statement, the transaction record variable is TRIN, the restart data set is RDS, and the exception variable is RSLT. Note that the colon preceding the exception variable is part of the exception handling syntax.

```
BEGINTRANSACTION (TRIN) RDS :RSLT;
```

TPS ENDTRANSACTION Statement

```
 ─ ENDTRANSACTION  ─ ( ─ <endtransaction parameters> ─ ) ─ <restart data set name>──→

 →─┬─────────┬─┬───────────────────────────┬──────────────────────────────────────────┤
   │         │ │                           │
   └─ SYNC  ─┘ └─ <exception handling> ─┘
```

<endtransaction parameters>

```
─ <transaction record variable ID> ─ , ─ <saveresponsetr procedure ID>──────────────────┤
```

Explanation

The TPS ENDTRANSACTION statement takes a program out of transaction state. This statement can be used only with audited databases. The database must be opened with the TRUPDATE form of the BDMS OPEN statement.

If an ENDTRANSACTION statement is attempted and the program is not in transaction state, an exception is returned. Records are freed in all cases of an exception and the transaction is not applied to the data base.

Refer to the *DMSII Application Program Interfaces Programming Guide* for information regarding audit and recovery.

Additional information relating to the ENDTRANSACTION statement is included under "Declaring Transaction Record Variables," "Transaction Processing Statements," and "BDMS OPEN Statement with TPS" in this section. Information is also available under "DMSII ENDTRANSACTION Statement" in Section 4, "Using the Data Management System II (DMSII) Interface."

The transaction record variable ID construct is the formal input transaction record variable. The saveresponsetr procedure ID identifies the SAVERESPONSETR formal procedure.

The restart data set name is detailed in the *DMSII TPS Programming Guide.*

The word "SYNC" forces a syncpoint.

Exception handling is detailed in the *DMSII Application Program Interfaces Programming Guide.*

Example

In the following ENDTRANSACTION statement, the transaction record variable is TRIN and the name of the saveresponsetr procedure variable is SAVERESPONSE. The restart data set is RDS. There is no forced syncpoint. The exception variable is RSLT. Note that the colon preceding the exception variable is part of the exception handling syntax.

```
ENDTRANSACTION (TRIN,SAVERESPONSE) RDS :RSLT;
```

MIDTRANSACTION Statement

```
– MIDTRANSACTION  – ( – <midtransaction parameters> – ) – <restart data set name>———→

→—┬————————————————————┬————————————————————————————————————————————————————————┤
  |                    |
  └─ <exception handling> ─┘
```

<midtransaction parameters>

```
– <transaction record variable ID> – , – <saveinputtr procedure ID> ———————┤
```

Explanation

The MIDTRANSACTION statement causes the compiler to generate calls on the given procedure immediately before the call on the DMS procedure in the Accessroutines.

The database must be opened with the TRUPDATE form of the BDMS OPEN statement.

The transaction record variable ID construct is the formal input transaction record variable. The saveinputtr procedure ID is the name of the SAVEINPUT formal procedure.

The restart data set name is detailed in the *DMSII TPS Programming Guide*.

Additional information relating to the MIDTRANSACTION statement is included under “Declaring Transaction Record Variables,” “Transaction Processing Statements,” and “BDMS OPEN Statement with TPS” in this section.

Exception handling is detailed in the *DMSII Application Program Interfaces Programming Guide*.

Example

In the following MIDTRANSACTION statement, the transaction record variable is TRIN and the name of the saveinputtr procedure variable is SAVEINPUT. The restart data set is RDS. The exception variable is RSLT. Note that the colon preceding the exception variable is part of the exception handling syntax.

```
MIDTRANSACTION (TRIN,SAVEINPUT) RDS :RSLT;
```

BDMS OPEN Statement with TPS

<BDMS open statement>

```
– OPEN ──┬──────────────┬─ <database identifier> ─┬──────────────────────────┬──┤
         ├─ INQUIRY ────┤                         └─ <exception handling> ──┘
         └─ TRUPDATE ───┘
```

Explanation

The BDMS OPEN statement opens a database for subsequent access and specifies the access mode.

An exception is returned if the database is already open. If an exception is returned, the state of the database remains unchanged.

An OPEN statement must be executed before the first access of the database; otherwise, the program terminates with a fault.

The word “INQUIRY” enforces read-only access to the database. This option is specified when no update operations are to be performed on the database. An exception is returned if the following BDMSALGOL statements are used when the database has been opened with the INQUIRY option:

ASSIGN	GENERATE
BEGINTRANSACTION	INSERT
DELETE	REMOVE
ENDTRANSACTION	STORE

The data management system does not open any audit files if the “OPEN INQUIRY” form has been used by all programs accessing the database.

The TRUPDATE option must be specified in order to use the MIDTRANSACTION statement or the transaction record variable form of the BEGINTRANSACTION or ENDTRANSACTION statements.

The database identifier specifies the database to be opened.

Additional information relating to the BDMS OPEN statement is included under “Transaction Processing Statements” and “Methods of Structuring the Update Library” in this section. Related information is also available under “BDMS OPEN Statement” in Section 4, “Using the Data Management System II (DMSII) Interface.”

Exception handling is detailed in the *DMSII Application Program Interfaces Programming Guide.*

Examples

In the following example, the word INQUIRY forces read-only access to the database DB. The exception variable is RSLT. Note that the colon preceding the exception variable is part of the exception handling syntax.

```
OPEN INQUIRY DB :RSLT;
```

In the following example, the word TRUPDATE enables write access to the database DB.

```
OPEN TRUPDATE DB;
```

Sample User-Written Applications

Three examples are shown in the following pages. The first example is a user-written skeleton program that demonstrates how the transaction base and Transaction Library entry points are declared. The second example shows a complete transaction base banking application. The third example is a detanking procedure.

The banking application, Example 2, includes the needed DASDL description, TFL description, and Update Library. The descriptions are written in their respective language (DADSL or TFL). The application program and Update Library are written in ALGOL.

Example 3, the detanking procedure, builds on the banking application shown in Example 2.

Related information about these user-written programs can be found in the *DMSII TPS Programming Guide.*

Example 1: Declaring a Transaction Base and Library

Any user-written program that invokes the TPS Transaction Library should not be a library itself. Each program that submits a transaction record for processing must have its own private library for recovery to be successful. If an application program is written as a shared library, then the Transaction Library might not work. The first program that invokes an entry point becomes the only TPS user recognized by the Transaction Library.

```
BEGIN   % Sample batch program using transactions.

      TRANSACTION BASE TRB = BANKTR;
      LIBRARY L(TITLE="BANKTR/CODE/HOSTLIB.");

% Declare all entrypoints to be used.

  INTEGER PROCEDURE CREATETRUSER(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE PURGETRUSER(IDNUM);
      INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE LOGONTRUSER(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE LOGOFFTRUSER(IDNUM);
      INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE RETURNRESTARTINFO(IDNUM, TROUT);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TROUT;
      LIBRARY L;
  INTEGER PROCEDURE RETURNLASTRESPONSE(IDNUM, TROUT);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TROUT;
      LIBRARY L;
  INTEGER PROCEDURE TANKTRNORESTART(IDNUM, TRIN);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TRIN;
      LIBRARY L;
  INTEGER PROCEDURE PROCESSTRNORESTART(IDNUM, TRIN, TROUT);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TRIN, TROUT;
      LIBRARY L;
  INTEGER PROCEDURE OPENTRBASE(USEROPTION, TIMEOUT);
      INTEGER USEROPTION, TIMEOUT;
      LIBRARY L;
  INTEGER PROCEDURE CLOSETRBASE;
      LIBRARY L;
  INTEGER PROCEDURE SEEKTRANSACTION(FILENUM, BLOCKNUM, OFFSET);
      INTEGER FILENUM, BLOCKNUM, OFFSET;
      LIBRARY L;
  INTEGER PROCEDURE READTRANSACTION (TRREC);
      TRANSACTION RECORD (TRB) TRREC;
      LIBRARY L;
  INTEGER PROCEDURE SWITCHTRFILE;
      LIBRARY L;
  INTEGER PROCEDURE HANDLESTATISTICS(STATOPTION);
      VALUE STATOPTION;
      INTEGER STATOPTION;
      LIBRARY L;

% Declare transaction record variables to be used.

       TRANSACTION RECORD (TRB)
           TRIN,
           TROUT,
           LASTINPUT,
           LASTRESPONSE;

       STRING JOURNALNAME;
       .
       .
       .
  %   Start of program.
  %   Set LIBPARAMETER in declaration or before first call on entrypoint.

        L.LIBPARAMETER := JOURNALNAME;

    %   Body of program.
        .
        .
        .
  END.
```

Example 2: Banking Application

The following example is a typical DMSII application using TPS. In the example, bank accounts are created and deleted, deposits and withdrawals are made, and account balances are determined.

For the application to operate properly, several pieces of user-supplied software are needed:

- A Data Structure and Definition Language (DASDL) description.

  In DMSII, DASDL is used to describe a database logically and physically.
- A Transaction Formatting Language (TFL) description.

  In the TPS, the TFL is used to describe the transaction base logically and physically.
- A user-written application program.

  The user-written ALGOL program shows how TPS can be used for a number of simple banking transactions.
- An Update Library.

  The Update Library is capable of maintaining database consistency and ensuring reproducibility.

The ALGOL application program and the TPS need both the DASDL and TPS descriptions to ensure the integrity of data stored in the database and transaction base.

Examples of the user-supplied software are included under “DASDL Description of the database,” “TFL Description of the Transaction Base,” “ALGOL Banking Application Program,” and “Update Library” in this section.

DASDL Description of the Database

```
OPTIONS(AUDIT);
 PARAMETERS(SYNCPOINT = 10 TRANSACTIONS);

   ACCOUNT DATA SET           % Specify a data set to hold the account
   (                          % numbers and info associated with them.
     ACCOUNT-NUM NUMBER(6);
     NAME ALPHA(20);
     BALANCE REAL(S10,2);

       DEPOSIT UNORDERED DATA SET  % Used to keep history of the deposits
       (                           % and withdrawals made.
          TRANDATE REAL;
          OLD-BALANCE REAL(S10,2);
          AMOUNT REAL(S10,2);      % Negative for withdrawal.
          NEW-BALANCE REAL(S10,2);
      );
    );
   ACCOUNT-SET SET OF ACCOUNT
     KEY ACCOUNT-NUM;

  RDS RESTART DATA SET  % Remember, a restart data set must be specified.
 (
     X ALPHA(10);
 );
```

TFL Description of the Transaction Base

```
BANKTR TRANSACTION BASE;    % First declare the name of the transaction
                            % base we are about to describe.
  PARAMETERS
  (
      STATISTICS,
      DATABASE = BANKDB ON DISK,
      RESTARTDATASET = RDS,
      HOSTSYSTEM = SYS456
  );
    DEFAULTS       % Specify defaults for items of transaction formats
                   % and for journal control and data files.
  (
      ALPHA (INITIALVALUE = BLANKS),
      BOOLEAN (INITIALVALUE = FALSE),
      NUMBER (INITIALVALUE = 0),
      REAL (INITIALVALUE = 0),
      CONTROL FILE
      (
          AREAS = 100,
          AREASIZE = 100 BLOCKS,
          BLOCKSIZE = 20 SEGMENTS,
          FAMILY = DISK,
          CHECKSUM = TRUE
      ),
      DATA FILE
      (
          AREAS = 100,
          AREASIZE = 100 BLOCKS,
          BLOCKSIZE = 30 SEGMENTS,
          FAMILY = DISK,
          CHECKSUM = TRUE
      )
  );

    CREATEACCT TRANSACTION FORMAT    % The following formats are
  (                                  % used in the application
      ACCTNUM NUMBER(6);             % program and the Update
      NAME ALPHA(20);                % Library.
  );

    PURGEACCT TRANSACTION FORMAT
  (
      ACCTNUM NUMBER(6);
  );
  DEPOSIT TRANSACTION FORMAT
  (
      ACCTNUM NUMBER(6);
      TRANDATE REAL;
      AMOUNT REAL(10,2);
```

```
);

  WITHDRAWAL TRANSACTION FORMAT
(
    ACCTNUM NUMBER(6);
    AMOUNT REAL(10,2);
    TRANDATE REAL;
);

    STATUS TRANSACTION FORMAT
(
    ACCTNUM NUMBER(6);
    BALANCE REAL(S10,2);
    G GROUP
    (  A ALPHA(20);
       B REAL; );

);

 RESTARTDETANKER TRANSACTION FORMAT   % This format illustrates possible
  (                                   % information to be kept in a
      TANKFILENUM FIELD(14);          % restart transaction record.
      TANKBLOCKNUM FIELD(32);
      TANKOFFSET FIELD(16);

);

  MANAGER TRANSACTION SUBBASE  % Example subbase that a manager might
(                              % use. Note that a GUARDFILE is attached
    CREATEACCT,                % to the subbase for security.
    PURGEACCT,
    DEPOSIT,
    WITHDRAWAL,
    STATUS,

),
    GUARDFILE = BANKTR/MANAGER/GUARDFILE;

  TELLER TRANSACTION SUBBASE    % Example subbase a teller might use.
(
    DEPOSIT,
    WITHDRAWAL,
    STATUS

);
TRHISTORY TRANSACTION JOURNAL  % Example of specifying explicit values
CONTROL FILE                   % for the attributes of the TRHISTORY
(                              % journal.
    AREAS = 100,
    AREASIZE = 100 BLOCKS,
    BLOCKSIZE = 20 SEGMENTS,
    FAMILY = DISK,
        CHECKSUM = TRUE

    ),
    DATA FILE
    (
        AREAS = 100,
        AREASIZE = 2 BLOCKS,
        BLOCKSIZE = 3 SEGMENTS,
        FAMILY = DISK,
        CHECKSUM = TRUE
    );

      TANK1 TRANSACTION JOURNAL      % Example of TANK journal attribute
      CONTROL FILE                   % specification.
    (
        USERCODE = SAMPLEUSER,
        FAMILY = PACK
    ),
    DATA FILE
    (
        USERCODE = SAMPLEUSER,
        DUPLICATED ON DISK
    );
```

ALGOL Banking Application Program

```
BEGIN   % Sample batch program using transactions.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
%  The library routines, declared below, provide the proper function  %
%  for either environment.                                             %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    ARRAY LIBPARAM[0:9];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
%  Declare the transaction base to be used.                            %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   TRANSACTION BASE TRB = BANKTR;     % Example of equating an internal
                                      % name to the transaction base.
LIBRARY L(TITLE="BANKTR/CODE/HOSTLIB.");

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
%  Declare all the library entry points to be used.                    %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

INTEGER PROCEDURE CREATETRUSER(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE PURGETRUSER(IDNUM);
      INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE LOGONTRUSER(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE LOGOFFTRUSER(IDNUM);
      INTEGER IDNUM;
      LIBRARY L;
  INTEGER PROCEDURE RETURNLASTADDRESS(FILENUM, BLOCKNUM, OFFSET, IDNUM);
      INTEGER IDNUM;
      REAL FILENUM, BLOCKNUM, OFFSET;
      LIBRARY L;
  INTEGER PROCEDURE RETURNRESTARTINFO(IDNUM, TROUT);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TROUT;
      LIBRARY L;
```

```
INTEGER PROCEDURE RETURNLASTRESPONSE(IDNUM, TROUT);
    INTEGER IDNUM;
    TRANSACTION RECORD (TRB) TROUT;
    LIBRARY L;
INTEGER PROCEDURE TANKTRNORESTART(IDNUM, TRIN);
    INTEGER IDNUM;
    TRANSACTION RECORD (TRB) TRIN;
    LIBRARY L;
INTEGER PROCEDURE PROCESSTRANSACTION(IDNUM, TRIN, TROUT, RESTARTTRREC);
    INTEGER IDNUM;
    TRANSACTION RECORD (TRB) TRIN, TROUT, RESTARTTRREC;
    LIBRARY L;
INTEGER PROCEDURE PROCESSTRNORESTART(IDNUM, TRIN, TROUT);
    INTEGER IDNUM;
    TRANSACTION RECORD (TRB) TRIN, TROUT;
    LIBRARY L;
INTEGER PROCEDURE OPENTRBASE(USEROPTION, TIMEOUT);
    INTEGER USEROPTION, TIMEOUT;
    LIBRARY L;
INTEGER PROCEDURE CLOSETRBASE;
    LIBRARY L;
INTEGER PROCEDURE SEEKTRANSACTION(FILENUM, BLOCKNUM, OFFSET);
    INTEGER FILENUM, BLOCKNUM, OFFSET;
    LIBRARY L;
INTEGER PROCEDURE READTRANSACTION (TRREC);
    TRANSACTION RECORD (TRB) TRREC;
    LIBRARY L;
INTEGER PROCEDURE SWITCHTRFILE;
    LIBRARY L;
INTEGER PROCEDURE HANDLESTATISTICS(STATOPTION);
    VALUE STATOPTION; INTEGER STATOPTION;
    LIBRARY L;

      FILE LINE(KIND=PRINTER);
    FILE RMOTE(KIND=REMOTE, MYUSE = IO);
    TRANSACTION RECORD (TRB)
         TRIN,
         TROUT,
         LASTINPUT,
         RESTARTTRREC,
         LASTRESPONSE;
    REAL IDNUM, N, OPT;
    INTEGER ACCT,TIMEOUT,STATISTICSOPTION;
    INTEGER ACCT,TIMEOUT;
    INTEGER RSLT;
    ARRAY SP[0:14];
    BOOLEAN ERROR;
    LABEL EXIT;
    STRING ID, FNAME, JOURNALNAME;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Body of the program.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DEFINE TANKING = 3#;

    DEFINE ERR =
  BEGIN
       WRITE(RMOTE, <"RSLT = ", I3, " @ ", I8>, RSLT, LINENUMBER);
       ERROR := TRUE;
  END#;

    DEFINE GETSTRING(S, X) =
  BEGIN
      REPLACE SP BY " " FOR 15 WORDS;
      REPLACE SP BY "ENTER STRING FOR ", S;
      WRITE(RMOTE, 15, SP);
      READ(RMOTE, 15, SP);
      SCAN SP FOR N:80 WHILE IN ALPHA;
      X := STRING(SP, 80-N);
  END#;
    DEFINE GETINTEGER(S, I) =
  BEGIN
       REPLACE SP BY " " FOR 15 WORDS;
       REPLACE SP BY "ENTER INTEGER FOR ", S;
       WRITE(RMOTE, 15, SP);
      READ(RMOTE, /, I);
  END#;

    DEFINE GETREAL(S, R) =
  BEGIN
      REPLACE SP BY " " FOR 15 WORDS;
      REPLACE SP BY "ENTER VALUE FOR ", S;
      WRITE(RMOTE, 15, SP);
      READ(RMOTE, /, R);
  END#;

    DEFINE GETACCT = GETINTEGER("ACCOUNT NUMBER", ACCT)#;

    PROCEDURE PROCESSTR;
  BEGIN
     IF OPT = TANKING THEN
     BEGIN
        IF RSLT := TANKTRNORESTART(IDNUM, TRIN) > 0 THEN ERR;
     END
     ELSE
        IF RSLT := PROCESSTRANSACTION(IDNUM, TRIN, TROUT, RESTARTTRREC)
              > 0 THEN ERR;
  END PROCESSTR;
```

```
PROCEDURE GETLASTP;
BEGIN
   IF OPT = TANKING THEN
   BEGIN
      IF RSLT := RETURNRESTARTINFO(IDNUM, LASTINPUT) > 0 THEN
         ERR
      ELSE
         WRITE(RMOTE,
              <"LAST RESTART  (FILE, BLOCK, OFFSET, FORMAT): ", 4I5>,
              LASTINPUT.TRFILENUM,
              LASTINPUT.TRBLOCKNUM, LASTINPUT.TROFFSET,
              LASTINPUT.TRFORMAT );
   END ELSE
   BEGIN
      IF RSLT := RETURNRESTARTINFO(IDNUM, LASTINPUT) > 0 THEN
         ERR
      ELSE
      BEGIN
         WRITE(RMOTE,
             <"LAST RESTART  (FILE, BLOCK, OFFSET, FORMAT): ", 4I5>,
             LASTINPUT.TRFILENUM,
             LASTINPUT.TRBLOCKNUM, LASTINPUT.TROFFSET,
             LASTINPUT.TRFORMAT );
         IF RSLT := RETURNLASTRESPONSE(IDNUM, LASTRESPONSE) > 0 THEN
            ERR
         ELSE
            WRITE(RMOTE,
                <"LAST RESPONSE (FILE, BLOCK, OFFSET, FORMAT): ", 4I5>,
                 LASTRESPONSE.TRFILENUM,
                 LASTRESPONSE.TRBLOCKNUM, LASTRESPONSE.TROFFSET,
                 LASTRESPONSE.TRFORMAT );
      END;
   END;
END GETLASTP;

    PROCEDURE DISPLAYSTATUS;
BEGIN
    IF TROUT.TRFORMAT NEQ TRFORMAT(STATUS) THEN
       ERR
    ELSE
       WRITE(RMOTE, <"ACCOUNT NUMBER ", I5,
            ": CURRENT BALANCE IS ", F10.2>,
            TROUT.STATUS.ACCTNUM,
            TROUT.STATUS.BALANCE);
END DISPLAYSTATUS;
PROCEDURE CREATEP;   % Create a new account number.
BEGIN
    STRING NAME;
    WRITE(RMOTE, <"FUNCTION IS CREATE">);
    GETACCT;
    GETSTRING("CUSTOMER NAME", NAME);
    CREATE TRIN.CREATEACCT;
```

```
        TRIN.CREATEACCT.ACCTNUM := ACCT;
        TRIN.CREATEACCT.NAME := NAME;
        PROCESSTR;
    END CREATEP;

      PROCEDURE PURGEP;    % Eliminate an account number.
    BEGIN
        WRITE(RMOTE, <"FUNCTION IS PURGE">);
        GETACCT;
        CREATE TRIN.PURGEACCT;
        TRIN.PURGEACCT.ACCTNUM := ACCT;
        PROCESSTR;
    END PURGEP;

        PROCEDURE STATUSP;   % Display the status of an account.
    BEGIN
        WRITE(RMOTE, <"FUNCTION IS STATUS">);
        GETACCT;
        CREATE TRIN.STATUS;
        TRIN.STATUS.ACCTNUM := ACCT;
        PROCESSTR;
        IF (OPT NEQ TANKING AND NOT ERROR) THEN DISPLAYSTATUS;
    END STATUSP;

        PROCEDURE DEPOSITP;    % Deposit some amount in an account.
    BEGIN
        REAL AMT;
        WRITE(RMOTE, <"FUNCTION IS DEPOSIT">);
        GETACCT;
        GETREAL("AMOUNT OF DEPOSIT", AMT);
        CREATE TRIN.DEPOSIT;
        TRIN.DEPOSIT.ACCTNUM := ACCT;
        TRIN.DEPOSIT.TRANDATE := TIME(6);
        TRIN.DEPOSIT.AMOUNT := AMT;
        PROCESSTR;
        IF (OPT NEQ TANKING AND NOT ERROR) THEN DISPLAYSTATUS;
    END DEPOSITP;
    PROCEDURE WITHDRAWALP;    % Withdraw some amount from an account.
    BEGIN
        REAL AMT;
        WRITE(RMOTE, <"FUNCTION IS WITHDRAWAL">);
        GETACCT;
        GETREAL("AMOUNT OF WITHDRAWAL", AMT);
        CREATE TRIN.WITHDRAWAL;
        TRIN.WITHDRAWAL.ACCTNUM := ACCT;
        TRIN.WITHDRAWAL.TRANDATE := TIME(6);
        TRIN.WITHDRAWAL.AMOUNT := AMT;
        PROCESSTR;
        IF (OPT NEQ TANKING AND NOT ERROR) THEN DISPLAYSTATUS;
    END WITHDRAWALP;
```

```
  PROCEDURE NEWUSERP;
BEGIN
    WRITE(RMOTE, <"FUNCTION IS NEWUSER">);
    GETSTRING("USER ID", ID);
    WRITE(RMOTE, <"USER: ", A15>, ID);
    IF RSLT := LOGONTRUSER(ID, IDNUM) > 0 THEN
       ERR
    ELSE
       WRITE(RMOTE, <"USER #: ", I3>, IDNUM);
END NEWUSERP;

  PROCEDURE REOPENP;
BEGIN
    WRITE(RMOTE, <"FUNCTION IS REOPEN">);
    IF RSLT := CLOSETRBASE > 0 THEN
       ERR
    ELSE
    BEGIN
       WRITE(RMOTE, <"WHAT DO YOU WANT TO DO?">);
       GETINTEGER("CHOICE (1=UPDATE, 2=INQUIRY, 3=TANK, 4=READ,"
                  "5=EXCLUSIVEUPDATE)", OPT);
       IF RSLT := OPENTRBASE(OPT, 0) > 0 THEN ERR;
    END;
END REOPENP;

  PROCEDURE SEEKP;
BEGIN
    REAL FILENUM, BLOCKNUM, OFFSET;
    WRITE(RMOTE, <"FUNCTION IS SEEK">);
    WRITE(RMOTE, <"ENTER FILENUM, BLOCKNUM, OFFSET">);
    READ(RMOTE, /, FILENUM, BLOCKNUM, OFFSET);
    IF RSLT := SEEKTRANSACTION(FILENUM, BLOCKNUM, OFFSET) > 0 THEN ERR;
END SEEKP;
PROCEDURE READP;
BEGIN
    WRITE(RMOTE, <"FUNCTION IS READ">);
    IF RSLT := READTRANSACTION(TRIN) > 0 THEN
       ERR
    ELSE
    BEGIN
       WRITE(RMOTE, <"FILE, BLOCK, OFFSET:", 3I5>,
         TRIN.TRFILENUM,
         TRIN.TRBLOCKNUM,
         TRIN.TROFFSET);
       WRITE(RMOTE, <"FORMAT, SUBFORMAT:", 2I5>,
         TRIN.TRFORMAT,
         TRIN.TRSUBFORMAT);
    END;
END READP;

  PROCEDURE CREATEUSERP;
BEGIN
```

```
            WRITE(RMOTE, <"FUNCTION IS CREATEUSER">);
            GETSTRING("USER ID", ID);
            WRITE(RMOTE, <"USER: ", A15>, ID);
            IF RSLT := CREATETRUSER(ID, IDNUM) > 0 THEN
               ERR
            ELSE
               IF RSLT := LOGONTRUSER(ID, IDNUM) > 0 THEN
                  ERR
               ELSE
                  WRITE(RMOTE, <"USER #: ", I3>, IDNUM);
        END
       CREATEUSERP;

          PROCEDURE PURGEUSERP;
        BEGIN
            WRITE(RMOTE, <"FUNCTION IS PURGEUSER">);
            IF RSLT := PURGETRUSER(IDNUM) > 0 THEN ERR;
        END PURGEUSERP
      PROCEDURE QUITP;
        BEGIN
            WRITE(RMOTE, <"FUNCTION IS QUIT">);
            CLOSETRBASE;
            GO EXIT;
        END QUITP;

          PROCEDURE SWITCHP;
        BEGIN
            WRITE(RMOTE, <"FUNCTION IS SWITCH">);
            IF RSLT := SWITCHTRFILE > 0 THEN ERR;
        END SWITCHP;
        PROCEDURE STATISTICSP;
        BEGIN
            WRITE(RMOTE, <"FUNCTION IS STATISTICS">);
            WRITE(RMOTE, <"WHAT DO YOU WANT TO DO?");
            GETINTEGER("CHOICE (1 = PRINT & RESET, 2 = PRINT"
                       "ONLY, 3 = RESET)", STATISTICSOPTION);
            IF RSLT := HANDLESTATISTICS(STATISTICSOPTION) > 0;
            THEN ERR;
        END STATISTICSP;

          PROCEDURE HELPP;
        BEGIN
            WRITE(RMOTE, <"FUNCTIONS ARE:", /,
                 "CREATE", /,
                 "PURGE", /,
                 "DEPOSIT", /,
                 "WITHDRAWAL", /,
                 "QUIT", /,
                 "STATUS", /,
                 "NEWUSER", /,
                 "REOPEN", /,
                 "SEEK", /,
          "READ", /,
          "GETLAST", /,
          "CREATEUSER", /,
          "PURGEUSER", /,
          "SWITCH", /,
          "STATISTICS", /,
          "HELP" > );
  END HELPP;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                     %
% Set LIBPARAMETER before first call on a library entry point. The    %
% LIBPARAMETER can be set in the library declaration rather than      %
% here.                                                               %
%                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      GETSTRING("JOURNAL NAME", JOURNALNAME);
      L.LIBPARAMETER := JOURNALNAME;

      WRITE(RMOTE, <"WHAT DO YOU WANT TO DO?">);
      GETINTEGER("CHOICE (1=UPDATE, 2=INQUIRY, 3=TANK, 4=READ,"
                 " 5=EXCLUSIVEUPDATE)", OPT);
      WRITE(RMOTE,<"WHAT VALUE FOR TIMEOUT SHALL WE USE?">);
      READ(RMOTE,/,TIMEOUT);
      IF RSLT := OPENTRBASE(OPT, TIMEOUT) > 0 THEN ERR;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %
% A restart transaction record is created.  It will be written to    %
% the TRHISTORY file along with an input transaction.  Here, we have %
% not assigned values to the items or this record.  Normally, values %
% are assigned but, for simplicity, the code was left out of this    %
% example.                                                           %
%                                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      IF NOT ERROR THEN
      BEGIN
         CREATE RESTARTTRREC.RESTARTDETANKER;
         GETSTRING("USER ID", ID);
         WRITE(RMOTE, <"USER: ", A15>, ID);
         IF RSLT := LOGONTRUSER(ID, IDNUM) > 0 THEN ERR;
         IF NOT ERROR THEN
            WRITE(RMOTE, <"USER #: ", I3>, IDNUM);
      END;
      ERROR := FALSE;
      WHILE TRUE DO
      BEGIN
         GETSTRING("FUNCTION NAME (OR HELP)", FNAME);
         IF SP = "CREATEUSER" THEN CREATEUSERP ELSE
         IF SP = "PURGEUSER" THEN PURGEUSERP ELSE
```

```
     IF SP = "CREATE" THEN CREATEP ELSE
     IF SP = "PURGE" THEN PURGEP ELSE
     IF SP = "DEPOSIT" THEN DEPOSITP ELSE
     IF SP = "WITHDRAWAL" THEN WITHDRAWALP ELSE
     IF SP = "QUIT" THEN QUITP ELSE
     IF SP = "STATUS" THEN STATUSP ELSE
     IF SP = "NEWUSER" THEN NEWUSERP ELSE
     IF SP = "REOPEN" THEN REOPENP ELSE
     IF SP = "SEEK" THEN SEEKP ELSE
     IF SP = "R" THEN READP ELSE
     IF SP = "HELP" THEN HELPP ELSE
     IF SP = "GETLAST" THEN GETLASTP ELSE
     IF SP = "SW" THEN SWITCHP ELSE
     IF SP = "STAT" THEN STATISTICSP ELSE
     WRITE(RMOTE, <"DID NOT RECOGNIZE FUNCTION NAME">);
     ERROR := FALSE;
  END;

EXIT:
END OF THE APPLICATION PROGRAM.
```

Update Library

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SHARING must be PRIVATE in order to ensure that each application      %
% program will get its own copy of the Update Library                   %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    $SET SHARING=PRIVATE
  BEGIN  % User's Transaction Update Library

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% This library is written by the user of the transaction system.       %
% It consists % of a single procedure, called "ACCESSDATABASE", which  %
% is designed to perform four basic functions: OPENDATABASE (for       %
% update or inquiry), UPDATE, FORCEABORT, and CLOSEDATABASE. The       %
% function to be performed is identified by the first parameter to     %
% the procedure.                                                       %
%                                                                      %
% OPENDATABASE for update or inquiry is required to open the data      %
% base.                                                                %
%                                                                      %
% UPDATE is called by the transaction system once for each input       %
% transaction to be processed. It must observe a few simple rules,     %
% such as when to lock records and when to call the formal             %
% procedures.  It is expected to examine each input transaction        %
% record, perform the appropriate actions, create a response           %
% transaction, and exit.                                               %
%                                                                      %
% FORCEABORT is required so that the transaction system can cause      %
% an abort, if necessary.                                              %
%                                                                      %
% CLOSEDATABASE must close the database.                               %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
%             Library global declarations.                             %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 DATABASE DB = BANKDB;             % Invoke the database and transaction
                                   % base to be used.
 TRANSACTION BASE TRB = BANKTR;
 EBCDIC ARRAY SPO[0:79];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Procedure update.
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    PROCEDURE UPDATE(TRIN, TROUT, SAVEINPUT, SAVERESPONSE);
      TRANSACTION RECORD (TRB) TRIN, TROUT;
      PROCEDURE SAVERESPONSE(); FORMAL;
      PROCEDURE SAVEINPUT(); FORMAL;
  BEGIN

        LABEL EXIT;
      BOOLEAN RSLT;
      CASE TRIN.TRFORMAT OF
      BEGIN

        (TRFORMAT(CREATEACCT)):      % Routine for creating a new account.
      BEGIN
           STRING SNAME;
           EBCDIC ARRAY NAME[0:29];
           SNAME := TRIN.CREATEACCT.NAME;
           REPLACE NAME[0] BY SNAME;
           CREATE ACCOUNT:RSLT;
           IF RSLT THEN GO EXIT;
           PUT ACCOUNT
           (
                ACCOUNT-NUM := TRIN.CREATEACCT.ACCTNUM,
                NAME := NAME[0];
           );
           BEGINTRANSACTION (TRIN) RDS :RSLT;
           IF RSLT THEN GO EXIT;
           MIDTRANSACTION (TRIN, SAVEINPUT) RDS: RSLT;
           IF RSLT THEN GO EXIT;
           STORE ACCOUNT:RSLT;
           IF RSLT THEN GO EXIT;
           TROUT := TRIN;  % Return same TR as good TR-RESPONSE.
           ENDTRANSACTION (TRIN, SAVERESPONSE) RDS :RSLT;
           IF RSLT THEN GO EXIT;
      END CREATEACCT FORMAT;
      (TRFORMAT(PURGEACCT)):      % Routine for purging an existing
                                  % account.

      BEGIN
           REAL ACCT;
           ACCT := TRIN.PURGEACCT.ACCTNUM;
           LOCK ACCOUNT-SET AT ACCOUNT-NUM = ACCT :RSLT;
           IF RSLT THEN GO EXIT;
           BEGINTRANSACTION (TRIN) RDS :RSLT;
           IF RSLT THEN GO EXIT;
           MIDTRANSACTION (TRIN, SAVEINPUT) RDS: RSLT;
           IF RSLT THEN GO EXIT;
           DELETE ACCOUNT:RSLT;
           IF RSLT THEN GO EXIT;
           TROUT := TRIN;  % Signal OK
```

```
          ENDTRANSACTION (TRIN, SAVERESPONSE) RDS :RSLT;
          IF RSLT THEN GO EXIT;
     END PURGEACCT FORMAT;

       (TRFORMAT(STATUS)):      % Example of an inquiry routine.  It
     BEGIN                      % returns the balance of a particular
                                % account.

          REAL ACCT, BAL;
          ACCT := TRIN.STATUS.ACCTNUM;
          FIND ACCOUNT-SET AT ACCOUNT-NUM = ACCT :RSLT;
          IF RSLT THEN GO EXIT;
          GET ACCOUNT
          (
               BAL := BALANCE
          );
          TROUT := TRIN;  % Signal OK
          TROUT.STATUS.BALANCE := BAL;
     END STATUS FORMAT;
     (TRFORMAT(DEPOSIT)):     % Routine to perform a deposit into an
     BEGIN                    % account.
          REAL OLDBAL, NEWBAL;
          REAL ACCT;
          ACCT := TRIN.DEPOSIT.ACCTNUM;
          LOCK ACCOUNT-SET AT ACCOUNT-NUM = ACCT :RSLT;
          IF RSLT THEN GO EXIT;
          GET ACCOUNT
          (
               OLDBAL := BALANCE
          );
          NEWBAL := OLDBAL + TRIN.DEPOSIT.AMOUNT;
          CREATE DEPOSIT:RSLT;
          IF RSLT THEN GO EXIT;
          PUT DEPOSIT
          (
               TRANDATE := TRIN.DEPOSIT.TRANDATE,
               AMOUNT := TRIN.DEPOSIT.AMOUNT,
               OLD-BALANCE := OLDBAL,
               NEW-BALANCE := NEWBAL
          );
          PUT ACCOUNT
          (
               BALANCE := NEWBAL
          );
          BEGINTRANSACTION (TRIN) RDS :RSLT;
          IF RSLT THEN GO EXIT;
          MIDTRANSACTION (TRIN, SAVEINPUT) RDS: RSLT;
          IF RSLT THEN GO EXIT;
          STORE ACCOUNT:RSLT;
          IF RSLT THEN GO EXIT;
          STORE DEPOSIT:RSLT;
          IF RSLT THEN GO EXIT;
```

```
                CREATE TROUT.STATUS;
                TROUT.STATUS.BALANCE := NEWBAL;
                TROUT.STATUS.ACCTNUM := TRIN.DEPOSIT.ACCTNUM;
                ENDTRANSACTION (TRIN, SAVERESPONSE) RDS :RSLT;
                IF RSLT THEN GO EXIT;
           END DEPOSIT FORMAT;
           (TRFORMAT(WITHDRAWAL)):     % Routine to withdraw money from an
                                       % account.
           BEGIN  % Uses DEPOSIT data set, not WITHDRAWAL
                REAL OLDBAL, NEWBAL;
                REAL ACCT;
                ACCT := TRIN.WITHDRAWAL.ACCTNUM;
                LOCK ACCOUNT-SET AT ACCOUNT-NUM = ACCT :RSLT;
                IF RSLT THEN GO EXIT;
                GET ACCOUNT
                (
                     OLDBAL := BALANCE
                );
                NEWBAL := OLDBAL - TRIN.WITHDRAWAL.AMOUNT;
                CREATE DEPOSIT:RSLT;
                IF RSLT THEN GO EXIT;
                PUT DEPOSIT
                (
                     TRANDATE := TRIN.WITHDRAWAL.TRANDATE,
                     AMOUNT := - TRIN.WITHDRAWAL.AMOUNT,
                     OLD-BALANCE := OLDBAL,
                     NEW-BALANCE := NEWBAL
                );
                PUT ACCOUNT
                (
                     BALANCE := NEWBAL
                );
                BEGINTRANSACTION (TRIN) RDS :RSLT;
                IF RSLT THEN GO EXIT;
                MIDTRANSACTION (TRIN, SAVEINPUT) RDS: RSLT;
                IF RSLT THEN GO EXIT;
                STORE ACCOUNT:RSLT;
                IF RSLT THEN GO EXIT;
                STORE DEPOSIT:RSLT;
                IF RSLT THEN GO EXIT;
                CREATE TROUT.STATUS;
                TROUT.STATUS.BALANCE := NEWBAL;
                TROUT.STATUS.ACCTNUM := TRIN.DEPOSIT.ACCTNUM;
                ENDTRANSACTION (TRIN, SAVERESPONSE) RDS :RSLT;
                IF RSLT THEN GO EXIT;
           END WITHDRAWAL FORMAT;
           ELSE:    % Flag an error
                DISPLAY("NO UPDATE ROUTINE FOR THE FORMAT PASSED IN");

             END CASES;
       EXIT:
           IF REAL(RSLT) ISNT 0 THEN
```

```
      BEGIN
           REPLACE SPO BY 0 FOR 10 WORDS;
           WRITE(SPO[*], <"UPDATE RSLT:", H13>, RSLT);
           DISPLAY(SPO);
      END;
  END UPDATE;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                     %
% Procedure ACCESSDATABASE.                                           %
%                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

PROCEDURE ACCESSDATABASE(FUNCTIONFLAG, TRIN, TROUT,
           SAVEINPUT, SAVERESPONSE);
      VALUE FUNCTIONFLAG;
      INTEGER FUNCTIONFLAG;
      TRANSACTION RECORD (TRB) TRIN, TROUT;
      PROCEDURE SAVERESPONSE(); FORMAL;
      PROCEDURE SAVEINPUT(); FORMAL;
  BEGIN
      CASE FUNCTIONFLAG OF
      BEGIN
           1:                   % Open update
                OPEN TRUPDATE DB;
           2:                   % Open inquiry
                OPEN INQUIRY DB;
           3:                   % Update
                UPDATE(TRIN, TROUT, SAVEINPUT, SAVERESPONSE);
           4:                   % FORCEABORT is called by the
                CLOSE DB;       % Transaction Library when the last call
                                % resulted in exiting this library while
                                % still in transaction state.
           5:                   % Close.
                CLOSE DB;
      END CASES;
  END ACCESSDATABASE;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                     %
% Initialize library.                                                 %
%                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   EXPORT
           ACCESSDATABASE;
        FREEZE(TEMPORARY);
    END UPDATE LIBRARY.
```

Example 3: Detanking Procedure

The ALGOL procedure on the following pages illustrates “detanking.” A detanking procedure reads transactions from a tank journal and processes them against the database.

The input parameter is the name of the Tank journal. This procedure opens both the Tank journal and the TRHISTORY journal, and then reads transactions from the Tank journal and processes them against the data base.

The transaction base invoked by this procedure is defined in “Example 2: Banking Application.” The procedure also uses the previously defined DASDL description and Update Library. Refer to “DASDL Description of the database,” “TFL Description of the Transaction Base,” and “Update Library” on the preceding pages for details.

```
PROCEDURE DETANKER(TANKNAME);
 ARRAY TANKNAME[*];
 BEGIN

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% This program can run on either the host system or a remote system.  %
% The library routines declared below provide the proper function      %
% for either environment.                                              %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   STRING TANKLIBPARAM;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% Declare the transaction base to be used.                             %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   TRANSACTION BASE TRB = BANKTR;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                        %
% Declare all the library entry points to be associated with the         %
% TRHISTORY journal.                                                     %
%                                                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   LIBRARY PROCESSLIB(TITLE="BANKTR/CODE/HOSTLIB.",
     LIBPARAMETER = "TRHISTORY" );

   INTEGER PROCEDURE CREATETRHISTORYUSER(IDSTRING, IDNUM);
     STRING IDSTRING; INTEGER IDNUM;
     LIBRARY PROCESSLIB(ACTUALNAME = "CREATETRUSER");
```

```
    INTEGER PROCEDURE LOGONTRHISTORYUSER(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY PROCESSLIB (ACTUALNAME = "LOGONTRUSER");

    INTEGER PROCEDURE RETURNRESTARTINFO(IDNUM, TROUT);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TROUT;
      LIBRARY PROCESSLIB;

    INTEGER PROCEDURE RETURNLASTRESPONSE(IDNUM, TROUT);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TROUT;
      LIBRARY PROCESSLIB;

    INTEGER PROCEDURE PROCESSTRFROMTANK(IDNUM, TRIN, RESTARTNUM, RESTARTTR);
      INTEGER IDNUM, RESTARTNUM;
      TRANSACTION RECORD (TRB) TRIN, RESTARTTR;
      LIBRARY PROCESSLIB;

    INTEGER PROCEDURE OPENTRHISTORY(USEROPTION, TIMEOUT);
      INTEGER USEROPTION, TIMEOUT;
      LIBRARY PROCESSLIB(ACTUALNAME = "OPENTRBASE");

    INTEGER PROCEDURE CLOSETRHISTORY;
      LIBRARY PROCESSLIB(ACTUALNAME = "CLOSETRBASE");

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% Declare all the library entry points to be associated with the       %
% Tank journal.                                                         %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  LIBRARY TANKLIB(TITLE="BANKTR/CODE/HOSTLIB.");

    INTEGER PROCEDURE CREATETANKUSER(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY TANKLIB (ACTUALNAME = "CREATETRUSER");

    INTEGER PROCEDURE LOGONTANKUSER(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY TANKLIB(ACTUALNAME = "LOGONTRUSER");

    INTEGER PROCEDURE TANKUSERIDSTRING(IDSTRING, IDNUM);
      STRING IDSTRING; INTEGER IDNUM;
      LIBRARY TANKLIB(ACTUALNAME = "TRUSERIDSTRING");

    INTEGER PROCEDURE TANKTRNORESTART(IDNUM, TRIN);
      INTEGER IDNUM;
      TRANSACTION RECORD (TRB) TRIN;
      LIBRARY TANKLIB;
```

```
INTEGER PROCEDURE OPENTANK(USEROPTION, TIMEOUT);
  INTEGER USEROPTION, TIMEOUT;
  LIBRARY TANKLIB(ACTUALNAME = "OPENTRBASE");

INTEGER PROCEDURE CLOSETANK;
  LIBRARY TANKLIB(ACTUALNAME = "CLOSETRBASE");

INTEGER PROCEDURE SEEKTRANSACTION(FILENUM, BLOCKNUM, OFFSET);
  INTEGER FILENUM, BLOCKNUM, OFFSET;
  LIBRARY TANKLIB;

INTEGER PROCEDURE READTRANSACTION (TRREC);
  TRANSACTION RECORD (TRB) TRREC;
  LIBRARY TANKLIB;
  TRANSACTION RECORD (TRB)
      TRIN,
      TROUT,
      RESTARTTR;
  INTEGER     IDNUM,        N,
              RESTARTNUM,   TANKNAMESIZE,
              FILENUM,      BLOCKNUM,
              OFFSET,       CT,
              TANKER,       UPDATER,
              MAXTANKER,    RSLT;
  ARRAY TRHISTORYUSERS[0:99];
  ARRAY SP[0:14];
  LABEL EXIT, LOOP, PRINTLAST;
  STRING ID, FNAME;
  BOOLEAN  ALLDONE;
  EBCDIC ARRAY SPO[0:79];

DEFINE ERR(L) =
     BEGIN
        REPLACE SPO BY "RSLT = ", RSLT FOR  * DIGITS,
                " @ ", LINENUMBER FOR 8 DIGITS, NULL;
        ACCEPT(SPO);
        GO L;
     END#,
     NULL         = 48"00"#,
     NORESTARTREC = 3#,
     REJECTED     = 2#,
     EOF          = 1#;

    MAXTANKER := 99;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %
% Set the library parameter "LIBPARAMETER" for the Tank journal. Then %
% open the TRHISTORY journal for updating and the Tank journal for   %
% reading.                                                           %
%                                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
       SCAN TANKNAME[*] FOR N:99 UNTIL = 0;
     TANKNAMESIZE := 99-N;
     TANKLIBPARAM := STRING(POINTER(TANKNAME,8), TANKNAMESIZE);
     TANKLIB.LIBPARAMETER := TANKLIBPARAM;
     IF RSLT := OPENTRHISTORY(1, 0) > 0 THEN ERR(EXIT);  % Open update.
     IF RSLT := OPENTANK(4, 0) > 0 THEN ERR(EXIT);  % Open for reading.
     ID := TANKLIBPARAM;

%  Create a user of the History file and then log him on.
%  CREATETRHISTORYUSER(ID, RESTARTNUM);
%  NO-OP if not necessary.
IF RSLT := LOGONTRHISTORYUSER(ID, RESTARTNUM) > 0 THEN ERR(EXIT);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                     %
%The following code determines if the program has been restarted after%
%a HALT/LOAD.  If so, it determines the location in the Tank journal  %
%where we should begin reading transactions. It does this by          %
%extracting the file, block, and offset from the items within the     %
%restart transaction record: TANKFILENUM, TANKBLOCKNUM, and           %
%the program TANKOFFSET.  If it was not restarted, start reading      %
%from the beginning of the Tank journal.                              %
%                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

       IF MYJOB.RESTARTED THEN
BEGIN
     REPLACE SPO BY "DETANKING PROCESS RESTARTING", NULL;
          DISPLAY(SPO);
          IF RSLT := RETURNRESTARTINFO(RESTARTNUM, RESTARTTR) =
                     NORESTARTREC    THEN
          BEGIN
             FILENUM := 1;
             BLOCKNUM := OFFSET := 0;  % Start at first record of file.
             IF RSLT := SEEKTRANSACTION(FILENUM,BLOCKNUM,OFFSET) > 0 THEN
                ERR(EXIT);
          END ELSE
             IF RSLT > 0 THEN
                ERR(EXIT)
             ELSE          % A restart record exists.
             BEGIN
                FILENUM := RESTARTTR.TANKFILENUM;
                BLOCKNUM := RESTARTTR.TANKBLOCKNUM;
                OFFSET := RESTARTTR.TANKOFFSET;
                REPLACE SPO BY "LAST GOOD TR FROM TANK AT (",
                     FILENUM FOR  * DIGITS, ",",
                     BLOCKNUM FOR  * DIGITS, ",",
                     OFFSET FOR  * DIGITS, ")", NULL;
                DISPLAY(SPO);
                IF RSLT := SEEKTRANSACTION(FILENUM, BLOCKNUM, OFFSET) > 0
                                           THEN
```

```
                 ERR(EXIT);

                % Now skip last good transaction.

                IF RSLT := READTRANSACTION(TRIN) > 0 THEN ERR(EXIT);
           END;
  END ELSE
  BEGIN
       FILENUM := 1;
       BLOCKNUM := 0;
       OFFSET := 0;  % Start at first record  of file 1.
       IF RSLT := SEEKTRANSACTION(FILENUM, BLOCKNUM, OFFSET) > 0 THEN
            ERR(EXIT);
  END;
  CREATE RESTARTTR.RESTARTDETANKER;

LOOP:
  IF ALLDONE THEN
  BEGIN
       CLOSETANK;
       CLOSETRHISTORY;
       GO PRINTLAST;
  END;

    %  Read a transaction from the Tank journal.
  IF RSLT := READTRANSACTION(TRIN) > 0 THEN
  BEGIN
       IF RSLT = EOF THEN
       BEGIN
            ALLDONE := TRUE;
            GO LOOP;
       END  ELSE
            ERR(EXIT);
  END;

  %  If we read a system transaction ignore it and continue with the
  %  next transaction in sequence.

    IF TRIN.TRFORMAT = TRFORMAT(SYSTEMTR) THEN GO LOOP;
  TANKER := TRIN.TRUSERNUM;
  IF TANKER > MAXTANKER THEN
  UPDATER := 0 ELSE
  UPDATER := TRHISTORYUSERS[TANKER];
  IF UPDATER = 0 THEN
  BEGIN
       IF RSLT := TANKUSERIDSTRING(ID, TANKER) > 0 THEN ERR(EXIT);
       CREATETRHISTORYUSER(ID, UPDATER);  % NO-OP if necessary.
       IF TANKER > MAXTANKER THEN
       RESIZE(TRHISTORYUSERS[*], (MAXTANKER:=TANKER)+1, RETAIN);
       TRHISTORYUSERS[TANKER] := UPDATER;
  END;
```

```
%  Set up the restart record values to be the address of the input
%  transaction and then process the transaction.

RESTARTTR.TANKFILENUM := TRIN.TRFILENUM;
     RESTARTTR.TANKBLOCKNUM := TRIN.TRBLOCKNUM;
     RESTARTTR.TANKOFFSET := TRIN.TROFFSET;
     IF RSLT := PROCESSTRFROMTANK(UPDATER, TRIN, RESTARTNUM, RESTARTTR)
               > 0 THEN

            ERR(EXIT);
     GO LOOP;

   PRINTLAST:
     FILENUM := RESTARTTR.TANKFILENUM;
     BLOCKNUM := RESTARTTR.TANKBLOCKNUM;
     OFFSET := RESTARTTR.TANKOFFSET;
          REPLACE SPO BY "LAST GOOD TR FROM TANK AT (",
               FILENUM FOR  * DIGITS, ",",
               BLOCKNUM FOR  * DIGITS, ",",
               OFFSET FOR  * DIGITS, ")", NULL;
          DISPLAY(SPO);

   EXIT:
 END OF THE DETANKER PROCEDURE;
```

Section 6
Using the Screen Design Facility Plus (SDF Plus) Interface

Screen Design Facility Plus (SDF Plus) is a user interface management system that gives programmers the ability to define a complete form-based user interface for an application system. It is a programming tool for simple and efficient designing and processing of forms. SDF Plus provides form processing that eliminates the need for complicated format language or code, and validates data entered on forms by application users.

The program interface developed for SDF Plus includes

- Extensions that enable you to read and write form records or form record libraries easily
- Extensions that enable you to send and receive form records or form record libraries easily
- Extensions that enable you to invoke form record library descriptions into your program as ALGOL declarations

This section provides information about the extensions developed for SDF Plus. Each extension is presented with its syntax and an example; sample programs are also included.

For an alphabetized list of the extensions, see “Screen Design Facility Plus (SDF Plus) Extensions” in Section 1, “Introduction to ALGOL Program Interfaces.”

Refer to the *Screen Design Facility Plus (SDF Plus) Capabilities Manual* for information defining the concepts and principles of SDF Plus. For information on general implementation and operation considerations, refer to the *Screen Design Facility Plus (SDF Plus) Installation and Operations Guide*. For information on general programming concepts and considerations, refer to the *Screen Design Facility Plus (SDF Plus) Technical Overview*.

SDF Plus can be be used with the Advanced Data Dictionary System (ADDS), and the Communications Management System (COMS). Refer to the specific product documentation for information on the concepts and programming considerations for using these products with SDF Plus. For more information on the extensions used with these products, refer to Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions,” and Section 3, “Using Communications Management System (COMS) Features.”

Understanding SDF Plus Interface Elements

Communication between ALGOL application programs and SDF Plus form record libraries is achieved through either the remote file interface or the COMS interface. Using the remote file interface, you can interact with SDF Plus applications by means of remote files. By using the COMS interface, you can interact with SDF Plus applications through COMS windows and have access to all COMS capabilities and features.

SDF Plus interface elements include

- Form record libraries
- Form records
- Form record numbers
- Transaction types
- Transaction numbers
- ALGOL functions used as SDF Plus extensions

Form Record Libraries

Form record libraries are collections of form records and transaction types. This union is achieved in the data dictionary. Form record libraries can be either retrieved or invoked by the ALGOL program. The form records can then be used in various ALGOL statements to transfer data.

Form Records

Form records are elements of form record libraries. Form records represent records of data. This data is used either to output data from a form or to input data to a form. A form can require several form records; therefore, a one-to-one relationship between forms and form records does not exist.

In some manuals the term “message type” is a synonym for “form record.”

Forms and form processing are established through the use of SDF Plus. The ALGOL program reads and writes data to these forms. This arrangement provides complete separation between data entered on a terminal and actions completed within the program. A user interface can be completely reconstructed without modifying the application program, provided the form records are not changed.

When referenced, a form record must be qualified with the form record library name by which it was invoked. Each form record within a form record library shares the same storage area. The storage area is created large enough to hold the largest form record.

Form Record Numbers

Form record numbers for form records are unique integers assigned at compile time to each form record in a form record library.

In some manuals the term “message type number” is a synonym for “form record number.”

A form record number for a form record library is an attribute of the form record library. This attribute contains the form record number of a specific form record. Form record numbers determine I/O operations for form record libraries, enabling the form record to be specified at run time.

A self-identifying read is used when the executing program has not established which form record in a specific form record library has been read. The program must access the form record number attribute for the form record library to determine the form record that has been read.

A self-identifying write enables the executing program to specifically identify the form record to be written by placing the appropriate form record number value into the form record number attribute of the form record library.

Transaction Types

Transaction types are elements of form record libraries. A transaction type contains a pair of form records: an input form record and an output form record. A transaction type identifies the relationship of the two form records that are under it, namely, the input form record to the transaction type and the output form record from the transaction type.

Transaction Numbers

Transaction numbers are similar to form record numbers. A transaction number is a unique integer assigned at compile time to each transaction type in a form record library.

A transaction number for a form record library is an attribute of the form record library. This attribute contains the transaction number of a specific transaction type. Transaction numbers provide another means of determining I/O operations for form record libraries at run time.

After a self-identifying read, the application program must access the transaction number attribute of the form record library being read to determine the transaction type that has been executed.

Using ALGOL Functions as SDF Plus Extensions

Several ALGOL fuctions have been extended to work with SDF Plus. The DICTIONARY compiler control option, as well as the LENGTH, OFFSET, POINTER, RESIZE, SIZE, STACK option and UNITS functions can be used as SDF Plus extensions.

Additional information relating to these functions is included in Section 2, "Using Advanced Data Dictionary System (ADDS) Extensions."

- DICTIONARY option

  The DICTIONARY compiler control value option establishes the data dictionary to use during compilation. This option is an ADDS extension that can be used when SDF Plus is used with ADDS. A dictionary must be established before the first executable statement. The dictionary specified in the first occurrence of a DICTIONARY option is used as the data dictionary. All other occurrences are ignored.

- LENGTH function

  The LENGTH function returns the length of a specified entity in the designated units.

- OFFSET function

  The OFFSET function returns the number of units that the specified entity is indexed from the beginning of the outermost record in which it is declared.

- POINTER function

  The POINTER function returns a pointer to the specified input.

- RESIZE function

  The RESIZE function changes the size of the array underlying a given record identifier. For SDF Plus form record libraries, the size is given in bytes. The size of the *entire* array is changed, regardless of the record's position in the array.

- SIZE function

  The SIZE function returns the size of the array underlying a given record identifier. For SDF Plus form record libraries, the size is given in bytes. The size returned is an integer representing the size of the *entire* array, regardless of the record's position in the array.

- STACK option

  The STACK compiler control option directs the ALGOL compiler to print data definition descriptions that are imported from SDF Plus during compilation. When set, the stack option causes the ALGOL compiler to include the definitions of imported form record field types, form record types, form record library types, and file types in the compilation.

- UNITS function

  The UNITS function accepts an entity as input and returns, as an integer value, the default unit size expected by the LENGTH and OFFSET functions.

Invoking the Form Record Library

<dictionary form record library declaration>

```
 — DICTIONARY FORMRECORDLIBRARY ————————————————————————————————————————→

  ┌←——————————————————————————— , ——————————————————————————┐
  │                                                         │
→—┴— <form record library ID> —┬————————————————————————┬———┴——————————————┤
                               │                        │
                               └— <entity qualifiers> ——┘
```

Explanation

A form record library is invoked from a data dictionary that is specified with the $SET DICTIONARY option. The form record library was placed in the data dictionary by SDF Plus at the time that the form library dictionary was created.

The DICTIONARY FORMRECORDLIBRARY declaration invokes a form record library with a description retrieved from the dictionary.

The form record library ID is the name by which the allocated record area is recognized within the program and within the compiler. If the entity qualifier is not specified, the form record library ID is used as the default name for both the type and space.

The DICTIONARY FORMRECORDLIBRARY declaration can be declared using a TYPE declaration and invocation. This type of declaration is not normally used.

Refer to Section 2, “Using the Advanced Data Dictionary System (ADDS) Extensions,” for information describing entity qualifiers and TYPE declarations.

Additional information relating to the entity qualifiers construct is included under “Entity Qualifiers” in Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

Examples

In the following example, the form record library titled APPLFORMRECLIB is invoked from the dictionary and is allocated a buffer:

```
DICTIONARY FORMRECORDLIBRARY APPLFORMRECLIB;
```

In the following example, the form record library titled APPLFORMRECLIB is invoked from the dictionary and is allocated a buffer called RECLIB:

```
DICTIONARY FORMRECORDLIBRARY RECLIB (NAME=APPLFORMRECLIB);
```

Using the SDF Plus Remote File Interface

The following paragraphs describe the syntax of the READFORM and WRITEFORM statements. These statements are used to perform I/O operations when interacting with SDF Plus by means of remote files.

READFORM Statement

<readform statement>

```
— READFORM  — ( — <file> — , —┬— <form record> ————————┬— ) ——————————┤
                              └— <form record library> —┘
```

Explanation

The READFORM statement causes a form record to be read from the specified remote file and stored in the specified storage area. Particular form records can be read by designating the form record name. Self-identifying form records are read by specifying the form record library name.

The READFORM statement returns the results of the I/O operation as a Boolean value. If the I/O operation succeeds, the result is FALSE. The file used with this statement must be a remote file. The compiler generates an error message if the file is declared DIRECT.

A specific read of a form record is completed by identifying the form record. The result of the READFORM statement is to store that particular form record in the storage area associated with the form record library.

A self-identifying read is performed by designating the form record library name in the READFORM syntax. The form record returned is determined by the forms processing performed in SDF Plus. A form record number is returned by SDF Plus to be used to determine the form record that was read.

Examples

In the following example, a self-identifying read of a form record library is performed. The form record number field contains the form record number of the form record that was read.

```
READFORM (RMTFILE, APPLFORMRECLIB);
 FORMNUM := APPLFORMRECLIB.FORMRECNUM;
```

In the following example, the form record FORMRECORDA is read from the form record library APPLFORMRECLIB:

```
READFORM (RMTFILE, APPLFORMRECLIB.FORMRECORDA);
```

WRITEFORM Statement

<writeform statement>

```
— WRITEFORM  — ( — <file> ─┬──────────────────────────────┬─ , ───────→
                           ├─ [ — DEFAULT  — ] ───────────┤
                           └─ [ — DATAERROR  — <error #> — ] ┘

→─┬─ <form record> ───────────────┬─ ) ──────────────────────────┤
  ├─ <form record library> ───────┤
  └─ <text length> — , — <text> ──┘
```

<text length>

```
— <arithmetic expression> ──────────────────────────────────────────┤
```

<text>

```
 — <EBCDIC pointer> ──────────────────────────────────────────────────┤
```

Explanation

The WRITEFORM statement causes a form record to be written to a specified remote file. Specific form records can be written by designating the form record name.

The WRITEFORM statement returns the results of the I/O operation as a Boolean value. If the I/O operation succeeds, the result is FALSE.

The file used with this statement must be a remote file. The compiler generates an error message if the file is declared DIRECT.

The DEFAULT option on a WRITEFORM statement causes SDF Plus to use default values when it displays the form. This option is used when the application program does not supply data for the form.

The DATAERROR option on a WRITEFORM statement enables you to respond to a record received from the dictionary with an error indicator instead of another record.

A specific write of a form record is completed by designating the form record. A self-identifying write is performed by designating the form record library name in the WRITEFORM statement and using the form record number attribute to assign the form record number for that form record library. The form record number or transaction number in the form record library must be assigned before the write operation; otherwise, an error occurs.

Using the WRITEFORM statement with the text option causes the contents of a text array to be written to a designated remote file.

Examples

In the following example, a self-identifying write of the form record library APPLFORMRECLIB is performed. The form record number attribute assigns the form record number of the form that is to be written.

```
APPLFORMRECLIB.FORMRECNUM :=
     APPLFORMRECLIB.FORMRECORDB.FORMRECNUM;
 WRITEFORM (RMTFILE, APPLFORMRECLIB);
```

In this example, the form record FORMRECORDA is written from the form record library APPLFORMRECLIB:

```
WRITEFORM (RMTFILE, APPLFORMRECLIB.FORMRECORDA);
```

In the following example, the default values for the form record FORMRECORDA are written:

```
WRITEFORM (RMTFILE [DEFAULT], APPLFORMRECLIB.FORMRECORDA);
```

In the following example, the program responds to the record received with an error indicator:

```
WRITEFORM (RMTFILE [DATAERROR 5], FORMRECLIB);
```

In this example, the first 30 words of the text array T_ARRAY are written to the remote file RMTFILE and displayed in the text area of the form:

```
WRITEFORM (RMTFILE, 30, T_ARRAY);
```

Using the Form Record Number Attribute

<form record number>

```
──┬─ <form record> ──────────┬─ . ─ FORMRECNUM ─────────────────────────────┤
  │                          │
  └─ <form record library> ──┘
```

<form record>

```
─ <form record library> ─ . ─ <form record name> ─────────────────────────┤
```

<form record name>

```
─ <identifier> ───────────────────────────────────────────────────────────┤
```

Explanation

The form record number attribute is used with either individual form records or form record libraries. In some manuals the term "message type number" is used as a synonym for "form record number."

The form record number attribute associated with individual form records is preassigned by SDF Plus at compile time.

The form record number attribute associated with form record libraries is used with self-identifying reads and self-identifying writes.

The form record name must be qualified with the form record library name.

A form record number attribute of a form record library contains the form record number field of the last form record read. This field should be queried after a read of a specific form record to verify that the specific form record was actually read. The transaction number field should be queried after a read of a self-identifying form record to determine the action to be taken.

Changing the form record number attribute of a form record library enables self-identifying writes. The form record number determines the form record that is written.

A form record number attribute of a form record is the preassigned form record number of the specified form record. These numbers are integer constants assigned at compile time.

Attempting to change the form record number attribute of a form record results in an error.

Examples

In the following example, B is assigned the integer value of the form record FORMRECORDB:

```
B :=APPLFORMRECLIB.FORMRECORDB.FORMRECNUM;
```

In this example, the integer value B is assigned as a form record number for a form record that is to be written in a self-identifying write:

```
APPLFORMRECLIB.FORMRECNUM := B;
```

In the following example, the form record number of the form record FORMRECORDB is assigned to be written using a self-identifying write:

```
APPLFORMRECLIB.FORMRECNUM :=
      APPLFORMRECLIB.FORMRECORDB.FORMRECNUM;
```

This example shows an attempt to change the form record number attribute of form record FORMRECORDB. This action results in an error.

```
FORMRECORDB.FORMRECNUM := B;
```

Using the Transaction Number Attribute

<transaction number>

```
─┬─ <transaction type> ────┬─ . ─ TRANSNUM ──────────────────────────────┤
 │                         │
 └─ <form record library> ─┘
```

<transaction type>

```
─ <form record library> ─ . ─ <transaction name> ─────────────────────────┤
```

<transaction name>

```
─ <identifier> ───────────────────────────────────────────────────────────┤
```

Explanation

The transaction number attribute is used with either individual transaction types or form record libraries. Each transaction number is associated with a transaction name.

The transaction name must be qualified with the form record library name.

A transaction number of a form record library contains the transaction number of the last transaction read. This field should be queried after every read to determine what action the program should take. Note that the transaction number uniquely indicates both the form record that was read and the action to take with it, but the same form record can appear in two different transactions. For example, one transaction might return an empty form record that is to be prefilled, while another transaction might return the same form record that now contains data to be processed. Both reads returned the same form record, but the actions to be taken by the application differed. Only the transaction number uniquely indicates which action to take—the form record number is not sufficient in most cases.

Changing the transaction number of a form record library or a transaction type is not permitted. You should use the form record number of the form record library to indicate to SDF Plus the action to take on the next write.

Attempting to change the transaction number attribute of a transaction type results in an error.

Examples

In the following example, G is assigned the integer value of the transaction TRANSACTIONL:

```
G := APPLFORMRECLIB.TRANSACTIONL.TRANSNUM;
```

In this example, the transaction number of a form record library is queried to verify that a specific transaction has just been read. MYFORMRECPTT is a prefill request from SDF Plus to the application program. MYFORMRECPRE is a prefill response from the application program to SDF PLus.

```
IF APPLFORMRECLIB.TRANSNUM = FORMRECLIB.MYFORMRECPTT.TRANSNUM THEN
      APPLFORMRECLIB.FORMRECNUM := FORMRECLIB.MYFORMRECPRE.FORMRECNUM;
```

The next example shows the processing of an update transaction. MYFORMRECTT is an update transaction that transfers data entered by the user from SDF Plus to the application program. FORMRECLIBSR is a standard response. There is one standard response *per library*. The standard response indicates that the application program accepted the update transaction. Use this technique when the application program enables SDF Plus to decide which form to display next.

```
IF FORMRECLIB.TRANSNUM = FORMRECLIB.MYFORMRECTT THEN
      FORMRECLIB.FORRECNUM := FORMRECLIB.FORMRECLIBSR.FORMRECNUM;
```

Using SDF PLUS with COMS

SDF Plus can be used with COMS to take advantage of COMS direct windows. Using SDF Plus with COMS provides enhanced routing capabilities for forms and also permits preprocessing and postprocessing of form records.

Refer to the *Communications Management System (COMS) Programming Guide* for detailed information on the use of the COMS direct window interface. The following guidelines explain the steps to follow when using SDF Plus and COMS together.

Using COMS Input/Output Headers

SDF Plus supports the use of COMS headers. Three fields are defined within the headers for use with SDF Plus. These fields are SDFINFO, SDFFORMRECNUM, and SDFTRANSNUM. A description of each follows.

The SDFINFO field is used to identify specific form message processing requests (on output) or to return form message processing errors (on input). On the output (sending) path, this field can contain the following values:

Value	Explanation
0	Normal form message processing
100	Last transaction error. This value is used for outgoing messages only.
101	Transaction error. Used when more than one transaction error is sent. The application can send multiple messages in which the value of the SDFINFO field is 101. This value is used for outgoing messages only.
200	Text message processing

On the input (receiving) path, this field can contain the following values, which correspond to status information concerning the requested form message processing:

Value	Explanation
0	No error
-100	Form message timestamp mismatch
-200	Incorrect form record number specified on the send operation
-300	Incorrect transaction number specified on the send operation

The SDFFORMRECNUM field is used to designate the form record to be written (on output) or the form record that is to be received (on input).

The SDFTRANSNUM field is meaningful only on input and contains the number of the SDF Plus transaction that was received. This field should not be altered by the user application.

Sending and Receiving Messages

When using SDF Plus and COMS together, follow the usual statements for each product, with the following guidelines:

- COMS input/output headers should be used instead of binary communication descriptions to take advantage of the new features in SDF Plus.
- To send normal messages, the application program must move the value 0 (zero) into the SDFINFO field of the output header. The application program must set the SDFFORMRECNUM field. The form record library must then be passed as the message area construct in a SEND statement.
- To receive a message, the application program must do the following:
  - If the SDFINFO field contains a value less than 0 (zero), this field also contains an error code that indicates a problem with message processing. In addition, the FUNCTION-INDEX field of the input header contains the value 100.
  - If the SDFINFO field contains the value 0 (zero), the application program can query the form record number and transaction number attributes for the form record library from the SDFFORMRECNUM and SDFTRANSNUM fields of the input header.

Sending Transaction Errors

SDF Plus supports the ability to send error codes in response to incorrect data received by the user application. These error codes are sent as integer values, which are used by SDF Plus to process a user-defined error procedure for the form record library.

To send transaction error codes, the user application must do the following:

1. Move the value 100 or 101 into the SDFINFO field of the output header.
2. Move the value of the transaction error into the SDFFORMRECNUM field of the output header.
3. Move the SDFTRANSNUM field from the input header to the output header.
4. Send the output header to display the message.

The user application can send any arbitrary message area along with the output header. SDF Plus only processes the information within the output header.

Example

In this example, INX contains the number of the transaction error.

```
COMS_OUT.SDFINFO := 100;
COMS_OUT.SDFFORMRECNUM := INX;
COMS_OUT.SDFTRANSNUM := COMS_IN.SDFTRANSNUM;
COMS_OUT.TEXTLENGTH := COMS_IN.TEXTLENGTH;
SEND(COMS_OUT,COMS_IN.TEXTLENGTH,APPLFORMRECLIB);
```

Sending Text Messages

SDF Plus supports the ability to send text messages for display on the text area of a form.

To send a text message, the user application must do the following:

- Move the value 200 into the SDFINFO field of the output header.
- Move the text message into a message area to be sent through COMS.
- Use the SEND statement to send the text message.

The text message will be displayed when the next form is displayed.

For information about the extensions used with COMS, refer to Section 3, “Using Communication Management System (COMS) Features.”

Example

In this example, literal text is moved into the message area. The form to display the text message is FORM1.

```
COMS_OUT.SDFINFO := 200;
REPLACE STEXT[0] BY "This is an example of application text" FOR 38;
SEND (COMS_OUT, 38, STEXT);
COMS_OUT.SDFINFO := 0;
COMS_OUT.SDFFORMRECNUM := FORM1.FORMRECNUM;
SEND (COMS_OUT, COMS_IN.TEXTLENGTH, FORM1);
```

SDF PLUS Sample Programs

Example 1 highlights the different uses of the SDF Plus program interface.

Example 2 demonstrates the use of the SDF Plus program interface with COMS.

Example 1: General Use of SDF Plus Program Interface

The following is a sample program showing different uses of the SDF Plus program interface. For information about handling remote file errors in an application program, refer to the *SDF Plus Technical Overview*.

In this program, a READFORM statement is performed. The transaction number attribute is then interrogated to determine the form record that was read. The appropriate response is then indicated by setting the form record number attribute.

The program accepts two string or binary inputs from a remote file, concatenates or adds them together, and returns the original inputs and the result as outputs on the terminal screen. The form record library was created in SDF Plus.

```
$SET LIST STACK
$SET DICTIONARY = "SDFPLUSDICT"
BEGIN
    FILE REMFILE (BLOCKSIZE  = 2040,
                  KIND       = REMOTE,
                  MAXRECSIZE = 2040,
                  BLOCKSTRUCTURE = EXTERNAL,
                  MYUSE      = IO,
                  UNITS      = CHARACTERS);
    DICTIONARY FORMRECORDLIBRARY DTCOMPLEXLIB
                 (DIRECTORY = "SMITH",
                  VERSION   = 1);
    BOOLEAN       END_PGMV;
    EBCDIC ARRAY  MYSTRING1[0:24],
                  MYSTRING2[0:24],
                  MYSTRING[0:49];
    INTEGER       MYBNUMBER,
                  MYBNUMBER1,
                  MYBNUMBER2;
    DEFINE        BLANK = " "#;
 %
    PROCEDURE INITIALIZE_ALL;
       BEGIN        REPLACE MYSTRING1 BY BLANK FOR 25;
       REPLACE MYSTRING2 BY BLANK FOR 25;
       REPLACE MYSTRING BY BLANK FOR 50;
       MYBNUMBER := 0;
       MYBNUMBER1 := 0;
       MYBNUMBER2 := 0;
       END; % INITIALIZE_ALL
```

```
%
   PROCEDURE CONCATSTRINGS;
      BEGIN       REPLACE MYSTRING1 BY
         DTCOMPLEXLIB.APUTALPHAS.PASTRING1 FOR
          LENGTH(DTCOMPLEXLIB.APUTALPHAS.PASTRING1);
      REPLACE MYSTRING2 BY
         DTCOMPLEXLIB.APUTALPHAS.PASTRING2 FOR
         LENGTH(DTCOMPLEXLIB.APUTALPHAS.PASTRING2);
      REPLACE MYSTRING BY MYSTRING1 FOR 25, MYSTRING2 FOR 25;
      END; % CONCATSTRINGS
%
   PROCEDURE BINARYADD;
      BEGIN       MYBNUMBER1 :=
        INTEGER(DTCOMPLEXLIB.APUTBINARY.PBNUMBER1);
      MYBNUMBER2 :=
         INTEGER(DTCOMPLEXLIB.APUTBINARY.PBNUMBER2);
      MYBNUMBER := MYBNUMBER1 + MYBNUMBER2;
      END; % BINARYADD
%
   PROCEDURE GETBINARY;
      BEGIN
      DTCOMPLEXLIB.AGETBINARYPRE.GBNUMBER1 := MYBNUMBER1;
      DTCOMPLEXLIB.AGETBINARYPRE.GBNUMBER2 := MYBNUMBER2;
      DTCOMPLEXLIB.AGETBINARYPRE.GBNUMBER  := MYNUMBER;
      END; % GETBINARY
%
   PROCEDURE GETALPHAS;
      BEGIN
      REPLACE DTCOMPLEXLIB.AGETALPHASPRE.GASTRING1
             BY MYSTRING1 FOR 25;
      REPLACE DTCOMPLEXLIB.AGETALPHASPRE.GASTRING2
             BY MYSTRING2 FOR 25;
      REPLACE DTCOMPLEXLIB.AGETALPHASPRE.GASTRING
             BY MYSTRING FOR 50;
      END; % GETALPHAS
%
   PROCEDURE MAIN_FORM;
      BEGIN
      LABEL MAIN_FORM-EXIT;
      IF READFORM (REMFILE, DTCOMPLEXLIB) THEN
           BEGIN % true result implies IO operation failed
           WRITE(REMFILE,//,"READFORM ERROR");
           END_PGMV := TRUE;
           GO MAIN_FORM_EXIT;
           END;
      CASE DTCOMPLEXLIB.TRANSNUM OF
         BEGIN
         (DTCOMPLEXLIB.AGETALPHASPTT.TRANSNUM):
             BEGIN
             DTCOMPLEXLIB.FORMRECNUM :=
                DTCOMPLEXLIB.AGETALPHASPRE.FORMRECNUM;
             GETALPHAS;
             END;
         (DTCOMPLEXLIB.AGETBINARYPTT.TRANSNUM):
             BEGIN
             DTCOMPLEXLIB.FORMRECNUM :=
                DTCOMPLEXLIB.AGETBINARYPRE.FORMRECNUM;
             GETBINARY;
             END;
         (DTCOMPLEXLIB.APUTALPHASTT.TRANSNUM):
             BEGIN
             DTCOMPLEXLIB.FORMRECNUM :=
                DTCOMPLEXLIB.DTCOMPLEXLIBSR.FORMRECNUM;
             CONCATSTRINGS;
             END;
         (DTCOMPLEXLIB.APUTBINARYTT.TRANSNUM):
             BEGIN
             DTCOMPLEXLIB.FORMRECNUM :=
                DTCOMPLEXLIB.DTCOMPLEXLIBSR.FORMRECNUM;
             BINARYADD;
         (DTCOMPLEXLIB.AGETALPHASTT.TRANSNUM):
         (DTCOMPLEXLIB.AGETBINARYTT.TRANSNUM):
             BEGIN
             DTCOMPLEXLIB.FORMRECNUM :=
                DTCOMPLEXLIB.DTCOMPLEXLIBSR.FORMRECNUM;
             END;
         ELSE:
             BEGIN
             WRITE(REMFILE,//,"UNKNOWN TRANSACTION");
             END_PGMV := TRUE;
             GO MAIN_FORM_EXIT;
             END;
         END;  % CASE
      IF WRITEFORM (REMFILE, DTCOMPLEXLIB) THEN
           BEGIN   % true result implies IO operation failed
           WRITE(REMFILE,//,"WRITEFORM ERROR");
           END_PGMV := TRUE;
           END;
   MAIN_FORM_EXIT:
       END MAIN_FORM;
%
   INITIALIZE_ALL;
   DO MAIN_FORM
   UNTIL END_PGMV;
   END.
```

Example 2: Using COMS with the SDF Plus Program Interface

This sample program uses the same programming logic as that in Example 1. However, this COMS interface example shows the application program interacting with users through a COMS window. The SDFTRANSNUM field, which is located in the COMS input header, is interrogated to determine the form record that was read. The response is indicated by setting the SDFFORMRECNUM field, located in the COMS output header. Additionally, the program accepts two string or binary inputs from COMS into a message area declared in the program.

Refer to the *COMS Programming Guide* for a discussion of COMS programming issues and a detailed explanation of the COMS features and functions available with each version of COMS.

```
$SET LIST STACK
$SET DICTIONARY "SDFPLUSDICT"
BEGIN
    DICTIONARY FORMRECORDLIBRARY DTCOMPLEXLIB
                           ( DIRECTORY = "SMITH",
                             STATUS = ANY,
                             VERSION = 1 );
 %
    BOOLEAN       END_PGMV;
    EBCDIC ARRAY  MYSTRING1[0:24],
                  MYSTRING2[0:24],
                  MYSTRING[0:49],
                  STEXT[0:32];
    INTEGER       MYBNUMBER,
                  MYBNUMBER1,
                  MYBYNUMBER2;
    DEFINE        BLANK = " "#;
 %
    % COMS declarations
    INPUTHEADER   COMS_IN;
    OUTPUTHEADER  COMS_OUT;
    EBCDIC ARRAY  MSG[0:255];
    REAL          SDF_AGENDA;
     DEFINE        EOF_NOTICE = 99#;
 %
    LIBRARY SERVICE_LIB
         (LIBACCESS = BYFUNCTION,
          FUNCTIONNAME = "COMSSUPPORT.",
            LIBPARAMETER = "02");
 %
    INTEGER PROCEDURE GET_DESIGNATOR_USING_NAME
                                    (ENTY_NAME,
                                     ENTY_TYPE,
                                     ENTY_DESIGNATOR);
               VALUE ENTY_TYPE;
               EBCDIC ARRAY ENTY_NAME[0];
               REAL  ENTY_DESIGNATOR;
```

```
              INTEGER ENTY_TYPE;
      LIBRARY SERVICE_LIB;
   %
      PROCEDURE INITIALIZE_COMS;
         BEGIN
         % get the title of COMS
         COMSSUPPORT.LIBACCESS := VALUE(BYTITLE);
         REPLACE MSG BY MYSELF.EXCEPTIONTASK.EXCEPTIONTASK.NAME;
         COMSSUPPORT.TITLE := STRING(MSG[0],256);
         ENABLE(COMS_IN,"ONLINE");
         % get the agenda designator
         REPLACE MSG[0] BY "JONES", " " FOR 251;
         GET_DESIGNATOR_USING_NAME(MSG,3,SDF_AGENDA);
         END;
  % INITIALIZE_COMS;
   %
      PROCEDURE INITIALIZE_ALL;
         BEGIN
         REPLACE MYSTRING1 BY BLANK FOR 25;
         REPLACE MYSTRING2 BY BLANK FOR 25;
         REPLACE MYSTRING BY BLANK FOR 50;
         MYBNUMBER := 0;
         MYBNUMBER1 := 0;
         MYBNUMBER2 := 0;
         END; % INITIALIZE_ALL
   %
      PROCEDURE CONCATSTRINGS;
         BEGIN
         REPLACE MYSTRING1 BY
            DTCOMPLEXLIB.APUTALPHAS.PASTRING1 FOR
            LENGTH(DTCOMPLEXLIB.APUTALPHAS.PASTRING1);
         REPLACE MYSTRING2 BY
            DTCOMPLEXLIB.APUTALPHAS.PASTRING2 FOR
            LENGTH(DTCOMPLEXLIB.APUTALPHAS.PASTRING2);
         REPLACE MYSTRING BY MYSTRING1 FOR 25, MYSTRING2 FOR 25;
         END; % CONCATSTRINGS
   %
      PROCEDURE BINARYADD;
         BEGIN
         MYBNUMBER1 :=
            INTEGER(DTCOMPLEXLIB.APUTBINARY.PBNUMBER1);
         MYBNUMBER2 :=
            INTEGER(DTCOMPLEXLIB.APUTBINARY.PBNUMBER2);
         MYBNUMBER := MYBNUMBER1 + MYBNUMBER2;
         END; % BINARYADD
   %
      PROCEDURE GETBINARY;
         BEGIN
         DTCOMPLEXLIB.AGETBINARYPRE.GBNUMBER1 := MYBNUMBER1;
         DTCOMPLEXLIB.AGETBINARYPRE.GBNUMBER2 := MYBNUMBER2;
         DTCOMPLEXLIB.AGETBINARYPRE.GBNUMBER := MYBNUMBER;
         END; % GETBINARY;
```

```
 %
     PROCEDURE GETALPHAS;
       BEGIN
       REPLACE DTCOMPLEXLIB.AGETALPHASPRE.GASTRING1
              BY MYSTRING1 FOR 25;
       REPLACE DTCOMPLEXLIB.AGETALPHASPRE.GASTRING2
              BY MYSTRING2 FOR 25;
       REPLACE DTCOMPLEXLIB.AGETALPHASPRE.GASTRING
              BY MYSTRING FOR 50;
       END; % GETALPHAS
 %
    PROCEDURE SENDTEXT;
       BEGIN
       REPLACE STEXT[0] BY
           "-- THIS IS A SEND TEXT TEST -- " FOR 31;
       COMS_OUT.SDFINFO := 200;
       COMS_OUT.TEXTLENGTH := 31;
       SENDSTATUS := SEND (COMS_OUT, 31, STEXT );
       END; % SENDTEXT
 %
 %============================================================%
 %                     MAIN PROGRAM                           %
 %============================================================%
 %
LABEL MAIN_EXIT;
 %
INITIALIZE_ALL;
INITIALIZE_COMS;
 %
DO
BEGIN
RECEIVE
(COMS_IN, DTCOMPLEXLIB);
IF COMS_IN.STATUSVALUE NEQ EOF_NOTICE THEN
    BEGIN
    IF COMS_IN.FUNCTIONSTATUS GEQ 0 THEN
       BEGIN
        COMS_OUT.DESTCOUNT := 1;
        COMS_OUT.DESTINATIONDESG := COMS_IN.STATION;
        COMS_OUT.SDFTRANSNUM := COMS_IN.SDFTRANSNUM;
        COMS_OUT.AGENDA := SDF_AGENDA;
 %
        CASE COMS_IN.SDFTRANSNUM OF
        BEGIN
        (DTCOMPLEXLIB.AGETALPHASPTT.TRANSNUM);
            BEGIN
            COMS_OUT.SDFFORMRECNUM :=
                     DTCOMPLEXLIB.AGETALPHASPRE.FORMRECNUM;
            GETALPHAS;
            SENDTEXT;
            END;
        (DTCOMPLEXLIB.AGETBINARYPTT.TRANSNUM):
                BEGIN
                COMS_OUT.SDFFORMRECNUM :=
                         DTCOMPLEXLIB.AGETBINARYPRE.FORMRECNUM;
                GETBINARY;
                END;
            (DTCOMPLEXLIB.APUTALPHAS.TRANSNUM):
                BEGIN
                COMS_OUT.SDFFORMRECNUM :=
                         DTCOMPLEXLIB.DTCOMPLEXLIBSR.FORMRECNUM;
                CONCATSTRINGS;
                END;
            (DTCOMPLEXLIB.APUTBINARYTT.TRANSNUM):
                BEGIN
                COMS_OUT.SDFFORMRECNUM :=
                         DTCOMPLEXLIB.DTCOMPLEXLIBSR.FORMRECNUM;
                BINARYADD;
                END;
            (DTCOMPLEXLIB.AGETALPHASTT.TRANSNUM):
            (DTCOMPLEXLIB.AGETBINARYTT.TRANSNUM):
                BEGIN
                DTCOMPLEXLIB.FORMRECNUM :=
                         DTCOMPLEXLIB.DTCOMPLEXLIBSR.FORMRECNUM;
                END;
            ELSE:
                BEGIN
                END_PGMV := TRUE;
                GO MAIN_EXIT;
                END;
            END; % CASE
     %
            % set up COMS output header
            COMS_OUT.TEXTLENGTH := COMS_IN.TEXTLENGTH;
            COMS_OUT.SDFINFO := 0;
            SEND (COMS_OUT, COMS_IN.TEXTLENGTH,DTCOMPLEXLIB);
            END;  % COMS_IN.FUNCTIONSTATUS GEQ 0
        END  % COM_IN.STATUSVALUE NEQ EOF_NOTICE
    ELSE
        END_PGMV := TRUE;
    END
     %
    MAIN_EXIT:
    UNTIL END_PGMV;
    END.
```

Section 7
Using the Semantic Information Manager (SIM) Interface

Semantic Information Manager (SIM) is a database management system that provides for the control, retrieval, and maintenance of data.

This section explains how to use ALGOL to manipulate data in an SIM database and provides samples of typical applications used with SIM. It contains discussions of the ALGOL extensions developed for the following functions:

- Declaring a SIM database
- Mapping SIM types into ALGOL
- Declaring or discarding a query to a SIM database
- Declaring an entity reference variable to explicitly hold a reference to a SIM database entity
- Opening and closing a SIM database
- Assigning SIM database attributes
- Using statements for transaction state and transaction points
- Using selection expressions to determine entities or values within SIM database statements
- Selecting a set of entities and associating it with the query
- Altering level values in a transitive closure retrieval
- Retrieving entities from the SIM database
- Updating entities with single- or multiple-statement updates
- Exception handling of SIM statements

Refer to the *InfoExec Semantic Information Manager (SIM) Programming Guide* for detailed information on SIM programming considerations. Consult the *InfoExec Semantic Information Manager (SIM) Technical Overview* for SIM concepts. For information on defining files and elements in SIM, refer to the *InfoExec ADDS Operations Guide.*

For programming considerations when using SIM and COMS together, consult the *InfoExec SIM Programming Guide.*

The SIM interface uses the following ALGOL type 2 reserved words:

ABORTTRANSACTION	DMMATCH	INVERSE
ALL	DMMAX	MODIFY
APPLYINSERT	DMMIN	NONE
APPLYMODIFY	DMNEXTEXCEPTION	ORDER
BINARY	DMPOS	ORDERING
CANCELTRPOINT	DMPRED	QUERY
COLLATING	DMRECORD	RECORD
CURRENT	DMRPT	REFERENCE
DISCARD	DMSQRT	RETRIEVE
DMABS	DMSUCC	SAVETRPOINT
DMAVG	DMSUM	SELECT
DMCHR	DMTRUNC	SEMANTIC
DMCONTAINS	ENTITY	SETTOCHILD
DMCOUNT	EQV_EQL	SETTOPARENT
DMEQUIV	EQV_GEQ	SOME
DMEXECEPTIONINFO	EQV_GTR	STARTINSERT
DMEXCEPTIONMSG	EQV_LEQ	STARTMODIFY
DMEXCLUDES	EQV_LSS	SUBROLE
DMEXISTS	EQV_NEQ	TRANSITIVE
DMEXT	EXCLUDE	TYPE
DMISA	EXISTS	USING
DMLENGTH	INCLUDE	WHERE

The SIM, Data Management System II (DMSII), Communications Management System (COMS), and Advanced Data Dictionary System (ADDS) interfaces can be used within the same program. For example, both SIM and DMSII data bases can be accessed in the same program. COMS and SIM work together to provide a recoverable transaction system for a SIM database. The DICTIONARY and RANGECHECK options of the ADDS interface can also be used as SIM extensions.

Note: *If the DICTIONARY compiler control option does not appear before the first executable statement, SIM defaults to the dictionary titled "DATADICTIONARY" and the program might not compile properly.*

Note that if DMSII and SIM databases are accessed in the same program, each database must be invoked, manipulated, and processed with its own extensions. Use DMSII and BDMSALGOL extensions for DMSII databases. Use SIM extensions for SIM databases.

Additional information relating to ADDS, COMS and DMSII is included in Section 3, "Using Communications Management System (COMS) Features," and Section 4, "Using the Data Management System II (DMSII) Interface."

Using ADDS Extensions as SIM Extensions

ADDS can be used to define a SIM database. However, different methods of data retrieval are used when directly interfacing to ADDS and when using SIM to interface to ADDS.

- When a program accesses ADDS directly, the compiler links directly to ADDS to get the non-SIM data descriptions.
- When a program accesses SIM, it indirectly accesses ADDS. The compiler does not link directly to ADDS.

If a program accesses both ADDS and SIM, it gets two links to ADDS; one direct and one indirect. Tracking data is not integrated.

The DICTIONARY and the RANGECHECK compiler control options, as well as the LENGTH, OFFSET, POINTER, and UNITS functions can also be used as SIM extensions. More detailed information about the ADDS extensions that are used with SIM is included in Section 2, "Using Advanced Data Dictionary System (ADDS) Extensions."

Purpose of the Dictionary Option

The DICTIONARY compiler control value option establishes the data dictionary to use during compilation. A dictionary must be established before the first executable statement. The dictionary specified in the first occurrence of a DICTIONARY option is used as the data dictionary. All other occurrences are ignored. If a dictionary is not specified, SIM defaults to the dictionary titled "DATADICTIONARY" and the program may not compile properly.

Purpose of the Rangecheck Option

The RANGECHECK option is a Boolean option that causes the compiler to generate code that performs range checking at run time on values that were not known at compile time. The option is set by default. A run-time fault occurs if a value fails a range check; the program is discontinued and an "Invalid Operation" is reported.

Purpose of Functions

The following ADDS functions can be used with SIM. All of these functions can be used with DMRECORDs.

- LENGTH function

  The LENGTH function returns the length of a specified entity in the designated units.

- OFFSET function

  The OFFSET function returns the number of units that the specified entity is indexed from the beginning of the outermost record in which it is declared.

- POINTER function

  The POINTER function returns a pointer to the specified input.

- RESIZE function

  The RESIZE function changes the size of the array underlying a given record identifier. For SIM DMRECORDs, the size is given in bytes. The size of the *entire* array is changed, regardless of the record's position in the array.

- SIZE function

  The SIZE function returns the size of the array underlying a given record identifier. For SIM DMRECORDs, the size is given in bytes. The size returned is an integer representing the size of the *entire* array, regardless of the record's position in the array.

- UNITS function

  The UNITS function accepts an entity as input and returns, as an integer value, the default unit size expected by the LENGTH and OFFSET functions.

Declaring a SIM Database

<database declaration>

```
── SEMANTIC ── DATABASE ── <database reference> ──────────────────────┤
```

<database reference>

```
── <database name> ─┬─────────────────────────┬─ : ── ( ────────────────→
                    └─ <entity qualifiers> ─┘

→── <classID list> ── ) ────────────────────────────────────────────────┤
```

<classID list>

```
   ┌←──────────────── , ───────────────┐
───┴─┬────────────────────┬─ <class ID> ─┴──────────────────────────────┤
     └─ <alias ID> ── = ─┘
```

<alias ID>

```
── <identifier> ────────────────────────────────────────────────────────┤
```

<class ID>

```
── <identifier> ────────────────────────────────────────────────────────┤
```

Explanation

A SEMANTIC DATABASE declaration specifies the SIM database to be used in a query. Only included classes and attributes belonging to the included classes can be used in a query.

Multiple SIM databases can be declared in a program. A SIM database can be declared more than once in the same program. Refer to the *InfoExec SIM Programming Guide* for the SIM-defined limit to the number of SIM databases that can be declared in one program.

SIM and DMSII databases can be used in the same program, including separately compiled programs that are bound. Each database must be declared in its own DATABASE declaration. A DMSII database is available only from BDMSALGOL.

Note that if DMSII and SIM databases are accessed in the same program, each database must be invoked, manipulated, and processed with its own extensions. Use DMSII and BDMSALGOL extensions for DMSII databases. Use SIM extensions for SIM databases.

Two different databases can be updated in the same program only if they are the same physical database.

Before a SIM database can be used in a SIM statement, it must be declared and opened. Also, an access method must be stated. The ADDS for the database must be specified in the DICTIONARY compiler control value option that appears before the first executable statement.

Any hyphens in the identifier of an entity are translated to underscores by the ALGOL compiler before the identifier is passed to SIM.

The prefix “SEMANTIC” identifies the database as a SIM database. If the prefix is not used, a DMSII database is assumed.

A SIM database can be invoked more than once. The database name construct is the name of the declared SIM data base. If there are multiple SIM databases involved in a query, the entity qualifiers are used to resolve any ambiguity. The database name must be unique within scope rules.

The class ID list construct is a list of the SIM database classes used by the program. If the program accesses more than one SIM database, naming conflicts can occur among the database classes. Using the alias ID construct ensures uniqueness of the class names. If only one SIM database is declared in the program or if there are no conflicts, no alias is needed.

Additional information relating to the entity qualifiers construct is included under “Entity Qualifiers” in Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

Additional information relating to SIM database declarations is included under “SIM OPEN statement” in this section and “Invoking a DMSII Database” in Section 4, “Using the Data Management System II (DMSII) Interface.” Related information is also available under “DICTIONARY Option: Establishing a Data Dictionary” and “Entity Qualifiers” in Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

Examples

In the following example, the SIM database UNIVDB is declared. It is qualified by its name and version. The class list includes the classes INSTRUCTOR and COURSE. An alias, CLASS, is equated with the class COURSE. Note the colon (:) preceding the class list.

```
SEMANTIC DATABASE UNIVDB
   ( NAME = UNIVERSITYDB, VERSION = 103 ) :
   ( INSTRUCTOR, CLASS = COURSE );
```

In the following example, the same database is declared without a repository. The database qualifiers specify that the database files can be located under the DEV usercode on the TEST pack.

```
SEMANTIC DATABASE UNIVDB
   ( NAME = UNIVERSITYDB,
     USERCODE = "DEV", PACKNAME = "TEST" ):
   ( PERSON, TEACHER = INSTRUCTOR );
```

Mapping SIM Types Into ALGOL

SIM data items are normally mapped from the SIM database into an ALGOL program according to the default types shown below in Table 7–1. Fields, however, can be declared in DMRECORDs with any of the permitted types. When this occurs, the compiler emits code to perform the mapping to the default type.

Table 7–1. Mapping SIM Types into ALGOL

SIM Type	Default Type	Permitted Type
Integer, Date, Time, Subrole	Integer	Real, Double, Integer
Real	Real	Double, Integer, Real
Number	Double, Integer	Integer, Real, Double
Character	EBCDIC array [0:0]	EBCDIC array[0:n]
Fixed & Variable String, Symbolic	EBCDIC array[0:n]	EBCDIC
KANJI Character	EBCDIC array[0:n*2]	None
KANJI String	Coerced into EBCDIC	None
Boolean	Boolean	Real, Boolean
Compound Attribute	Record	Record
Entity Reference	Entity Reference	Entity Reference
Range	Base type	Base type
Enumeration	Base type	Base type

The SIM type “date” is mapped into an ALGOL integer. In an arithmetic form, the date can be used with arithmetic operators such as MOD and DIV. The date is in the format “YYYYMMDD”. The format is explained in the following table.

Symbol	Meaning
YYYY	A four-digit representation of the year
MM	A two-digit representation of the month
DD	A two-digit representation of the day

For example, “19891003” is October 3, 1989.

The SIM type “time” is also mapped into an ALGOL integer. The time is in the format “HHMMSS”. The format is explained in the following table.

Symbol	Meaning
HH	A two-digit representation of the hour
MM	A two-digit representation of the minutes
SS	A two-digit representation of the seconds

For example, “102359” is 10:23:59. Note that if you use a value that is less than six digits, leading zeroes are assumed.

As a default, the SIM type “number” is mapped as either a double or an integer. The default is double when the number is greater than 11 digits. The default is integer when the number is less than or equal to 11 digits.

If you use a string type or a symbolic type, remember that the upper bound of the default ALGOL type is set up to handle the largest possible string or symbolic value.

- For string types, the default upper bound is equal to the maximum string length permitted by the compiler minus one.
- Symbolic types have a fixed length of 30 regardless of the symbolic value. The default upper bound is therefore 29.

If you declare an upper bound that is less than the default, the compiled program displays a warning message when an associated SELECT statement is executed.

SIM types are explained in the *InfoExec Semantic Information Manager (SIM) Technical Overview*. ALGOL types, except Record and Entity Reference, are explained in Volume 1 of this manual.

Additional information relating to SIM types is included under “Declaring an Entity Reference Variable Data Type,” and “Type Declaration and Invocation for SIM” in this section. Related information is also available in the definintion of record types under “ALGOL Data Types for ADDS” in Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

Queries

A query refers to both inquiry and update requests to a SIM database. A query consists of the query statement, the query variable, and the DMRECORD.

A query statement is sent to SIM to instruct the SIM database about the action to be performed. Query statements are constructed by the compiler from the SELECT, MODIFY, INSERT, and DELETE statements. (The multiple-statement MODIFY and INSERT update assignments are constructed as query statement fragments.) The query statements are precompiled and stored with the object code until run time, when SIM acts on the precompiled statements. One query statement can be associated with more than one query variable.

The query variable represents an active query. It contains information about the state of the query. The query variable can be associated with more than one query statement, but only one query variable can be active at any time.

The DMRECORD gives the format of the data to be retrieved. A DMRECORD can be used for multiple query statements, as long as the structure of the record is compatible with the data to be retrieved.

Perform the following steps to create and use a query:

1. Declare and open the SIM database.
2. Declare the query variable and all other needed variables.
3. If desired, put the program in transaction state.
4. Execute the query. A query consists of statements that select, retrieve, and manipulate the entities.
5. Take the program out of transaction state as needed. When the query is no longer needed, close it with a DISCARD statement.

The DATABASE declaration specifies the SIM database. Only the classes included in the declaration can be used in queries. The OPEN statement makes the SIM database accessible and specifies an access mode.

The BEGINTRANSACTION statement initiates transaction state.

The SELECT, SETTOPARENT, SETTOCHILD, and RETRIEVE statements are used to select the entities for the query and to retrieve the data.

The SELECT statement is used to associate a selected set of entities with the query and to map SIM database attributes to previously defined DMRECORDs. Selection expressions can be used within the SELECT statement to specify which entities are to be included in the selected set.

For all queries, a selection expression is used to identify the set of entities upon which the query operates. The selection expression serves to narrow the group of entities in the perspective class and classes of interest for the scope of the query.

A global selection expression applies to the whole query. A local selection expression applies only to a specific entity-valued attribute (EVA).

The RETRIEVE statement is used to retrieve the data.

The SETTOCHILD and SETTOPARENT statements are used to manipulate the levels involved in a selection and retrieval in transitive closure.

The query can be closed by ending the transaction state with an ENDTRANSACTION statement (if the selection occurred within transaction state), by closing the SIM database with a CLOSE statement, or discarding the current query with a DISCARD statement.

Query variables can be passed as by-name parameters. Program variables and expressions can be used within query statements. DMRECORDs cannot be the target of an assignment; however, database attributes to be modified or inserted that are associated with a query variable can be the target of a SIM database assignment.

The SIM statements and the data management (DM) functions described in this section are used to manipulate the query and the retrieved data.

SIM supports a variety of functions which, when used within a query, are evaluated during the course of the query execution by the SIM system. These functions are explained in this section.

Additional information relating to SIM queries is included under “SIM Statements,” “Using Data Management Functions and Expressions,” “Type Declaration and Invocation for SIM” and “Declaring an Entity Reference Variable Data Type” in this section.

Retrieval and Update Queries

Retrieval queries are always used with the SELECT statement.

A retrieval query can span one or more classes. Generally, there is one class that a query is directed from, the perspective class. Additional classes are viewed in relation to the perspective class. The relationships are maintained via entity-valued attributes (EVAs).

When there are multiple classes of interest in a retrieval query, the classes must be connected so that common entities can be selected. For example, if STUDENT and INSTRUCTOR are both classes, it is possible to find students and instructors with the same name or with the same age.

The layout of retrieved data is specified as part of the query and does not need to bear any direct resemblance to the physical or conceptual layout of the data. However, it must be consistent with the conceptual layout, as determined by SIM.

Update queries are used in transaction state to update entities using the attribute assignment statements. Update queries can be used for limited purposes with the SELECT statement.

There are two different forms of update statements, single- and multiple-statement updates.

- With a single-statement update, all assignments are executed at the point where a MODIFY or INSERT statement is encountered.
- The multiple-statement update enables dynamic specification of the attribute assignments to be applied for the specified update. At run time, the multiple-statement update is dynamically delimited by the START and APPLY syntax of the MODIFY and INSERT statements.

Additional information relating to retrieval and update queries is included under "Declaring a Query Data Type," "Queries," "RETRIEVE Statement," "SELECT Statement," "SIM MODIFY statement," "SIM INSERT statement," and "SIM DELETE statement" in this section.

Declaring a Query Data Type

<query declaration>

```
── QUERY ── <query ID> ── ( ─┬─ <class ID> ──────────┬─ ) ─────────────┤
                             ├─ <DMRECORD ID> ───────┤
                             └─ <DMRECORD type ID> ──┘
```

<query ID>

```
── <identifier> ─────────────────────────────────────────────────────────┤
```

Explanation

The QUERY declaration specifies the name of the query variable and the classes or types used in the query.

Additional information relating to the query declaration is included under “Retrieval and Update Queries,” and “Queries” in this section.

The query ID construct identifies the query.

The class ID construct identifies the class to be modified in a multiple-statement update. A class cannot be accessed unless it has been declared in the class list of an opened SIM database.

In retrieval queries and in update queries that use a SELECT statement, the class ID is used to establish the current path in the SIM database. The query ID can be passed as an argument to the CURRENT function.

For a retrieval query, the construct DMRECORD type ID identifies a previously defined DMRECORD type. This construct is the user-defined name associated with the format.

If you use a query that is *not* declared in the outer block of the program, use the DISCARD statement to close that query when it is no longer needed. This statement prevents the program from unnecessarily consuming and failing to allow reuse of valuable resources from SIM and potentially causing a run-time limit error.

Additional information relating to the class ID construct is included under “Declaring a SIM Database” in this section. Information on the DMRECORD ID construct is included under “Declaring DMRECORDS” in this section. Information on the DMRECORD type ID contruct is included under “Type Declaration and Invocation for SIM” in this section.

Additional information relating to SIM queries is included under “Type Declaration and Invocation for SIM,” “Declaring a SIM Database,” and “DISCARD Statement” in this section. Related information is also available in the description of the CURRENT function under “Selection Expressions” in this section. Refer to the *InfoExec SIM Programming Guide* for information about query concepts, use of update and retrieval queries, and use of the CURRENT function with queries.

Example

The following example declares several queries. The first two queries specify class identifiers. The class STUDENT will be used in UPDATE_STU_QUERY and the class INSTRUCTOR will be used in INSTRUCTOR_QUERY. The next queries specify DMRECORD identifiers. The previously defined DMRECORD STU_REC will be used in STUQ and the previously defined DMRECORD COURSE_REC will be used in COURSEQ. And, the previously defined format INQ_TYPE will be used in INQUIRYQ.

```
TYPE DMRECORD INQ_TYPE
     (INTEGER SOC_SEC_NO;
      EBCDIC ARRAY NAME [0:29]);
QUERY UPDATE_STU_QUERY (STUDENT),     % STUDENT is a class ID
      INSTRUCTOR_QUERY (INSTRUCTOR),  % INSTRUCTOR is a class ID
      STUQ (STU_REC),                 % STU_REC is a DMRECORD ID
      COURSEQ (COURSE_REC),           % COURSE_REC is a DMRECORD ID
      INQUIRYQ (INQ_TYPE);            % INQ_TYPE is a DMRECORD TYPE
```

Declaring DMRECORDS

<DMRECORD declaration>

```
──┬──────────────────┬─ DMRECORD ─────────────────────────────────────→
  └─ <packing spec> ─┘

 ┌←─────────────────── , ───────────────┐
→┴─ <DMRECORD ID> ─┬─────────────────┬──┴───────────────────────────────┤
                   └─ <field list> ─┘
```

<packing spec>

```
── UNPACKED ────────────────────────────────────────────────────────────┤
```

<field list>

```
      ┌←──────────────────────────── ; ────────────────────────┐
– ( ──┴─┬──┬─ REAL ─────┬─ <field ID> ─────────────────────┬───┴─ ) ──────────┤
        │  ├─ BOOLEAN ──┤                                  │
        │  ├─ DOUBLE ───┤                                  │
        │  └─ INTEGER ──┘                                  │
        ├─ <entity reference declaration> ─────────────────┤
        │            ┌←──────────── , ──────────────┐      │
        ├─ RECORD ───┴─ <field ID> – <field list> ─┴──────┤
        │                  ┌←──────────── , ──────────────┐│
        └─ EBCDIC ARRAY ───┴─ <field ID> – <bound pair> ──┴┘
```

<DMRECORD ID>

```
– <identifier> ─────────────────────────────────────────────────────────┤
```

<field ID>

```
  ┌←──────── , ───┐
──┴─ <identifier> ─┴────────────────────────────────────────────────────┤
```

Explanation

A DMRECORD consists of fields which are used to hold information retrieved from SIM. References to a field must be fully qualified. The type of the DMRECORD variable must be compatible with the data to be retrieved.

The DMRECORD hold hidden control information provided by SIM to show which fields are null and have no current value. The EXISITS function is provided to determine whether or not a field is marked as being null. The RETRIEVE statement is used to write data into a DMRECORD variable. All other uses of DMRECORD variables are read-only. The compiler does not provide any protection to prevent the user from accessing a DMRECORD variable before a RETRIEVE has been executed.

A DMRECORD can be bound to other DMRECORDs. Refer to “Binding Considerations” in this section for more information.

Use the DMRECORD declaration to declare a DMRECORD variable. Use the DMRECORD type declaration to declare a DMRECORD record structure description. The DMRECORD type declaration must be used whenever a DMRECORD is passed as a parameter.

A packing spec construct specifies the record packing. The default is UNPACKED. Unpacked records begin each field on a word boundary, regardless of where the previous field ends.

The DMRECORD ID is the name within the program of the variable being declared.

The field list contains the type and field ID of the fields that comprise the DMRECORD. The fields can be of type Real, Boolean, Double, Integer, Entity Reference, Record, or EBCDIC array. All types other than Entity Reference and Record are described in Volume 1. Also consult Volume 1 for a complete explanation of bound pairs in an array.

Fields of type Record enable nested structured data and are used to hold compound attributes. A compound attribute has several parts. For example, a name might be a compound attribute with the first, middle, and last names comprising the parts. As a result, Record fields are broken down into subfields, each one associated with one part of the compound attribute. Note that although Record fields can be nested, a DMRECORD itself cannot be nested in another DMRECORD.

Record type fields do not provide generalized records in ALGOL. They can be declared only as fields within DMRECORDs, and are subject to the same restrictions as the other field types.

The field ID construct is the name of the field. A field and a variable can share the same name, since context can be used to determine which one is being referenced.

Additional information relating to the fields of a DMRECORD is included under “Referencing DMRECORD Fields” in this section. Information on the class ID construct is included under “Declaring a Database” in this section. Information on the entity reference declaration construct is included under “Declaring an Entity Reference Variable Data Type” in this section.

Additional information relating to DMRECORDS is included under “Declaring an Entity Reference Variable Data Type,” “Mapping SIM Types into ALGOL,” “TYPE Declaration and Invocation for SIM,” “Binding Considerations for SIM,” “Using DMRECORDS and Their Fields,” “RETRIEVE Statement,” and “Referencing DMRECORD Fields” in this section. Information on the EXISTS function is included under “DM Boolean Functions” in this section.

Related information is also available under “ALGOL Data Types for ADDS” in Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

Example

In the following example, a DMRECORD with the name STU_REC is declared. STU_REC has a nested Record field, a Real field, an EBCDIC array field, and an Entity Reference field.

```
DMRECORD STU_REC
     (RECORD STU_NAME (EBCDIC ARRAY FIRST [0:50],
                                    LAST  [0:50],
                                    MIDDLE [0:50]);
      REAL TITLE_CODE;
      EBCDIC ARRAY MINOR [0:10];
      ENTITY REFERENCE COURSE_TAKING (COURSE));
```

Type Declaration and Invocation for SIM

<DMRECORD type declaration>

```
── TYPE ─┬─────────────────┬─ DMRECORD ──────────────────────────→
         └─ <packing spec> ─┘

   ┌←──────────────────── , ──────────────┐
→──┴─ <DMRECORD type ID> ─┬───────────────┴─────────────────────┤
                          └─ <field list> ─┘
```

<DMRECORD type invocation>

```
                        ┌←─────────── , ───────┐
─ <DMRECORD type ID> ───┴─ <DMRECORD ID> ─────┴──────────────────────┤
```

<DMRECORD type ID>

```
─ <identifier> ──────────────────────────────────────────────────────┤
```

Explanation

A DMRECORD is a structured data type, consisting of fields, which is used to hold information retrieved from SIM. The structure is described in a TYPE declaration. The type of the DMRECORD variable must be compatible with the data to be retrieved.

The TYPE declaration is used to associate a user-defined name with a user-defined format. The format can then be used as a data description. Normally, a declaration creates a structure as a variable. In contrast, the TYPE declaration does not create a variable; it simply defines a type identifier that can be used to declare record variables. A type identifier is associated with the DMRECORD declaration. In effect, the type identifier is the name of a record structure description.

Only variables that share the same entity description and type are compatible. The TYPE declaration provides compatibility for the DMRECORDs. Records described by separate, distinct entities and identical in content are not compatible if they do not share the same type identifier.

A TYPE declaration must precede a type invocation. The type invocation declares records that have the structure associated with the type identifier.

Additional information relating to SIM type declarations is included under "Declaring DMRECORDS," "Referencing DMRECORD Fields," and "Binding Considerations for SIM" in this section. Related information is also available under "ALGOL Data Types for ADDS " in Section 2, "Using Advanced Data Dictionary System (ADDS) Extensions."

The DMRECORD type ID construct is the user-defined name associated with the format. In the type invocation, each DMRECORD specified by a DMRECORD type identifier has the structure defined by the type identifier in the TYPE declaration. In the DMRECORD type declaration syntax, the DMRECORD type ID is the name of a DMRECORD structure description.

The field ID construct is the name of the field. The names of the fields in a TYPE declaration must be unique across that specification. However, field names need not be unique across different TYPE declarations. A field and a variable can share the same name, since context can be used to determine which one is being referenced.

Additional information relating to the DMRECORD declaration, DMRECORD ID, packing spec, and field list constructs is included under “Declaring DMRECORDS” in this section. Related information is also available under “Referencing DMRECORD Fields” in this section.

Examples

In this example, a TYPE declaration defines the DMRECORD INSTR_REC_TYPE as three fields. The first two fields, EMPLOYEE_NO and HDATE, are type Integer. The third field is an EBCDIC array whose field ID is NAME. There is no record until the type is invoked and referenced by the DMRECORD identifier.

```
TYPE DMRECORD INSTR_REC_TYPE
     (INTEGER EMPLOYEE_NO, HDATE;
      EBCDIC ARRAY NAME [0:10]);
INSTR_REC_TYPE INSTR_REC;
```

In the following example, the TYPE declaration defines the DMRECORD STU_REC_TYPE as having a nested Record field, a Real field, an EBCDIC array field, and an Entity Reference field. The variable STUDENT_RECORD will have the format described by STU_REC_TYPE. There is no record until the type is invoked and referenced by the DMRECORD identifier.

```
TYPE DMRECORD STU_REC_TYPE
     (RECORD STU_NAME (EBCDIC ARRAY FIRST [0:50],
                                    LAST  [0:50],
                                    MIDDLE [0:50]);
      REAL TITLE_CODE;
      EBCDIC ARRAY MINOR [0:10];
      ENTITY REFERENCE COURSE_TAKING (COURSE));
STU_REC_TYPE STU_RECORD;
```

In the following example, the DMRECORD COURSE_REC_TYPE is defined as having three EBCDIC array fields. The variable COURSE_RECORD will have the format described by COURSE_REC_TYPE. There is no record until the type is invoked and referenced by the DMRECORD identifier.

```
TYPE DMRECORD COURSE_REC_TYPE
     (EBCDIC ARRAY COURSE_TITLE [0:100],
                   COURSE_MAJOR [0:50],
                   PROFESSOR [0:20]);
COURSE_REC_TYPE COURSE_RECORD;
```

Referencing DMRECORD Fields

<DM field reference>

```
                   ┌←──────────────────────┐
── <DMRECORD ID> ──┴── . ── <field ID> ────┴─┬───────────────────────────┬──────────────┤
                                             ├─ <subscript> ─────────────┤
                                             └─ <partial word part> ─────┘
```

References to a field in a DMRECORD must be fully qualified; all nested field names must be specified.

Note that DMRECORD variables are basically read-only. Only the RETRIEVE statement can write to a DMRECORD variable. Therefore, references to the fields are restricted to read-only instances.

Additional information relating to the DMRECORD ID and field ID constructs is included under “Declaring DMRECORDS” in this section.

Additional information relating to DMRECORD fields is included under “RETRIEVE Statement” and “Type Declaration and Invocation for SIM” in this section.

Explanation

The DMRECORD ID construct is the variable for the previously specified format and type.

The field ID construct is the name of a field in the previously specified format and type. Field names need not be unique across different TYPE DMRECORD declarations. A field and a variable can share the same name if the context can be used to determine which one is being referenced. Each reference to a field must, however, be fully specified.

A subscript specification is only permitted for fields of type EBCDIC array.

The partial word part syntax is permitted only for fields of type Real, Integer, and Boolean.

Example

This example references the field PROFESSOR in the DMRECORD variable COURSE_RECORD.

```
IF PROF_REC.EMPLOYEE_NO = 12 THEN ...
WHILE COURSE_RECORD.PROFESSOR = "PROFA" DO ...
```

Using DMRECORDS and Their Fields

Fields in a DMRECORD can be individually examined. They can be individually validated through the EXISTS function. However, the fields can be altered only by SIM. The value in a field can be assigned to variables of compatible types.

Fields can be passed as parameters to procedures. When an individual field is passed, information about whether the field is null is not passed with the field.

Passing Fields of Type Real, Boolean, Double, and Integer

Fields of type Real, Boolean, Double, and Integer can be used as actual pass-by-value parameters to a formal parameter of the appropriate type.

Passing Fields of Type Entity Reference

Fields of type Entity Reference cannot be passed directly; they can be assigned to a regular Entity Reference variable which can then be passed.

Passing Fields of Type Record

Fields of type Record cannot be passed directly; however, the fields of the Record field can be passed individually. For example, if a record contains three EBCDIC fields, each can be passed separately.

Passing Fields of Type EBCDIC Array

Fields of type EBCDIC array can be passed to a formal parameter that is declared as an “*”-bounded EBCDIC array. An attempt by the procedure to write into an actual parameter that is an EBCDIC array field in a DMRECORD results in a run-time error.

Passing an Entire DMRECORD Variable

If the formal parameter is declared and invoked through a DMRECORD type declaration and invocation as having exactly the same format and type as the actual parameter, then the entire DMRECORD variable can be passed as a parameter.

The following example shows both the correct and incorrect usage of a DMRECORD as a parameter.

```
BEGIN
TYPE STU_REC_TYPE DMRECORD (REAL STU_NUM;
   RECORD STU_NAME (EBCDIC ARRAY LAST [0:20];
                    REAL TITLE_CODE));
TYPE INSTR_REC_TYPE DMRECORD (REAL INSTR_NUM);

STU_REC_TYPE STUDENT_RECORD;
INSTR_REC_TYPE INSTR;

PROCEDURE P (X);
   STU_REC_TYPE X;
   BEGIN
   REAL A;
   A := X.STU_NAME.TITLE_CODE;
   END;  % OF PROCEDURE P

P (STUDENT_RECORD);  % LEGAL BECAUSE ACTUAL AND FORMAL ARE
                     % EXACTLY THE SAME TYPE.

P (INSTR);           % ERROR BECAUSE ACTUAL AND FORMAL ARE
                     % NOT EXACTLY THE SAME TYPE.
          .
          .
          .
END.
```

Note that two DMRECORD formats, even if they have exactly the same layout, are not the same; they are considered to be two different types.

Assigning Pointers

Pointers can be assigned to a DMRECORD variable and to a field within a DMRECORD. Any attempt to replace into a DMRECORD variable through a pointer results in a run-time error.

Output of Real, Boolean, Double, Integer, and EBCDIC Array Fields

Formatted and regular output of type Real, Boolean, Double, Integer, and EBCDIC array fields is supported exactly as formatted I/O for variables of these types.

Output of Entity Reference and Record Fields

Output of fields of type Entity Reference is not supported in any form because Entity References are basically pointers into a SIM database and they are only valid while the program is in transaction state.

Record fields cannot be elements in a write statement. Regular and formatted output of Record fields is not permitted because they can contain an Entity Reference field.

Output of DMRECORD Variables

Regular and formatted output of DMRECORD variables is not supported.

Binding Considerations for SIM

A DMRECORD variable can be bound to another DMRECORD variable or to an "*"-bound EBCDIC array. A DMRECORD can also be bound to any other record type that can be bound to an "*"-bound EBCDIC array. The Binder program does not check the record structures for compatibility; therefore, it binds DMRECORD variables to similarly defined DMRECORDs.

Procedures that have DMRECORD formal parameters can also be bound, but type checking will not be performed at bind time. The user must ensure that the types of the formal and actual parameters are identical.

Refer to the *Binder Programming Reference Manual* for more information.

Impact of Declaring a Variable in a Subprogram

How the variable is declared in a subprogram determines what the subprogram can do with the variable and whether the variable is properly protected against write access.

- If the subprogram declares the variable as a DMRECORD variable, the DMRECORD variable can be accessed through the described fields. Functions such as EXISTS can be used.
- If the subprogram declares the variable as another type of record variable, the variable can be accessed through the field names of the record. The semantic rules for that type of record variable are enforced.
- If the subprogram declares the variable as an EBCDIC array, no field-oriented access can be used. Assignment to the variable is permitted.

Impact of Packing

The type of packing being used is an important consideration when more than one language is being bound together. The default packing type is not the same for every language. For example, the ALGOL default begins each field on a word boundary while COBOL starts a field immediately after the previous field. It might be necessary to declare filler fields in the COBOL description of a DMRECORD in order to have it match an ALGOL DMRECORD correctly.

The type of packing for a DMRECORD is specified in the TYPE DMRECORD declaration. The default packing is UNPACKED.

Declaring an Entity Reference Variable Data Type

<entity reference declaration>

```
                    ┌←─────────────────── , ──────────────────┐
── ENTITY REFERENCE ─┴─ <entity ref ID> ── ( ── <class ID> ── ) ─┴──────┤
```

<entity reference array declaration>

```
── ENTITY REFERENCE ARRAY ─────────────────────────────────────────────→

 ┌←──────────────────────────── , ──────────────────────────────┐
→┴─ <ent ref array ID> ── ( ── <class ID> ── ) ── [ ── <b.p. list> ── ] ─┴──┤
```

<entity ref ID>

```
─ <identifier> ────────────────────────────────────────────────────┤
```

<entity ref array ID>

```
─ <identifier> ────────────────────────────────────────────────────┤
```

Explanation

An Entity Reference variable is used to contain an explicit reference to a SIM database class entity. The variable can be an array. The SIM database containing the class must be declared prior to the ENTITY REFERENCE declaration.

An Entity Reference variable can be used to compare and assign entity-valued attribute (EVA) values without having to select and retrieve the entities involved. For example, you might need to know if the advisor of two selected students is the same entity. You can retrieve the entity reference value for each student's advisor and compare them. The entity reference values can also be assigned to EVAs.

In general, extended attributes are qualified by EVAs. An extended attribute can be immediate to a class which is connected to a perspective class through intermediary classes. In these cases, chains of EVAs are used in the qualification.

Each Entity Reference variable is associated with one specific class. Entity Reference variables can only be assigned and compared with Entity Reference variables and EVAs associated with the same class. This data type cannot be compared with arithmetic or string variables or used in arithmetic or string expressions.

Entity reference values are only valid in the transaction state in which they are retrieved. Using an entity reference value outside of transaction state, or in a different transaction state, will result in a run-time error. Entity References can be passed as by-name parameters provided the program remains in transaction state.

Additional information relating to entity reference variables is included under “Selection Expressions” in this section, particularly in the description of the CURRENT function.

The constructs entity ref ID and ent ref array ID identify the variable.

The class ID construct specifies the class associated with the Entity Reference variable.

Additional information relating to the class ID construct is included under “Declaring a SIM Database” in this section.

The b.p. list construct designates the bound pair list. The subscript bounds for an array are given in the first bound pair list following the array identifier. Refer to Volume 1 for a complete explanation of bound pair lists in an array declaration.

Examples

In the first example, the Entity Reference variable ADVISOR1 references the class INSTRUCTOR. STUDENT_TRANSCRIPT references the class TRANSCRIPT.

```
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR),
                 STUDENT_TRANSCRIPT (TRANSCRIPT);
```

In the second example the Entity Reference variable ADVISORS is an array. Its class, with a bound pair list, is INSTRUCTOR.

```
ENTITY REFERENCE ARRAY ADVISORS (INSTRUCTOR) [0:9];
```

Using Data Management Functions and Expressions

All data management (DM) functions are forwarded to SIM for complete evaluation. The arguments of a function can contain references to unretrieved values in the SIM database. At run time, ALGOL expressions are evaluated to single values and passed to SIM by value. ALGOL operators, precedence rules, and type compatibility are expected in all ALGOL expressions.

The DM functions are

- Arithmetic functions
- String functions
- Symbolic functions
- Boolean functions

DM expressions are ALGOL expressions in which the following primaries are permitted:

- DM functions
- Qualification identification
- Class identification
- Selection expression
- Inverse entity-valued attributes

A DM primary is not permitted in a pointer expression, complex expression, ALGOL function, or array subscript. Consult Volume 1 for more information on primaries.

The selection expression is used to determine which database entities are available for retrieval, deletion, or modification. Both global and local selection expressions are supported. All selection expressions are evaluated according to ALGOL rules and then sent to SIM.

The ALGOL formats for using SIM functions and the selection expression are covered in this section. Refer to Volume 1 for a comprehensive explanation of

- Arithmetic expressions and operators
- Boolean expressions and operators
- String expressions
- Relational operators

The *InfoExec SIM Programming Guide* discusses these concepts as they relate to SIM.

DM Arithmetic Functions

<DM arithmetic functions>

```
─┬─ DMCOUNT  ─ ( ─┬─ <qual ID> ──┬─ ) ──────────────────────────────┬──────┤
 │                └─ <class ID> ─┘                                   │
 │                                                                   │
 ├──┬─ DMAVG  ─┬─ ( ─ <qual ID> ─ ) ─────────────────────────────────┤
 │  ├─ DMSUM  ─┤                                                     │
 │  ├─ DMMIN  ─┤                                                     │
 │  └─ DMMAX  ─┘                                                     │
 │                                                                   │
 ├──┬─ DMROUND  ─┬─ ( ─ <arithmetic expression> ─ ) ─────────────────┤
 │  ├─ DMTRUNC  ─┤                                                   │
 │  ├─ DMABS  ───┤                                                   │
 │  └─ DMSQRT  ──┘                                                   │
 │                                                                   │
 ├─ DMPOS  ─ ( ─ <str exp> ─ , ─ <str exp> ─┬─────────────────┬─ ) ─┤
 │                                          └─ , ─ <integer> ─┘      │
 │                                                                   │
 ├─ SUBROLE  ─ ( ─ <class ID> ─ ) ───────────────────────────────────┤
 │                                                                   │
 └─ DMLENGTH  ─ ( ─ <string expression> ─ ) ─────────────────────────┘
```

Explanation

The DM arithmetic functions return arithmetic values.

The DMCOUNT function accepts a class, a data-valued attribute (DVA), or an entity-valued attribute (EVA). All other functions accept only a data-valued attribute.

The qual ID construct qualifies a DVA or EVA to the environment to which the function is attached.

Arithmetic expressions are expressions that return numerical values. String expressions using the SIM interface return EBCDIC strings and must be constant string expressions. Their length must be able to be determined at compile time. Constant string expressions and primaries include string constants, SIM attributes of type string or symbolic, EBCDIC fields of DMRECORDS, and Advanced Data Dictionary System (ADDS) structures. String variables and string arrays cannot be used in SIM expressions.

Additional information relating to the class ID construct is included under "Declaring a SIM Database" in this section. Information on the qual ID construct is included under "Selection Expressions" in this section. Refer to Volume 1 of this manual and to the *InfoExec SIM Programming Guide* for a complete discussion of arithmetic and string expressions.

Table 7–2 below gives the function keyword and the corresponding value returned.

Table 7–2. DM Function Keywords and Values Returned

Keyword	Value Returned
DMABS	Absolute Real value of specified arithmetic expression.
DMAVG	Average or mean of a collection of numeric values.
DMCOUNT	Number of entities in a class or the number of values multivalued attribute. Can also be used on single-valued attributes.
DMLENGTH	Length of string expression.
DMMAX	Maximum value from a collection of values.
DMMIN	Minimum value from a collection of values.
DMPOS	Returns the starting position of the specified occurrence of a designated string within a string. The string to be searched is given first, followed by the string to search for.
DMROUND	Arithmetic expression rounded to nearest integer.
DMSQRT	Nonnegative real number that is square root of arithmetic expression.
DMSUM	Sum of all numeric values in a collection.
DMTRUNC	Integer portion of truncated arithmetic expression.
SUBROLE	Used for testing the value of subrole data-valued attributes.

Examples

In the following example, the value of the INPUT_AGE is assigned as the value of CHILDREN_AGE and truncated. Therefore, if INPUT_AGE is 10 years and 2 months (10 and 1/6) or 10 years and 10 months (10 and 5/6), the value of CHILDREN_AGE is 10. (The fraction is not rounded to the nearest value.)

```
INSERT PERSON
    (ASSIGN (CHILDREN_AGE,DTRUNC(INPUT_AGE));
```

In this example, the selection of the minimum value of STUDENT_AGE is qualified by a Social Security number criteria. Once the value is selected, it is assigned to SPOUSE_AGE.

```
MODIFY PERSON
   (ASSIGN (SPOUSE_AGE,DMMIN(STUDENT_AGE))
    WHERE SOC_SEC_NO = INPUT_SOCIAL;
```

DM String Functions

<DM string functions>

```
─┬─ DMEXT  ─ ( ─ <str exp> ─ , ─ <integer> ─ , ─┬─ <integer> ─┬─ ) ─┬──────┤
 │                                               └─ * ──────────┘     │
 │                                                                    │
 ├─ DMRPT  ─ ( ─ <str exp> ─ , ─ <integer> ─ ) ───────────────────────┤
 │                                                                    │
 │             ┌←──────────── , ──────────┐                           │
 └─ DMCHR  ─ ( ─┴─ <hex string literal> ─┴─ ) ────────────────────────┘
```

Explanation

DM string functions can take one or more strings as an argument, or produce a string as the function value, or perform both operations.

String expressions using the SIM interface return EBCDIC strings and must be constant string expressions. Their length must be able to be determined at compile time. Constant string expressions and primaries include string constants, SIM attributes of type string or symbolic, EBCDIC fields of DMRECORDS, and Advanced Data Dictionary System (ADDS) structures. String variables and string arrays cannot be used in SIM expressions.

Refer to Volume 1 of this manual for a discussion of string expressions (shown here as str exp), integers, and hexadecimal string literals.

DMEXT returns the substring of the string expression with the specified beginning and ending positions. (The first integer is the beginning position, the second integer is the ending position.) An asterisk (*) as the ending position indicates the end of the string.

DMRPT returns the specified string a designated number of times.

DMCHR constructs a string that is a concatenation of the EBCDIC characters represented by the hexadecimal numbers used in the argument.

Examples

In the following example, a substring of NAME, beginning in position 16 and going to the end of the string, is returned in the variable MIDDLE_NAME.

```
SELECT STUQ FROM STUDENT
    (MIDDLE_NAME = DMEXT(NAME,16,*));
```

The following example assigns the string NAME to ID_CODE. The string will appear twice in ID_CODE so that if the string was “MIDDLENAME”, ID_CODE would be assigned “MIDDLENAMEMIDDLENAME”.

```
SELECT INSTRQ FROM INSTRUCTOR
    (ID_CODE = DMRPT(NAME,2))
     WHERE SOC_SEC_NO < A;
```

Using the perspective class DEPARTMENT, the query DEPTQ selects the name of the department and the department number if the instructor's name is equivalent to the hexadecimal string of characters “ABCD”.

```
SELECT DEPTQ FROM DEPARTMENT
    (NAME_OF_DEPT; DEPT.NO)
    WHERE INSTRUCTOR_EMPLOYED.NAME = DMCHR(4"C1C2C3C4");
```

DM Symbolic Functions

<DM symbolic functions>

```
──┬─ DMPRED ─┬─ ( ─ <attribute chain> ─ ) ──────────────────────────────┤
  └─ DMSUCC ─┘
```

Explanation

DM symbolic functions operate on SIM symbolic types. In ALGOL, a data type of symbolic is supported as a string (EBCDIC array).

The attribute chain construct must have as its final element an attribute that has a type of symbolic.

DMPRED returns the previous symbolic value.

DMSUCC returns the value of the successive symbolic.

Consult Volume 1 of this manual for a detailed explanation of identifiers. Refer to the *InfoExec SIM Programming Guide* for detailed information about symbolic types and functions.

Additional information is included under “Mapping SIM Types into ALGOL” in this section.

Additional information relating to the attribute chain construct is included under “DM Primaries” in this section.

Example

In the following example, the query PERSONQ uses the perspective class PERSON to select the value of NAME if the preceding and successive symbol values are “SINGLE”.

```
SELECT PERSONQ FROM PERSON (NAME)
    WHERE DMPRED(MSTATUS) = "SINGLE" AND
    DMSUCC(MSTATUS) = "SINGLE";
```

DM Boolean Functions

<DM Boolean functions>

```
─┬─ DMEXISTS  ─ ( ─┬─ <MVA qual ID> ─┬─ ) ──────────────────────────────┬──┤
 │                 └─ <SVA qual ID> ─┘                                   │
 ├─ EXISTS  ─ ( ─ <DMRECORD field ID> ─ ) ──────────────────────────────┤
 ├─ DMMATCH  ─ ( ─ <DM string exp> ─ , ─ <pattern> ─ ) ─────────────────┤
 ├─ DMISA  ─ ( ─┬─ <EVA qual ID> ─┬─ , ─ <class ID> ─ ) ────────────────┤
 │              └─ <class ID> ────┘                                      │
 └─ DMEQUIV  ─ ( ─ <DM string exp> ─ , ─ <rel op> ─ , ─ <DM string exp> ─ ) ─┘
```

Explanation

DM Boolean functions include relational and Boolean operators.

The DMEXISTS and the EXISTS functions are used to determine if an entity exists. EXISTS also determines whether or not a field is marked as being null. The functions are TRUE when the operand has a value other than null. The operand for DMEXISTS is either a qualified multivalued attribute (MVA) or a qualified single-valued attribute (SVA). The operand for EXISTS is a field within a DMRECORD.

The DMMATCH function tests whether the DM string expression matches a specified pattern. It is comparable to the SIM "ISIN" operator. A <pattern> construct is specified using literal characters plus metacharacters. For more information, refer to the *InfoExec SIM Programming Guide.*

A DM string exp construct consists of valid combinations of DM string functions and constant string expressions. Constant string expressions include EBCDIC string constants, SIM attributes of type string or symbolic, EBCDIC fields of DMRECORDS, EBCDIC fields of Advanced Data Dictionary System (ADDS) structures, and a limited form of the string function. The limited string function can have only a constant arithmetic expression as its second argument. String variables and string arrays cannot be used in SIM expressions. Consult Volume 1 of this manual for a detailed explanation of string expressions. Additional information relating to DM string functions is included under "DM String Functions" in this section.

Additional information relating to the DMRECORD ID and field ID constructs is included under "Declaring DMRECORDS" in this section. Information on the qual ID construct is included under "Selection Expressions" in this section. Information on the class ID construct is included under "Declaring a SIM Database."

The DMISA function tests whether an entity plays a certain role in a class. The function is TRUE if the qualified entity-valued attribute (EVA) or the first designated class is a member of the second designated class.

The DMEQUIV function compares two DM string expressions, using the values of the characters in a pre-defined ordering sequence. This is called an “equivalent” string comparison.

Refer to the *InfoExec SIM Programming Guide* for more detailed information about these DM Boolean expressions functions.

Example

In the following example, the query PERSONQ uses the perspective class PERSON to select values for NAME and SOC_SEC_NO. Values are selected if the DMISA function returns as TRUE.

The next query, STUQ, uses the perspective of the class STUDENT. The values for NAME and STUDENT_NUM will be returned if DMEXISTS tests as true.

The EXISTS function then determines if the field TRANSCRIPT_RECORD within the DMRECORD STUDENT_RECORD exists. If it does, the query STUQ uses the perspective class STUDENT to select the SOC_SEC_NO where the DM string expression NAME matches the pattern “JULES VERN”.

```
SELECT PERSONQ FROM PERSON
   (NAME;SOC_SEC_NO)
    WHERE DMISA(SPOUSE,STUDENT);
SELECT STUQ FROM STUDENT
   (NAME;STUDENT_NUM=STUDENT_NO)
    WHERE DMEXISTS(STUDENT_NO);
IF EXISTS(STUDENT_RECORD.TRANSCRIPT_RECORD) THEN
    SELECT STUQ FROM STUDENT
   (SOC_SEC_NO)
    WHERE DMMATCH(NAME,"JULES VERN");
```

DM Primaries

<DM primaries>

```
──┬─ <DM function> ────────────────────────────────┬──────────────────────┤
  ├─ <attribute chain> ────────────────────────────┤
  ├─ <qual ID> ────────────────────────────────────┤
  ├─ <class ID> ───────────────────────────────────┤
  ├─ <local selection expression> ─────────────────┤
  └─ INVERSE  ─ ( ─ <entity-valued attribute> ─ ) ─┘
```

<attribute chain>

```
──┬─────────────────┬─ <attribute ID> ──────────────────────────────────┤
  └─ <qual ID> ─ . ─┘
```

Explanation

A specified entity (SIM database or class) can be a primary.

The local selection expression, which includes several other primary elements, can be a primary.

The INVERSE function uses an inverse attribute for the specified entity-valued attribute (EVA). The result of the function can then be used as a primary.

The following primaries can be used to form a DM expression. Consult the section "Expressions" in Volume 1 for more information on primaries.

Additional information relating to the attribute ID, local selection expression, and qual ID constructs is included under "Selection Expressions" in this section. Information on the entity reference ID construct is included under "Declaring an Entity Reference Variable Data Type" in this section. Information on the DM function construct is included under "DM Arithmetic Functions," "DM String Functions," "DM Boolean Functions," and "DM Symbolic Functions" in this section.

Example

A local selection expression is shown in the following example. It is used as part of the INCLUDE syntax to further narrow the scope of instructors from INSTRUCTOR_EMPLOYED. The local selection expression uses the class INSTRUCTOR as the class ID. Only those INSTRUCTOR_EMPLOYED who have an EMP_NO of 1 or 2 are included.

```
MODIFY DEPARTMENT
   (INCLUDE (INSTRUCTOR_EMPLOYED,
             [INSTRUCTOR WHERE EMP_NO = 1 OR EMP_NO = 2]))
    WHERE DEPT_NO = INPUT_DEPTNO;
```

Selection Expressions

<selection expression>

```
— <DM Boolean expression> ————————————————————————————————————————————┤
```

<DM Boolean expression>

```
 ┌←——— <Boolean operator> ——┐
—┴┬— <DM Boolean primary> —┬┴——————————————————————————————————————————┤
  └— <Boolean primary> ————┘
```

<DM Boolean primary>

```
—┬— ( — <selection expression> — ) —┬——————————————————————————————————┤
 ├— <entity-valued relation> ———————┤
 └— <DM Boolean function> ——————————┘
```

<entity-valued relation>

```
—┬—┬— <local selection expression> ————————┬┬— EQL  —┬— <EVA qual ID> —┬┤
 │ ├— INVERSE  — ( — <EVA> — ) ————————————┤├— = ————┤                 │
 │ ├— CURRENT  — ( — <query ID> — ) ———————┤├— NEQ  —┤                 │
 │ ├— <entity reference ID> ———————————————┤└— ^= ———┘                 │
 │ └— <class ID> — ( — <entity ref ID> — ) ┘                           │
 └——┬— CURRENT  — ( — <query ID> — ) ———————┬┬— EQL  —┬— <class ID> ————┘
    ├— <entity reference ID> ———————————————┤├— = ————┤
    └— <class ID> — ( — <entity ref ID> — ) ┘├— NEQ  —┤
                                             └— ^= ———┘
```

<qual ID>

```
—┬————————————————————┬— <qual term> ——————————————————————————————————┤
 └— <class ID> — . ———┘
```

<qual term>

```
—┬— <attribute ID> —┬————————————————————————┬┬—————————————————————————┤
 │                  └— <compound selector> ——┘│
 ├— <quantifier> — ( — <qual term> — ) ———————┤
 └— <path expression> — . — <qual term> ——————┘
```

<attribute ID>

```
– <identifier> ———————————————————————————————————————————————————————————┤
```

<path expression>

```
—┬— <entity-valued attribute chain> ——————————————————┬———————————————————┤
 ├— INVERSE  – ( – <entity-valued attribute chain> – ) —┤
 ├— <quantifier> – ( – <entity valued qual term> – ) ——┤
 ├— <transitive expression> ———————————————————————————┤
 ├— <local selection expression> ——————————————————————┤
 ├— <class ID> – ( – <entity-valued qual term> – ) ————┤
 └— <Called ref ID> ———————————————————————————————————┘
```

<transitive expression>

```
– TRANSITIVE ( <trans arg> ——┬————————————————————————————————┬— ) ———————┤
                             └— END LEVEL <integer constant> —┘
```

<transitive argument>

```
– <reflexive path expression> ————————————————————————————————————————————┤
```

<quantifier>

```
—┬— ALL  —┬———————————————————————————————————————————————————————————————┤
 ├— SOME —┤
 └— NONE —┘
```

<local selection expression>

```
– [ ——┬— <class ID> ——————————————————————┬— WHERE – <selection expression> ]—┤
      ├— <entity-valued attribute chain> —┤
      └— INVERSE  – ( – <EVA ID> – ) —————┘
```

<Called ref ID>

```
– <identifier> ———————————————————————————————————————————————————————————┤
```

Explanation

A selection expression is used to determine which entities from the SIM database are eligible for retrieval, deletion, or modification. It is used to identify the set of entities upon which a query is to operate. It narrows the group of entities in the perspective class for the scope of the query.

If an entity meets the stated conditions of the selection expression, the query uses the entity once the entity is retrieved.

A global selection expression applies to the whole query. A local selection expression applies to only a specific attribute.

The selection expression is a Boolean expression in which DM primaries and functions are permitted. Both arithmetic and string expressions can be used in a selection expression. Any part of the selection expression that is strictly ALGOL is evaluated according to ALGOL rules. The value is then sent to SIM.

In addition, the following can be used in a selection expression:

- Standard arithmetic operators
- Relational operators
- Boolean operators
- Order functions
- Aggregate functions
- Arithmetic functions
- Boolean functions
- Primaries
- String functions
- Symbolic functions

Relational operators test for relationships between values. They produce values of TRUE, FALSE, or NULL. Boolean operators also produce values of TRUE, FALSE or NULL. A Boolean null signifies SIM cannot determine if a Boolean expression is TRUE or FALSE.

SIM also provides DM string relational operators to perform “equivalent” string operations. Equivalent string operations compare two strings based on the values of characters in an ordering sequence (instead of the actual binary value of the characters). The available DM string relational operators include the following:

```
EQV_EQL          EQV_LEQ
EQV_GEQ          EQV_LSS
EQV_GTR          EQV_NEQ
```

EQV_LEQ, EQV_LSS, EQV_GEQ, and EQV_GTR are only valid for string operations among ordered types.

An ordering sequence is a predefined arrangement of members in a character set. In an ordering sequence, characters can be placed in order based on criteria other than their binary value. Different characters can be assigned the same ordering sequence value (for example, the characters “A” and “a”).

A collating sequence is a predefined arrangment of members in a character set based on ordering sequence *and* additional priority sequence values. Each character has an ordering sequence value and a priority sequence value. Any characters that have the same ordering sequence value are assigned differing priority sequence values. This gives each character a unique combination of values and determines the character's position in the collating sequence.

Order functions work on attribute values only. Aggregate functions apply to a collection of values and produce one value.

For a detailed explanation of the use of selection expressions with SIM, consult the *InfoExec SIM Programming Guide*.

Additional information relating to selection expressions is included under “Declaring an Entity Reference Variable Data Type,” “DM Arithmetic Functions,” “DM String Functions,” “DM Symbolic Functions,” “DM Boolean Functions,” and “DM Primaries” in this section.

Additional information relating to DM functions is included under “Using Data Management Funcions and Expressions,” “DM Arithmetic Functions,” “DM String Functions,” “DM Symbolic Functions,” and “DM Primaries” in this section.

Consult the *InfoExec SIM Programming Guide* for a discussion of DM Boolean expressions and functions.

The Boolean primary construct within a DM Boolean expression can include either entity-valued relations or relations made up of DM expressions.

A local selection expression affects specific attributes only. It specifies conditions under which values for the attribute are chosen. It corresponds to the SIM “WITH” construct as discussed in the *InfoExec SIM Programming Guide*.

The query ID identifies a previously declared query.

The INVERSE function uses an inverse attribute for the specified entity-valued attribute (EVA).

The CURRENT function should be used inside of transaction state only. It can be used with both update and retrieval queries. In retrieval queries, you can use the CURRENT function to view or compare data without acting on it. In update queries, you can use the CURRENT function to retrieve data and act on it. Consult the *InfoExec SIM Programming Guide* for details of the SIM “CURRENT” function.

The construct entity reference ID identifies the previously declared entity reference variable.

The construct class ID identifies a class in the SIM database.

Entity-valued relations are established using operators. The operators for “equal to” and “not equal to” can be used.

The qual ID construct is used to uniquely identify an entity. The syntax can include the entity's class and SIM database. It always includes qualifying terms.

The qual term and path expression constructs must evaluate to an entity value. The final element must point to a class.

An attribute ID identifies an attribute. The attribute can be single- or multivalued. A compound attribute is an attribute that consists of other attributes. Each of the attributes in a compound attribute are unique. Qualification must be used whenever there is ambiguity. The compound selector is the series of identifiers that uniquely identify the attribute.

For quantifiers, the attribute must be multivalued. The valid quantifiers are ALL, SOME, and NONE. ALL means that each value of the attribute must meet the condition. SOME means that at least one value must meet the condition. NONE means that no value can meet the condition.

The Called ref ID construct is a SIM reference variable. The SIM “CALLED” function is used to assign a variable to a set of entities. The function also can be used in update queries. (Refer to the *InfoExec SIM Programming Guide* for more information. The ALGOL uses can be found in this section's discussion of the SELECT statement.)

Information on the entity reference ID construct is included under “Declaring an Entity Reference Variable Data Type” in this section. Information on the class ID construct is included under “Declaring a SIM Database” in this section. Information on the query ID construct is included under “Declaring a Query Data Type” in this section. Information on the compound spec construct is included under “Database Attribute Assignments” in this section.

The construct transitive expression describes a TRANSITIVE function. A transitive path is used for reflexive attributes. The function returns the transitive closure of a recursive path expression. Refer to the SETTO statements in this section and to the *InfoExec SIM Programming Guide* for further information on the transitive closure facility.

The reflexive path expression is a path expression that originates and ends with the same class. The END LEVEL syntax specifies a level of recursion to be included if a complete closure is not performed.

Additional information relating to selection expressions is included under “SELECT Statement” and “SET TO Statements” in this section.

Examples: Selection Expressions

In the first example, the current query is COURSE_QUERY. If it is a retrieval query, data from the entity COURSE_TAKING can be retrieved and viewed or compared. If it is an update query, you can retrieve and then modify the entity.

```
CURRENT (COURSE_QUERY) = COURSE_TAKING
```

Below, the attribute COURSE_NO is qualified by the entity-valued attribute COURSE_TAKING. From this collection, the lowest number of the upper division courses is compared with UPPER_DIVISION_COURSE.

```
DMMIN (COURSE_TAKING.COURSE_NO) >= UPPER_DIVISION_COURSE
```

The third example of a selection narrows the scope to students that meet two criteria. (The Boolean "AND" means both conditions must be met.)

```
STUDENT = STUDENT_ERV AND S_NUM = 1234
```

Examples: Path Expressions

In the following example, a path consisting of COURSE_TAKEN and COURSE.NO is established.

```
COURSE_TAKEN.COURSE_NO
```

A multiple level path is established by the following example.

```
INSTRUCTOR.STUDENT_ADVISED.TRANSCRIPT_RECORD.GRADE
```

In the next examples, the reflexive attribute PREREQUISITES is a circular path. In the first case, where there is no END LEVEL, it will be applied until all appropriate data is returned. In the second case, the maximum number of times the path expression will be applied is three.

```
TRANSITIVE (PREREQUISITE)
TRANSITIVE (PREREQUISITE END LEVEL 3)
```

SIM Statements

The following SIM statements are supported through the ALGOL interface. Note that all statements are valid only when the SIM database has been declared and opened.

CLOSE	OPEN
DATABASE ATTRIBUTE ASSIGNMENTS	RETRIEVE
DELETE	SELECT
DISCARD	SETTOCHILD
INSERT	SETTOPARENT
MODIFY	

In addition, several SIM transaction statements are supported. These statements enable you to define when the program is in transaction state.

ABORTTRANSACTION	ENDTRANSACTION
BEGINTRANSACTION	SAVETRPOINT
CANCELTRPOINT	

An overview of a transaction, transaction state, transaction points, and COMS's role in transactions are included here.

All of the above SIM statements, their syntax, and examples are explained in this section. The statements are presented in alphabetical order.

Using Transactions

A transaction is an action that causes a change in the SIM database. Transaction state is that period of execution time when the SIM database can be updated. The transaction statements enable SIM to treat two or more query statements as a unit by grouping the statements within a transaction.

A transaction consists of a series of statements begun by a BEGINTRANSACTION statement and concluded by an ENDTRANSACTION statement. SIM assigns transaction points at the beginning and ending statements. These points are used to recover data in case of a failure. Intermediate transaction points can be explicitly created and cancelled using the SAVETRPOINT and CANCELTRPOINT statements.

Transactions are applied but not actually committed until the ENDTRANSACTION statement is executed. If an ABORTTRANSACTION statement is executed before an ENDTRANSACTION statement, none of the accumulated transactions are applied. Instead, the SIM database returns to the state before the BEGINTRANSACTION statement was encountered, before the program entered transaction state. (If the SIM database involved in a transaction is closed before the transaction is ended, the transaction is automatically aborted.) With intermediate transaction points you can control how far to back out and still remain in the transaction state.

To prevent simultaneous transactions from affecting each other, no updates can be done outside of transaction state. Any entities selected inside transaction state are locked; all locks are released at end of transaction. Therefore, while a program can select and retrieve either in or out of transaction state, it can update the SIM database only in transaction state. (Refer to the SELECT statement in this section for more information.)

Entity reference values are only valid within the transaction in which they were retrieved.

A transaction can update only one database. The database is identified by the first update or data retrieval operation. Any attempt to update another database causes an exception to be returned.

In order to provide a recoverable transaction system, SIM and COMS use the ENDTRANSACTION and ABORTTRANSACTION statements. If one or more COMS messages constitute a transaction, the name of the COMS header is used with the ENDTRANSACTION or ABORTTRANSACTION statement and the proper communication is generated by the compiler. If the system fails while the program is in transaction state, COMS resubmits the messages which constituted the transaction when the program is reexecuted.

Note: *At any given time, a program can be in transaction state with only one database. For proper recovery, the name of the database in transaction state should be the name of the database noted in the COMS Utility.*

For more information on programming SIM and COMS together, consult the *InfoExec SIM Programming Guide.*

Additional information relating to COMS extensions is included in Section 3, “Using Communications Management System (COMS) Features.”

ABORTTRANSACTION Statement

<aborttransaction statement>

```
— ABORTTRANSACTION ——┬——————————————————————————┬———————————————————————┤
                     │                          │
                     └— <COMS outputheadername> —┘
```

Explanation

The ABORTTRANSACTION statement cancels all accumulated operations in the current transaction. The program is taken out of the transaction state and the SIM database returns to the point before the BEGINTRANSACTION statement (which initiated the transaction) was executed.

Additional information relating to the ABORTTRANSACTION statement is included under “Declaring Input and Output Headers” in Section 3, “Using Communications Management System (COMS) Features.”

The COMS outputheadername construct identifies the COMS Output Header. If the system fails during transaction state, COMS resubmits the message when the program is reexecuted.

Additional information relating to the outputheadername construct is included under “Declaring Input and Output Headers” in Section 3, “Using Communications Management System (COMS) Features.”

Example

The following example shows an abort when there is no COMS message in the transaction. The second example shows an abort when there is a COMS message in the transaction.

```
ABORTTRANSACTION;
ABORTTRANSACTION MYOUTHEADER;
```

SIM BEGINTRANSACTION Statement

<begintransaction statement>

```
— BEGINTRANSACTION ─┬──────────────┬───────────────────────────────→
                    └─ EXCLUSIVE ──┘

→─┬─────────────────────────────────────────┬─────────────────────┤
  └─ <COMS inputheadername> ─┬──────────────┤
                             └─ <message area> ─┘
```

Explanation

The BEGINTRANSACTION statement places the program in a transaction state.

At any given time, a program can be in a transaction state with only one database.

The EXCLUSIVE option informs SIM that the program is going to perform a long or extensive transaction. SIM assigns the program an exclusive transaction state; that is, a transaction state in which there is no interference from or with other transaction states.

Using the EXCLUSIVE option is one means of preventing deadlocks. However, the option can degrade throughput. Never use the option in an online environment.

Consult the *InfoExec SIM Programming Guide* for more information on the use of the EXCLUSIVE option.

The COMS inputheadername construct identifies the COMS Input Header.

Additional information relating to the inputheadername construct is included under “Declaring Input and Output Headers” in Section 3, “Using Communications Management System (COMS) Features.” Information on the message area construct is included under “RECEIVE Statement” in Section 3, “Using Communications Management System (COMS) Features.”

The message area construct specifies the variable reserved for the actual message. Information on the message area construct is included under “Declaring a Message Area” in Section 3, “Using Communications Management System (COMS) Features.”

CANCELTRPOINT Statement

<canceltrpoint statement>

```
— CANCELTRPOINT ──┬──────────────────────────────┬──────────────────┤
                  └─ ( — <integer expression> — ) ─┘
```

Explanation

The CANCELTRPOINT statement prevents a range of accumulated transactions from being applied. The accumulated operations, from the current point back to either an intermediate transaction point or the beginning of the transaction, are not applied. In all cases, the program is left in transaction state.

The integer expression construct represents a marker set in a SAVETRPOINT statement. If an integer expression is specified, all database changes between the current point and the specified point are not applied. If no integer expression is specified, all SIM database changes from the beginning of the transaction to the current point in the transaction are not applied.

Additional information relating to accumulated transactions and the integer expression construct is included under “SAVETRPOINT Statement” in this section.

Example

In the following example, there is an intermediate transaction point with a marker of “1”. If an error is detected, the CANCELTRPOINT statement will rollback the accumulated transactions to the marker.

```
BEGINTRANSACTION;
  ...
  SAVETRPOINT (1);
  ...
  IF ERROR ... THEN CANCELTRPOINT (1);
ENDTRANSACTION;
```

SIM CLOSE Statement

<close statement>

```
 — CLOSE  — <database name> ————————————————————————————————————————————┤
```

The SIM CLOSE statement closes a previously declared and opened SIM database. If the SIM database involving a transaction is closed before the transaction is ended, the transaction is automatically aborted. Any active query is closed.

The SIM CLOSE statement returns a Boolean result. The statement is used as a usual Boolean expression.

Additional information relating to the CLOSE statement is included under "SIM OPEN Statement" and "Declaring a SIM Database" in this section.

Additional information relating to the database name construct is included under "Declaring a Database" in Section 4, "Using the Data Management System II (DMSII) Interface."

Explanation

The named SIM database name must have been declared and opened.

Example

In the following example, the SIM database UNIVDB is closed.

```
CLOSE UNIVDB;
```

Database Attribute Assignments

<assign spec>

```
– ASSIGN  – ( – <destination> – , ─┬─ <DM expression> ──────────────┬─ ) ─┤
                                   ├─ <local selection expression> ─┤
                                   └─ <compound assign spec> ───────┘
```

<include spec>

```
– INCLUDE  – ( – <destination> – , ─┬─ <DM expression> ──────────────┬─ ) ┤
                                    └─ <local selection expression> ─┘
```

<exclude spec>

```
– EXCLUDE  – ( – <destination> ─┬──────────────────────────┬───────────→
                                └─ <limit specification> ──┘

→─┬─────────────────────────────────────┬─ ) ─────────────────┤
  └─ , – <local selection expression> ──┘
```

<destination>

```
─┬─ <single-statement update destination> ───┬──────────────────────┤
 └─ <multiple-statement update destination> ─┘
```

<compound assign spec>

```
  ┌←─────────────────────────── , ──────────────────────────┐
──┴─ ASSIGN  – ( – <compound spec> – , – <DM expression> – ) ┴──────────┤
```

<single-statement update destination>

```
─┬─ <attribute ID> ─┬────────────────────────────────────────────────┤
 └─ <compound spec> ┘
```

<multiple-statement update destination>

```
– <query ID> – . – <attribute ID> ────────────────────────────────────┤
```

<compound spec>

```
– <record ID> – . – <field ID> ───────────────────────────────────────┤
```

Explanation

The database attribute assignment statements (db attribute assignments) add or remove values from attributes. ASSIGN, INCLUDE, and EXCLUDE are used to assign database attributes.

- As clauses in a single-statement INSERT or MODIFY update
- As statements in a multiple-statement INSERT or MODIFY update

A query variable must be identified and specified for all multiple-statement INSERT and MODIFY updates. Additional information relating to database updates is included under "SIM INSERT Statement" and "SIM MODIFY Statement" in this section.

Database attribute assignments can only update immediate attributes of the current perspective. The assignment of attributes must be done through assign statements appropriate for the data type of the attribute. Attributes of other classes cannot be modified through entity-valued attributes.

ASSIGN updates single-valued attributes (SVAs). Use the DM expression clause for data-valued attributes. For entity-valued attributes, you must use the local selection expression clause. The object of a local selection expression is restricted to class IDs associated with destinations. Omitted parts are assigned null values.

The DM expression construct must evaluate to an appropriate type for the destination specified using normal ALGOL coercion.

The compound assign spec construct must be used when assigning more than one field of a compound. The compound spec construct identifies nested fields of a compound type attribute.

INCLUDE adds values to multivalued attributes (MVAs). Use the DM expression clause for data-valued attributes and the local selection expression clause for entity-valued attributes. The object of a local selection expression is restricted to class IDs associated with destinations. Use ASSIGN to add values to single-valued attributes (SVAs).

EXCLUDE removes values from both MVAs and SVAs. The object of a local selection expression is restricted to destinations for MVAs. The optional local selection expression is not valid when EXCLUDE is used for SVAs.

The limit specification determines the number of values to be excluded. Where NOLIMIT is specified, all values are removed. Where LIMIT is specified, a maximum number of values can be designated. If more values are found, an exception is returned and no values are removed. The default limit is "1".

The query ID construct identifies the destination for the SIM database assignment. It is required with multiple-statement INSERT and MODIFY updates. The specified query must match the query designated in the START and APPLY update statements.

Additional information on the field id construct is included under “Declaring DMRECORDs” in this section. Information on the limit specification construct is included under “SIM DELETE Statement” in this section. Information on the local selection expression construct is included under “Selection Expression” in this section. Information on the record id construct is included under “Specifying a Dictionary Record” in Section 2, “Using Advanced Data Dictionary System (ADDS) Extensions.”

All data management (DM) expressions that are valid for the type can be used. (Refer to the *InfoExec SIM Programming Guide* and Volume 1 of this manual for more information concerning DM expressions in SIM and ALGOL.) The CURRENT function cannot be used in an assignment except in the selection expression syntax of a local selection expression.

Additional information relating to data management expressions is included under “Using Data Management Functions and Expressions” in this section.

Examples

In the following example, the attribute BIRTHDATE of the query STUQ is assigned the value BDATE.

```
ASSIGN (STUQ.BIRTHDATE ,BDATE);
```

The next two examples show the syntax that can be used to assign values to attributes.

```
ASSIGN (SPOUSE.NAME, "HELEN"); % single-statement update
ASSIGN (STUQ.CHILDREN.NAME_OF_CHILD, "BILLY"); % multiple-statement
```

The following example shows the compound attribute assign construct.

```
INCLUDE MANAGER
   (ASSIGN (CHILDREN,
           ASSIGN (CHILDREN.NAME_OF_CHILD,"HARRY"),
           ASSIGN (CHILDREN.AGE,16),
           ASSIGN (CHILDREN.SEX,"MALE"));
    ASSIGN (NAME,"LARRY");
    ASSIGN (SOC_SEC_NO_,99999999));
```

In this example, the value CLASS_ERV is removed from the multivalued attribute COURSE_TAKING. And, in the same query, a new value is added to the attribute COURSE_TAKING. The new value is the course with number 512A.

```
EXCLUDE (STUQ.COURSE_TAKING (NOLIMIT), CLASS_ERV);
INCLUDE (STUQ.COURSE_TAKING ,[COURSE WITH COURSE_NO = 512A]);
```

The following example is a multiple-statement modify query and an INSERT statement. The EXCLUDE destination must be a multivalued entity-valued attribute. No local selection expression can be used.

```
INSERT DEPARTMENT
     (ASSIGN(DEPT_NO.,4321);
     (ASSIGN(NAME_OF_DEPT, "MATHEMATICS");
      EXCLUDE(COURSE_OFFERED));
STARTMODIFY DEPTQ WHERE DEPT_NO = 4321;
EXCLUDE (DEPTQ.COURSE_OFFERED,
     [COURSE_OFFERED WHERE TITLE = "REMEDIAL MATH II"]);
APPLYMODIFY (DEPTQ)
```

SIM DELETE Statement

<delete statement>

```
── DELETE ── <class ID> ─┬──────────────────────────┬─ WHERE ──────────→
                         └─ <limit specification> ─┘

→─ <selection expression> ──────────────────────────────────────────────┤
```

<limit specification>

```
─ ( ──┬─ LIMIT  ─ <integer expression> ─┬─ ) ───────────────────────────┤
      │                                 │
      └─ NOLIMIT ───────────────────────┘
```

Explanation

The DELETE statement removes entities from the SIM database. All entities from the class that satisfy the selection expression are deleted. If an entity is deleted from a class that is a superclass, the entity is deleted from all its subclasses. If an entity is deleted from a subclass, it does not affect its superclass.

You can use DMUPDATECOUNT to access the number of entities that were deleted. DMUPDATECOUNT is an exception field of the exception word. If the delete operation did not get an exception, but no entity was deleted, a warning is issued. The warning bit in the exception word (bit 1:1) is turned on.

Additional information relating to the DMUPDATECOUNT exception field is included under "Exception Handling of SIM Statements" in this section.

The class ID construct identifies the class.

The limit specification determines the number of entities to be deleted. With the NOLIMIT specification, all occurrences are deleted. With the LIMIT specification, the maximum number of occurrences to delete is designated by the integer expression. If the actual number of occurrences is greater than the integer expression, an exception is returned and no deletions are processed. The default limit is 1.

The WHERE selection expression syntax associates a selection expression with the statement.

Additional information relating to the class ID construct is included under "Declaring a SIM Database" in this section. Information on the selection expression construct is included under "Selection Expressions" in this section.

Examples

In the following example, all entities of the class INSTRUCTOR are deleted if their rank is CONTRACT. If one or more entities are deleted, the message “ONE ENTITY DELETED” is displayed. STATUS would contain the number of entities deleted.

```
STATUS := DELETE INSTRUCTOR WHERE RANK="CONTRACT";
IF REAL (STATUS).DMUPDATECOUNT > 1 THEN DISPLAY
     ("ONE ENTITY DELETED");
```

In the following example, a maximum of 10 members of the class DEPARTMENT can be deleted. The entities must meet the selection criteria of the PHYSICAL SCIENCES building.

```
DELETE DEPARTMENT (LIMIT 10) WHERE BUILDING = "PHYSICAL SCIENCES";
```

This example deletes entities in the class STUDENT depending on the selection criteria. TRANS1 and TRANS2 are reference identifiers. They are different occurrences of the multivalued attribute.

```
TRANSCRIPT_RECORD.
    DELETE STUDENT
    WHERE TRANS1.SEMESTER = FALL AND TRANS2 = CURRENT (TRANS_QUERY);
```

DISCARD Statement

<discard statement>

```
— DISCARD  — ( — <query ID> — ) ——————————————————————————————┤
```

Explanation

The discard statement frees the control structure resources associated with a query. The query is closed but the transaction remains active and the SIM database remains open.

The query ID is a currently active query.

Additional information relating to the query ID construct is included under "Declaring a Query Data Type" in this section.

Example

GET_INSTRUCTOR is the currently active query. In this example, the query is terminated.

```
  DISCARD (GET_INSTRUCTOR);
```

SIM ENDTRANSACTION Statement

<endtransaction statement>

```
— ENDTRANSACTION  ──┬─────────────────────────────────────────┬────────────┤
                    │                                         │
                    └─ <outputheadername with send options> ─┘
```

Explanation

The SIM ENDTRANSACTION statement takes a program out of transaction state. All applied transactions are committed once the statement is executed. All queries using SELECT statements within the transaction state and all nonapplied multiple-statement queries are closed.

The ENDTRANSACTION statement can be used in conjunction with the COMS Data Communication Interface (DCI) library.

Additional information relating to the ENDTRANSACTION statement and COMS is included under “Linking to COMS,” “Declaring Input and Output Headers,” and “SEND Statement” in Section 3, “Using Communications Management System (COMS) Features.”

Additional information relating to the outputheadername with send options construct is included under “COMS ENDTRANSACTION Statement” in Section 3, “Using Communications Management System (COMS) Features.”

The outputheadername construct identifies the COMS Output Header. If the system fails during transaction state, COMS resubmits the message when the program is reexecuted.

The send options of the COMS SEND statement can be included in the syntax.

Examples

Three examples are shown below. In the first statement, no COMS message was included in the transaction. The second statement notes that a COMS message was included. The last statement illustrates a COMS header ID with SEND.

```
ENDTRANSACTION;
ENDTRANSACTION MYOUTHEADER;
ENDTRANSACTION MYOUTHEADER [EMI AFTER SKIP 10];
```

SIM INSERT Statement

<single-statement insert>

```
── INSERT ── <class ID> ─┬────────────────────┬─ ( ──────────────────→
                         └─ <subclass expr> ─┘
  ┌←─────────────── ; ──────────────┐
→─┴─ <db attribute assignment> ─┴─ ) ────────────────────────────────┤
```

<subclass expr>

```
── FROM ──<class ID>── WHERE ──<selection expression>────────────────┤
```

<multiple-statement insert>

```
─ STARTINSERT  ─ ( ─ <query ID> ──┬──────────────────────┬─ ) ─────────────┤
                                  │                      │
                                  └─ <subclass expr> ─┘
```

<apply insert statement>

```
─ APPLYINSERT  ─ ( ─ <query ID> ─ ) ──────────────────────────────────┤
```

Explanation

The INSERT statement inserts new roles for existing entities or new entities with values for their immediate attributes in the declared and opened SIM database. It does not permit assignment of values to extended attributes.

An INSERT statement is valid only in transaction state. It can affect only the specified class, its superclasses, or its inverses.

There are two types of INSERT statements, single and multiple. The single-statement insert is executed as soon as it is encountered. A multiple-statement insert is used to mix attribute assignments among other program statements or to place assignments in other procedures and functions. All computations must be complete before a multiple-statement update is processed. Only one type of insert update can be used in the same query.

The single-statement insert is initiated by an INSERT statement. The multiple-statement insert is initiated by a STARTINSERT statement and must be concluded by an APPLYINSERT statement. The APPLYINSERT statement causes the SIM database to perform the multiple-statement update.

Additional information relating to the INSERT statement is included under “Database Attribute Assignments” in this section.

The class ID specifies what class is affected by the statement.

If the INSERT syntax does not use a subclass expr construct, a new entity and its role are inserted. If the subclass expr construct is used, a role for an existing entity is inserted.

The FROM/WHERE selection expression syntax associates a selection expression with the statement. The subclass expression enables the programmer to take an entity which exists in a class and establish the entity as a member of a subclass in the same generalization hierarchy. The selection expression must select exactly one entity; otherwise, an error is returned. If the subclass expr construct does not appear, the entity is inserted as a new entity in the subclass and its superclasses.

The insert superclass (the class appearing after FROM) can be the immediate superclass of the insert subclass (the class named in the query-name following INSERT), or many intermediate class levels can exist between the two. For each intermediate level, SIM creates the entity as part of the execution of the insert from statement. If an entity already exists at an intermediate level, at the insert subclass expression, a “subrole already exists” error occurs at run time.

In the attribute assignment statements, you can reference attributes in these intermediate classes. In fact, if an intermediate class has a required attribute, the at run time, as assignment statement for that attribute must be executed before the apply insert statement, or a “missing required attribute” error occurs.

The valid SIM database attribute assignments are ASSIGN, INCLUDE, and EXCLUDE.

For a multiple-statement update, the query ID construct associates a query with the update. The same query is also specified in the SIM database attribute assignments that are applied. The query ID must be associated with a database class ID in its declaration.

For STARTINSERT statements, the query ID must refer to a query that has been associated with a class, not a DMRECORD.

Additional information relating to the class ID construct is included under “Declaring a SIM Database” in this section. Information on the query ID construct is included under “Declaring a Query Data Type” in this section. Information on the selection expression construct is included under “Selection Expressions” in this section. Information on the db attribute assignments is included under “Database Attribute Assignments” in this section.

Examples

The following example illustrates a single-statement insert update for the class STUDENT. The INCLUDE updates the multivalued attribute MAJOR_DEPARTMENT.

```
INSERT STUDENT
       (ASSIGN (STUDENT_NO, INPUT_NO);
        ASSIGN (NAME, INPUT_NAME);
        ASSIGN (BIRTHDATE, INPUT_BDATE);
        INCLUDE (MAJOR_DEPARTMENT, DEPT_REFERENCE);
        ASSIGN (CURRENT_ADDRESS, INPUT_CADDR);
        ASSIGN (PERMANENT_ADDRESS, INPUT_PADDR));
```

This example of a multiple-statement insert is used for the query UPDATE_STU_QUERY. The multivalued attribute COURSE_TAKING and the single-valued attribute AGE are updated when the APPLYINSERT is executed.

```
STARTINSERT (UPDATE_STU_QUERY);
     ...
INCLUDE (UPDATE_STU_QUERY.COURSE_TAKING, CURRENT(COURSEQ));
     ...
ASSIGN (UPDATE_STU_QUERY.AGE, CALCULATE_AGE (INPUT_BDATE));
     ...
APPLYINSERT (UPDATE_STU_QUERY);
```

In the following example, by using the subclass expression, only one entity can be selected.

```
INSERT STUDENT FROM INSTRUCTOR WHERE NAME = "DELAWARE"
      (ASSIGN (NAME, "DELAWARE"));
```

SIM MODIFY Statement

<single-statement modify>

```
── MODIFY ──<class ID>─┬──────────────────────────┬─────────────────────→
                       └─<limit specification>────┘

     ┌←─────────── ; ──────────┐
── ( ──┴─<db attribute assignment>─┴──────────────────────────────────→

── ) ─┬──────────────────────────────────┬────────────────────────────┤
      └─ WHERE ──<selection expression>──┘
```

<multiple-statement modify>

```
── STARTMODIFY ── ( ─┬──────────────────────────┬─ <query ID> ───────→
                     └─ <limit specification> ──┘

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

<apply modify statement>

```
── APPLYMODIFY ── ( ── <query ID> ── ) ────────────────────────────────┤
```

Explanation

The MODIFY statement changes existing entities in the declared and opened SIM database. It must have a global selection expression. The number of entries to modify can be limited.

MODIFY statements are valid only in transaction state. Each can affect only the specified class.

There are two types of MODIFY statements, single and multiple. The single-statement modify is executed as soon as it is encountered. A multiple-statement modify is used to mix attribute assignments among other program statements or to place assignments in other procedures and functions. All computations must be complete before a multiple-statement update is processed.

The single-statement modify is initiated by a MODIFY statement. The multiple-statement modify is initiated by a STARTMODIFY statement and must be concluded by an APPLYMODIFY statement. The APPLYMODIFY causes the SIM database system to perform the multiple-statement update.

You can use DUMPDATECOUNT to access the number of entities that were modified. DUMPDATECOUNT is an exception field of the exception word. If the modify operation did not get an exception, but no entity was modified, a warning is issued. The warning bit in the exception word (bit 1:1) is turned on.

Additional information relating to the SIM MODIFY statement is included under "Database Attribute Assignments" in this section. Information on the DMUPDATECOUNT exception field is included under "Exception Handling of SIM Statements" in this section.

The class ID specifies what class will be affected by the statement.

A limit specification determines the number of entities to be modified. Where NOLIMIT is specified, all occurrences are modified. Where LIMIT is specified, all occurrences to the maximum number designated by an integer expression are modified. If more occurrences are found (that it, the actual number of occurrences is greater than the integer expression), an exception is returned. No modifications are processed. The default limit is 1.

The WHERE selection expression syntax associates a selection expression with the statements. It is not needed for modifying class attributes.

The valid SIM database attribute assignments are ASSIGN, INCLUDE, and EXCLUDE.

For a multiple-statement update, the query ID construct associates a query with the update. The same query is also specified in the SIM database attribute assignments that are applied. The query ID must be associated with a database class ID in its declaration.

For STARTMODIFY statements, the query ID must refer to a query that has been associated with a class, not a DMRECORD.

Additional information relating to the class ID construct is included under "Declaring a SIM Database" in this section. Information on the limit specification construct is included under "SIM DELETE Statement" in this section. Information on the db attribute assignments construct is included under "Database Attribute Assignments" in this section. Information on the query ID construct is included under "Declaring a Query Data Type" in this section. Information on the selection expression construct is included under "Selection Expressions" in this section.

Examples

This single-statement MODIFY makes changes to the class STUDENT. When the student number is the input student number, a value is added to the attributes MAJOR_DEPARTMENT, MINOR_DEPARTMENT, and CURRENT_ADDRESS. If one or more entities are modified, the message "ONE ENTITY MODIFIED" is displayed. STATUS contains the actual count.

```
STATUS := MODIFY STUDENT
       (INCLUDE (MAJOR_DEPARTMENT, DEPT_REFERENCE);
       INCLUDE (MINOR_DEPARTMENT, MINOR_DEPT_REFERENCE);
       ASSIGN (CURRENT_ADDRESS,INPUT_CADDR))
WHERE STUDENT_NO = INPUT_STU_NO;
IF REAL(STATUS).DMUPDATECOUNT >1 THEN DISPLAY
       ("ONE ENTITY MODIFIED");
```

This multiple-statement MODIFY is associated with the query UPDATE_STU_QUERY. It is operative only where the major department is PHYSICS. The database attribute EXCLUDE and INCLUDE assignments are applied when APPLYMODIFY is executed. There is no limit on the number of changes that can be applied.

```
STARTMODIFY ((NOLIMIT) UPDATE_STU_QUERY
   WHERE MAJOR_DEPARTMENT = "PHYSICS");
   ...
   EXCLUDE (UPDATE_STU_QUERY.ADVISOR);
   INCLUDE (UPDATE_STU_QUERY.ADVISOR, CURRENT(INSTRUCTOR_QUERY));
 ...
APPLYMODIFY (UPDATE_STU_QUERY);
```

SIM OPEN Statement

<open statement>

```
– OPEN ——┬—————————————┬— <database name> ——————————————————————————————┤
         │             │
         ├— INQUIRY —┤
         │             │
         └— UPDATE ——┘
```

Explanation

A SIM OPEN statement opens a previously declared SIM database and specifies the access mode. An OPEN statements must precede all other SIM statements.

The SIM OPEN statement returns a Boolean result. The statement is used as a usual Boolean expression.

Additional information relating to the SIM OPEN statement is included under “Declaring a SIM Database” and “SIM CLOSE Statement” in this section.

INQUIRY access is read-only access. No update operations can be performed on the SIM database. For INQUIRY access, an exception is returned if any of the following statements are used when the SIM database has been opened:

ABORTTRANSACTION	ENDTRANSACTION
APPLYINSERT	INSERT
APPLYMODIFY	MODIFY
BEGINTRANSACTION	SAVEINSERT
CANCELTRPOINT	SAVEMODIFY
DELETE	SAVETRPOINT

UPDATE access is read/write access. The UPDATE option enables the program to modify the previously declared data base. An exception is returned if the SIM database is already open. If an exception is returned, the state of the database is unchanged.

If neither INQUIRY and UPDATE access is specified, the default access is UPDATE.

The database name is the name of the previously declared SIM database.

Additional information relating to the database name construct is included under “Declaring a Database” in Section 4, “Using the Database Management System II (DMSII) Interface.”

Examples

Shown below, the SIM database UNIVDB is opened. The access method is UPDATE. Therefore the database can be modifies by the program.

```
OPEN UPDATE UNIVDB;
```

In this example, the SIM database TOOLS is opened. The access method is INQUIRY. The Access to the database is read only.

```
OPEN INQUIRY TOOLS;
```

In the following example, the default access method UPDATE is used when opening the SIM database ACCOUNTING.

```
OPEN ACCOUNTING;
```

RETRIEVE Statement

<retrieve statement>

```
── RETRIEVE ── ( ── <query ID> ─┬──────────────────────┬─ ) ─────────┤
                                └─ ; ── <DMRECORD ID> ─┘
```

Explanation

The RETRIEVE statement requests information from the declared and opened SIM database. It retrieves the query in order to make the entities available to your program.

The query ID construct identifies the query to be retrieved. Additional information relating to the query ID construct is included under “declaring a Query Data Type” in this section.

The DMRECORD ID construct identifies a previously defined DMRECORD. If a DMRECORD is specified, the attributes associated with the retrieved query variable are returned into the DMRECORD. The DMRECORD variable must be the same type as the query variable. Additional information on the DMRECORD ID construct is included under “Declaring DMRECORDS” in this section.

If the retrieval is being used only to establish a current path, do not specify a DMRECORD. For example, do not specify a DMRECORD if the query ID will be used as an argument to the CURRENT function.

Additional information regarding the RETRIEVE statement is included under the description of the CURRENT function under “Selection Expressions,” “Type Declaration and Invocation for SIM”, “SELECT Statement”, and “SETTO Statements” in this section.

Examples

The first example demonstrates how a retrieval can be used to establish a current path. INSTRUCTOR_QUERY can be used later as an argument to the CURRENT function. The query variable INSTRUCTOR_QUERY was declared to be associated with a class ID.

```
RETRIEVE (INSTRUCTOR_QUERY);
```

In the second example, the attributes associated with the query STUQ are retrieved and placed into the DMRECORD STUDENT_RECORD.

```
RETRIEVE (STUQ, STUDENT_RECORD);
```

SAVETRPOINT Statement

<savetrpoint statement>

```
── SAVETRPOINT ── ( ── <integer expressions> ── ) ──────────────────┤
```

Explanation

The SAVETRPOINT statement creates intermediate transaction points. These points can be used to specify the extent of a rollback. The intermediate transaction points can be used to cancel transactions without aborting the entire transaction.

The integer expression is used as a marker. SIM requires the marker to be a positive, nonzero value that is unique to the transaction. The integer expression is assigned to a point in the transaction.

An integer is an arithmetic value that has an exponent of zero and no fractional part. Refer to Volume 1 for a discussion of integer expressions.

Additional information relating to accumulated transactions is included under “CANCELTRPOINT Statement” in this section.

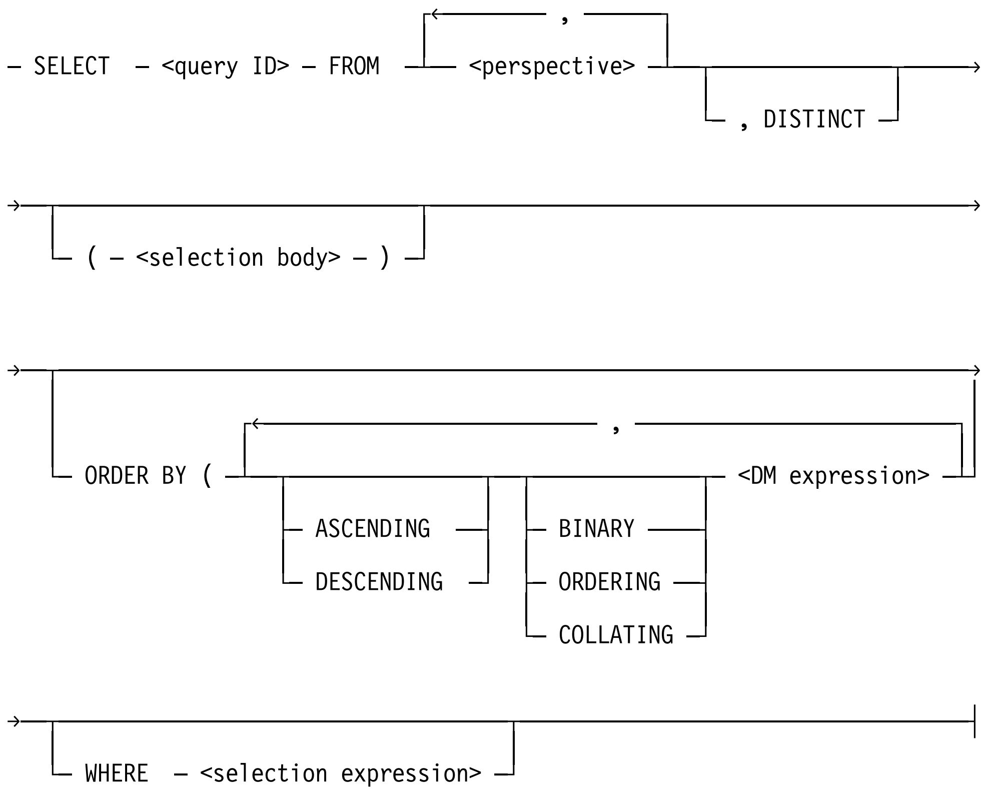
Example

In the following example, an intermediate transaction point is created.

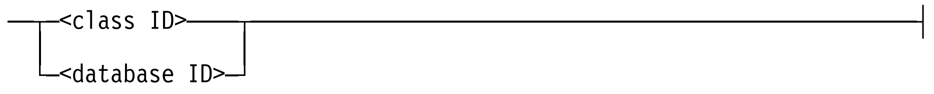
```
SAVETRPOINT (5);
```

SELECT Statement

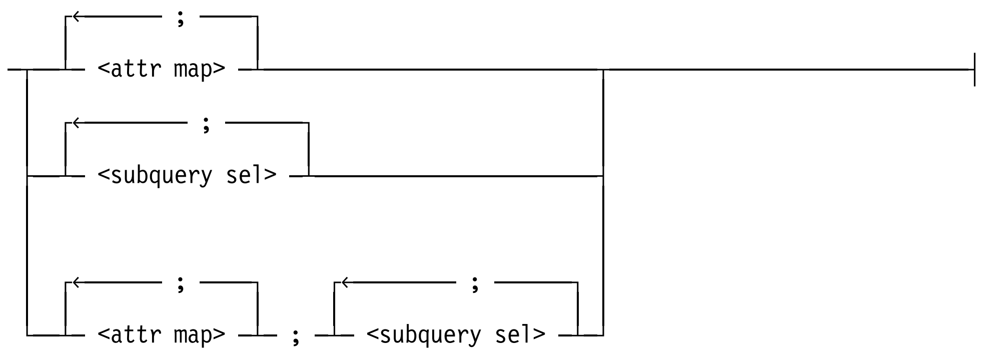
<select statement>



<perspective>



<selection body>



<attr map>

```
─┬─ <field ID> ─┬──────────────────────┬─────────────┬──────────────────────┤
 │              └─ = ─ <DM expression> ─┘             │
 │                                                   │
 │                              ┌←────── ; ───┐      │
 └─ WITH ─ <path expression> ─ ( ─┴─ <attr map> ─┴─ ) ─┘
```

<subquery sel>

```
─ SELECT  ─ <query ID> ─ FROM  ─ <subquery select domain> ──────────────→

→─┬─────────────────────────────┬──────────────────────────────────────┤
  └─ ( ─ <selection body> ─ ) ─┘
```

<subquery select domain>

```
─┬─ <entity-valued qual ID> ─────────────────┬─────────────────────────┤
 ├─ INVERSE  ─ ( ─ <entity-valued qual ID> ─ ) ─┤
 ├─ <multivalued data-valued attribute> ──────┤
 └─ <transitive expression> ──────────────────┘
```

Explanation

The SELECT statement is used to specify what is to be returned from the SIM database and how it is to be returned. A SELECT statement selects a set of entities from the perspective class and associates it with the query variable. (The RETRIEVE statement retrieves the data.) Retrieved data can be presented in a tabular, structured, or hybrid format.

When and how the SELECT statement is issued determines the action taken. There are three possible cases:

- If it is issued within transaction state, all selected entities are locked. No other user can access the locked entities. This ensures a protected read of the data. The query is closed automatically at the end of transaction. All corresponding RETRIEVE statements must be done in the transaction state.
- If issued outside of transaction state, the selected entities are not locked. Retrieval can be either inside or outside of transaction state; however, in either case, entities are not locked. Multiple users can access the data concurrently and there is a risk that the database can change as the data is updated. If the selection is done outside of transaction state and the program issues a RETRIEVE statement inside transaction state, the query is still open when the program leaves transaction state. The DISCARD statement can be used to close the query or the query can stay open until the database is closed.

- If issued outside of transaction state with the SECURED option, the SELECT statement opens a query outside of transaction state and locks the entities. A recommended practice is to use the corresponding RETRIEVE statements within transaction state. The DISCARD statement must be used to close the query.

The global selection is associated with the perspective class (or classes). The local selections are associated with particular paths. A subquery selection can be associated with a local selection.

When a SELECT statement is performed on an already active query, implicit discard and close operations are performed before the SELECT operations is executed; that is, the query is closed and then opened again.

The query ID construct identifies the query. One query variable can be used for several query statements. However, you cannot use the same variable in several SELECT statements at the same time.

The word “FROM” signifies that the following syntax will give the perspective class through which SIM associates the query.

The perspective construct specifies either a class or database ID. Retrieval queries can have more than one class in the perspective. When a retrieval query has more than one class in its perspective, any attributes in the SELECT statement must be fully qualified to identify the class that they apply to. If the perspective is a database ID, only one database can appear in the perspective.

All called constructs are comparable to the SIM “CALLED” function. The function is used to assign a variable to a set of entities. Since these reference variables are not declared in the program, they cannot be referenced beyond the scope of the SELECT statement. The CALLED function can follow a class name, as in the perspective clause syntax, but it cannot follow a variable name that was declared by another CALLED function. Quantifiers cannot be used on variables created by a CALLED function. (Refer to the *InfoExec SIM Programming Guide* for more information on the CALLED function.)

Reference variables are specified before the attribute mapping list to enables the explicit assignment of reference variable or variables to an occurrence of a class or multivalued attributes (MVAs). The program can then manipulate difference occurrences of the class or MVA in the mapping list or selection expression.

The DISTINCT option removes any duplicates and selects only a unique set of data. This option is valid for strictly tabular output only, not for subqueries.

The selection body construct is used to map attributes for retrieval and specify the output format. The mapping constraints are as follows:

- For tabular formatting, do not use the subquery SELECT clause. The compiler requests tabular form if no subquery SELECT clause is specified; otherwise structured formatting is requested.
- For structured formatting, use the subquery SELECT clause. Structured formatting is a subset of hybrid selection.
- For hybrid formatting, combining structured with tabular formats, use attribute maps for items to be displayed in a table and the subquery SELECT clause for items to be displayed in a structured format.

The attr map construct determines how the SELECT statement maps data from the database into DMRECORD fields.

- The field ID construct specifies one field of a record identifier.
- The DM expression construct identifies the attribute from which data is taken. If the attribute's name is different than the name used in the field ID construct, the DM expression is used to clearly identify the attribute.

  If the DM expression contains an attribute and the perspective of the SELECT statement contains more than one class, the attribute must be fully qualified to identify the class it applies to.

  The DM expression cannot include the CURRENT function.
- The WITH option enables you to use a path expression in each of the attribute qualifications of a subsequent attr map construct. The path expression temporarily translates the perspective of the query into another class of the SIM database.

Refer to the *InfoExec SIM Programming Guide* for more information on tabular, structured, and hybrid formatting, as well as the CURRENT and WITH options.

In the attribute mapping list, a SIM attribute with the same name as the DMRECORD field ID that it will be retrieved into, does not need to be specified.

The subquery sel construct identifies and qualifies a query, establishes the attribute mapping characteristics, and associates a selection expression.

The subquery select domain is one of the following:

- A fully qualified entity-valued attribute (EVA)
- The inverse of a fully qualified EVA
- A transitive expression
- A multivalued data-valued attribute

Consult the *InfoExec SIM Programming Guide* for a discussion of these terms.

Pay special attention to situations where the multivalued data-valued attribute is a compound attribute (for example, a record that is made up of several fields). In this situation, any fields that are used in the selection body of a subquery selection should be fully qualified with the complete compound attribute. If they are not fully qualified, the compiler cannot accept them as valid DM expressions in the attr map construct.

The ORDER BY option is used to sort output before it is returned.

- The DM expression must be able to be ordered. Unless tabular output is requested, the DM expression must result in a single value.
- The ASCENDING keyword indicates ascending sort order (from low to high). This is the default sort order. If more than one sort key is indicated, the leftmost is the most significant for ordering. Nulls always sort to the end.
- The DESCENDING keyword indicates descending sort order (from high to low). If more than one sort key is indicated, the leftmost is the most significant for ordering. Nulls always sort to the end.
- The BINARY keyword indicates that as the retrieved data is sorted, SIM compares strings based on their binary value. That is the default type of string comparison when the strings are ASERIESNATIVE (sixteen-bit strings).
- The ORDERING keyword indicates that as the retrieved data is sorted, SIM compares strings based on the current ordering sequence.

  An ordering sequence is a predefined arrangement of members in a character set. In an ordering sequence, characters can be placed in order based on criteria other than their binary value. Difference characters can be assigned the same ordering sequence value (for example, the characters “A” and “a”).
- The COLLATING keyword indicates that as the retrieved data is sorted, SIM compares strings based on the current collating sequence. This is the default type of string comparison when the strings are eight-bit strings (that is, *not* ASERIESNATIVE strings).

  A collating sequence is a predefined arrangement of members in a character set based on ordering sequence *and* additional priority sequence values. Each character has an ordering sequence value and a priority sequence value. Any characters that have the same ordering sequence value are assigned differing priority sequence values. This gives each character a unique combination of values and determines the character's position in the collating sequence.

The WHERE selection expression syntax associates a selection expression with the statement. If the perspective of the SELECT statement contains more than one class, any attributes in the selection expression must be fully qualified to identify the class that they apply to.

Additional information relating to the class ID construct is included under “Declaring a SIM Database” in this section. Information on the field id construct is included under “Declaring DMRECORDs” in this section. Information on the query ID construct is included under “Declaring a Query Data Type” in this section. Information on the selection expression, transitive expression, and path expression constructs is included under “Selection Expressions” in this section.

Additional information relating to the SELECT statement is included under “Using Data Management Functions and Expressions,” “Referencing DMRECORD Fields,” “Selection Expressions,” “RETRIEVE Statement,” “DISCARD Statement,” “SIM ENDTRANSACTION Statement,” “Queries,” and “Retrieval and Update Queries” in this section. Related information is also available in description of the CURRENT function under “Selection Expressions” in this section.

Examples

In the first example, the query INSTRUCTOR_QUERY uses the perspective class INSTRUCTOR. The employee number and hire date are selected if the employee has a salary greater than $20,000. Because of the SECURED option, the selected entities are locked if this SELECT statement is issued outside transaction state. Because the DISTINCT option is used, only a unique set of data is selected. Duplicates are not selected. The selected data is presented in tabular format and in ascending order by employee number.

```
SELECT INSTRUCTOR_QUERY FROM INSTRUCTOR, SECURED, DISTINCT,
     (EMPLOYEE_NO = EMP_NO;
      HDATE = HIRE_DATE)  ORDER BY (ASCENDING EMP_NO)
      WHERE SALARY > 20000;
```

Below are two ways of coding the same query. The query STUQ uses the perspective of STUDENT. In this query, all students majoring in drama will be selected. The report will include their name and minor. In the first case, a subquery, COURSEQ, is used to select the title of the course and the professor teaching the course. The subquery will select only courses where the course number is greater than 300. In the second case no subquery is used.

```
SELECT STUQ FROM STUDENT
     (STU_NAME = NAME;
      MINOR = MINOR_DEPARTMENT.NAME_OF_DEPT;
      SELECT COURSEQ FROM [COURSE_TAKING WHERE COURSE.NO>300]
           (COURSE_TITLE = TITLE;
            PROFESSOR = INSTRUCTOR_TEACHING.NAME))
WHERE MAJOR_DEPARTMENT.NAME_OF_DEPT = "DRAMA";
SELECT STUQ FROM STUDENT
     (STU_NAME = NAME;
      MINOR = MINOR_DEPARTMENT.NAME_OF_DEPT;
      WITH [COURSE_TAKING WHERE COURSE.NO>300]
           (COURSE_TITLE = TITLE;
            PROFESSOR = INSTRUCTOR_TEACHING.NAME))
WHERE MAJOR_DEPARTMENT.NAME_OF_DEPT = "DRAMA";
```

The following example assumes that the query STU_Q selected an entity, namely a student, in a previous action. The statement can then compare the courses taken by that student against the prerequisites for a specific class.

```
SELECT PRE_Q FROM COURSE
    WHERE CURRENT (STU_Q) = STUDENT_TAKING;
```

SETTO Statements

<settoparent state>

```
— SETTOPARENT  — ( — <query ID> — ) ————————————————————————————————————┤
```

<settochild statement>

```
— SETTOCHILD  — ( — <query ID> — ) —————————————————————————————————————┤
```

Explanation

The SETTO statements alter the value for the expected level in a retrieval query. The statements are used with the SIM transitive closure facility.

A reflexive attribute is an entity-valued attribute that refers to the same class of which it is an attribute. The transitive closure facility enables the program to recursively access a reflexive attribute during a retrieval query. This can be used to create circular path expressions. The program can specify at what levels of recursion the transitive retrieval starts and stops.

For tabular output, a reflexive attribute is treated as a multivalued attribute. For structured output, the reflexive attribute can have different values at each level of the structure.

By default, a retrieval traverses the same level and then ends. The SETTO statements can be used to detect and manipulate level changes during traversal. The level can be adjusted, one level at a time, for the next retrieval. SETTOPARENT adjusts the level to the next lower number (the parent level). SETTOCHILD adjusts the level to the next higher number (the child level). These levels are then used in a subsequent RETRIEVE statement.

If SETTOPARENT is used and the current level is not yet exhausted, SIM abandons further accesses at the current level and returns to the parent level. If SETTOCHILD is used, SIM accesses the next child level rather than accessing the next entity at the current level. If there are no entities are the expected level, SIM returns an error condition on the RETRIEVE statement.

The query ID construct identifies the current query.

Additional information relating to the SETTO statements is included under “RETRIEVE Statement” in this section.

Examples

In the first example, the current query is INSTR_QUERY. The level is lowered by one.

```
SETTOPARENT (INSTR_QUERY);
```

In this example, the current query is STUQ. The level is raised by one.

```
SETTOCHILD (STUQ);
```

Exception Handling of SIM Statements

<exception field>

```
──┬─ DMEXCEPTION ──────┬──────────────────────────────────────────┤
  ├─ DMSUBEXCEPTION ───┤
  ├─ DMMOREEXCEPTIONS ─┤
  └─ DMUPDATECOUNT ────┘
```

<DMEXCEPTION mnemonic>

```
──┬─ DMNOERROR ──┬─────────────────────────────────────────────────┤
  ├─ DMWARNING ──┤
  ├─ DMCOMPLETE ─┤
  ├─ DMFAILED ───┤
  └─ DMSYSTEM ───┘
```

Explanation

When compiling SIM database statements, both the compiler and SIM can detect a syntax error. In both cases, the error or warning is returned in the normal way.

When executing SIM statements, exceptions can occur. For example, the program can encounter a fault. Each SIM statement returns a status word. The value of this word specifies whether an exception has occurred and the nature of the exception.

If an exception results from a SIM database operation, but the value is not assigned to an exception variable in the program, the program is terminated. If the value is assigned, no other indication of the exception is given. The ALGOL program is responsible for determining the nature of the exception and responding appropriately.

Consult the *InfoExec SIM Programming Guide* for exception categories and subcategories.

The exception words is a Boolean variable. The value is TRUE if the operation results in an exception; otherwise, it is FALSE.

When an exception occurs, the DM exception routines listed below can be called for further information about the exception.

DMEXCEPTIONMSG is an integer function that translates the current exception to text in the user language. DMEXCEPTIONMSG requires two REAL array rows. The first specifies the language of the message. The second contains the actual message.

The first word of the first REAL array row gives the length of the name of the language. Then name of the language begins in the second row. If zero-length text is passed in first word, the normal MultiLingual System (MLS) selection conventions are used. Otherwise, the second word must specify a language.

The first word of the second REAL array row gives the number of characters returned in the error message. The array, beginning in the second word, should be long enough to receive two lines of 78 characters. This is where the translated text of the error or exception message is returned.

DMNEXTEXCEPTION is a Boolean function that returns the next exception word in the function value. It will not return the text corresponding to the returned exception word. Use DMEXCEPTIONMSG to return the text.

The array returned from DMEXCEPTIONINFO is described by the compiler predeclared DMEXCEPTIONRECORD structure. DMEXCEPTIONRECORD gives exception information about the underlying structure of the database where the exception was encountered.

The layout of DMEXCEPTIONRECORD is

```
TYPE PACKED DMRECORD DMEXCEPTIONTYPE
    (REAL DMSTATUS;
     EBCDIC ARRAY DMLUCNAME [0:29];
     EBCDIC ARRAY DMVERIFYNAME [0:29];
     EBCDIC ARRAY DMDBNAME [0:29];
     EBCDIC ARRAY DMSTRUCTURENAME [0:17]);
DMEXCEPTION TYPE DMEXCEPTIONRECORD;
```

The construct exception field can be used to interrogate the DMSTATUS field of the DMEXCEPTIONRECORD. The DMEXCEPTION field can be compared to the DMEXCEPTION mnemonic to clarify the exception. The DMSUBEXCEPTION field values are defined in the *InfoExec SIM Programming Guide*. The DMMOREEXCEPTION field should be used as a pseudo-Boolean to retrieve the next message.

DMEXCEPTIONINFO is a Boolean function that returns detailed information about the current exception. It returns a structure that can be accessed using the DMEXCEPTIONINFO record field names. Fields of the structure can be meaningful only with certain exceptions. Referencing a field that has no meaning produces an unpredictable value. The DMEXCEPTIONINFO record fields are:

DMSTATUS

> Contains the DMSII result for a physical database exception. It is meaningless for logical database exceptions.

DMSTRUCTURENAME

> Contains the name of the DMSII structure on which a physical database exception was detected. It is meaningless for logical database exceptions.

DMLUCNAME

> Contains the name of the SIM logical component on which a physical or logical database exception was detected

DMDBNAME

> Contains the internal database name upon which a physical or logical database exception, a verification or constraint exception, or a transaction exception occurred.

DMVERIFYNAME

> Contains the name of the VERIFY which caused the verification exception or a description of the attribute option which caused the constraint exception.

The DMEXCEPTION mnemonics are used when a major type of exception is detected to distinguish the exception type. These mnemonics and their corresponding integer values are explained below.

DMNOERROR=1

> Indicates that the last operation was successful (if returned as a result of the operation) or, when calling DMNEXTEXCEPTION, that no further errors exist.

DMWARNING=1

> Contains information about occurrences within the system that the user should be aware of, but which do not affect the results of the operations. This warning is returned in a result word which is FALSE.

DMCOMPLETE=2

> Contains an indication of the end of a sequence of operations.

DMFAILED=3

> Contains reasons for the failure of a query or an operation to complete properly.

DMSYSTEM=4

> Contains exceptions detected by the SIM system which are fatal to the user program and possibly to SIM itself. The program should discontinue operations against the current database. The current database should be closed.

DMSUBEXCEPTION is an exception type. It provides more details as to the exact nature of the exception. It yields a numeric value identifying the subexception of the major exception. Refer to the *InfoExec SIM Programming Guide* for the numeric values and a detailed explanation.

DMMOREEXCEPTIONS is another exception type. it is used to indicate that there were multiple errors. The errors are returned in descending order of severity, ascending order of occurrence. Only the last detected error is returned in the exception word. To access all the errors, use the function DMNEXTEXCEPTION.

DMUPDATECOUNT is not an exception type. It is used to access the number of entities updated in an update operation. DMUPDATECOUNT is valid only when used with MODIFY and DELETE statements.

Additional information relating to the exception fields is included under "SIM MODIFY Statement" and "SIM DELETE Statement" in this section.

Example

In the following example, the ERRORWORD is a Boolean variable. In the retrieval of PROF_QUERY, when the sequence of operations is complete, close the input file. If any error occurs, place the text of the error, in English, into ERRTEXT. Write the content of the message.

```
BOOLEAN ERRORWORD;
   .
   .
ERRORWORD := RETRIEVE (PROF_QUERY);
IF REAL (ERRORWORD).DMERROR THEN
   IF REAL (ERRORWORD).DMEXCEPTION = DMCOMPLETE THEN
      CLOSE (INPUT_FILE)
         ...
   ELSE
   BEGIN
      DMEXCEPTIONMSG (ENGLISH_LANG, ERRTEXT[*]);
      WRITE (ERRFILE,FMT,ERRTEXT[*]);
   END;
```

SIM Sample Programs

Example 1 highlights the use of multiple-statement MODIFY and INSERT updates. It also illustrates the use of EXCLUDE assignments.

Example 2 demonstrates the hybrid retrieval technique. Some extended attributes are retrieved in tabular form, and some in structure form.

Example 3 demonstrates the use of transitive closure and the statements SETTOPARENT and SETTOCHILD.

Example 4 updates a SIM database by using the features of the COMS direct-window interface.

Example 1: Using Project-Employee Projects

The following program enables you to add or drop projects from a database. If a project is dropped, the program completes all related assignments by asking for ratings. The program then updates the overall-rating of the project employee.

The program uses multiple-statement updates, tabular retrieval, the AVERAGE function, single-statement updates, the CURRENT function.

```
BEGIN
SEMANTIC DATABASE PROJECTMANAGER (NAME = PROJEMP) :
         (EMPLOYEE,MANAGER,PROJ_EMPLOYEE,INTERIM_MANAGER,
          PROJECT,DEPARTMENT,ASSIGNMENT,PERSON);

TYPE UNPACKED DMRECORD ASS1_REC_TYPE
              (INTEGER ASS1_START_DATE);

ASS1_REC_TYPE ASS1_REC;

QUERY ASS1_Q(ASS1_REC),
      PEMP_Q(PROJ_EMPLOYEE);

DEFINE        PROJ_ADD  = "ADD "#,
              PROJ_DROP = "DROP"#;

EBCDIC ARRAY PROJ_INDICATOR[0:3];

INTEGER PROJ_NUM,
        SS_NUM,
        MSG_LENGTH;

REAL    INPUT_RATING;
BOOLEAN QUERY_RESULT;

ARRAY MESSAGE_ARRAY[0:12],
      LANG_ARRAY[0:5];

DEFINE ABORT_GRACEFULLY =
       MYSELF.STATUS := -1#;

PROCEDURE PROCESS_THE_MESSAGE;
 %        -------------------
   BEGIN
    DMEXCEPTIONMSG(LANG_ARRAY, MESSAGE_ARRAY);
    MSG_LENGTH := MESSAGE_ARRAY[0];
    WRITE(RMT,MSG_LENGTH,POINTER(MESSAGE_ARRAY[1],8));
    END PROCESS_THE_MESSAGE;

PROCEDURE PROCESS_AN_ASSIGNMENT;
 %        ---------------------
```

```
   BEGIN
    QUERY_RESULT := RETRIEVE(ASS1_Q,ASS1_REC);

   WRITE(RMT,<I8,X4,F3.1>,ASS1_REC.ASS1_START_DATE,INPUT_RATING);
    MODIFY ASSIGNMENT
              (ASSIGN(RATING,INPUT_RATING))
              WHERE ASSIGNMENT = CURRENT (ASS1_Q);

   END PROCESS_AN_ASSIGNMENT;

QUERY_RESULT := OPEN UPDATE PROJECTMANAGER;
IF QUERY_RESULT THEN
   BEGIN
   PROCESS_THE_MESSAGE;
   ABORT_GRACEFULLY;
   END;

BEGINTRANSACTION;

READ(RMT,<A4,I6,I6,I6>,PROJ_INDICATOR,PROJ_NUM,SS_NUM,INPUT_RATING);

STARTMODIFY PEMP_Q WHERE SOC_SEC_NO = SS_NUM;

IF PROJ_INDICATOR = PROJ_ADD THEN
   INCLUDE(PEMP_Q.CURRENT_PROJECT,
                  [PROJECT WHERE PROJECT_NUMBER = PROJ_NUM])
ELSE
IF PROJ_INDICATOR = PROJ_DROP THEN
   SELECT ASS1_Q FROM ASSIGNMENT
          (ASS1_START_DATE = START_DATE)
           WHERE
              PROJECT_OF.PROJECT_TEAM.SOC_SEC_NO = SS_NUM AND
              PROJECT_OF.PROJECT_NUMBER = PROJ_NUM AND
              NOT DMEXISTS(RATING);

WHILE NOT QUERY_RESULT DO PROCESS_AN_ASSIGNMENT;

ASSIGN(PEMP_Q.OVERALL_RATING,DMAVG(ASSIGNMENT_RECORD.RATING));
EXCLUDE(PEMP_Q.CURRENT_PROJECT,[PROJECT_TAKING WHERE
                                PROJECT_NUMBER = PROJ_NUM]);
APPLYMODIFY(PEMP_Q);
ENDTRANSACTION;

QUERY_RESULT:= CLOSE PROJECTMANAGER;

IF REAL(QUERY_RESULT).DMEXCEPTION NEQ DMCOMPLETE THEN
       PROCESS_THE_MESSAGE;

END.
```

Example 2: Archiving Assignments

The following program removes assignments that were completed at least five years ago from the database and stores those assignments on tape.

The program illustrate hybrid retrieval: the program formats some extended attributes in tabular form and others in structured form.

```
BEGIN
SEMANTIC DATABASE PROJEMP:
         (EMPLOYEE,MANAGER,PROJ_EMPLOYEE,INTERIM_MANAGER,
          PROJECT,DEPARTMENT,ASSIGNMENT,PERSON);

TYPE UNPACKED DMRECORD PEQ_REC_TYPE
              (EBCDIC ARRAY PEQ_NAME[0:19];
               INTEGER SOC_SEC_NO;
               EBCDIC ARRAY DEPT[0:19]);

TYPE UNPACKED DMRECORD AQ_REC_TYPE
              (INTEGER AQ_START_DATE;
               INTEGER AQ_END_DATE;
               REAL    AQ_RATING);

PEQ_REC_TYPE  PEQ_REC;
AQ_REC_TYPE   AQ_REC;

QUERY PEQ(PEQ_REC),
      AQ(AQ_REC);

DEFINE  DEADLINE = 010187#;

ARRAY OUT_TEXT[0:26],
      LANG_ARRAY[0:5];

BOOLEAN QUERY_RESULT;

DEFINE ABORT_GRACEFULLY =
       MYSELF.STATUS := -1#;

PROCEDURE PROCESS_THE_MESSAGE;
 %        -------------------
   BEGIN
    DMEXCEPTIONMSG(LANG_ARRAY, OUT_TEXT);
    MSG_LENGTH := OUT_TEXT[0];
    WRITE(RMT,MSG_LENGTH,POINTER(OUT_TEXT[1],8));
    END PROCESS_THE_MESSAGE;

PROCEDURE DO_EMPLOYEE;
 %        -----------
   BEGIN
    QUERY_RESULT := RETRIEVE(PEQ,PEQ_REC);
    IF NOT QUERY_RESULT THEN
      BEGIN
       WRITE(RMT,<A20,X4,I10,X4,A20>,
                   PEQ_REC.PEQ_NAME,PEQ_REC.SOC_SEC_NO,PEQ_REC.DEPT);
       DO
         BEGIN
         QUERY_RESULT := RETRIEVE(AQ,AQ_REC);
         IF REAL(QUERY_RESULT) THEN
            IF QUERY_RESULT.DMEXCEPTION NEQ DMCOMPLETE THEN
               PROCESS_THE_MESSAGE;
            ELSE
         ELSE
            BEGIN
            %  Archive the assignment to TAPE
            DELETE ASSIGNMENT WHERE ASSIGNMENT = CURRENT(AQ);
            END;
         END
       UNTIL QUERY_RESULT;
       END
    ELSE
    IF REAL(QUERY_RESULT).DMEXCEPTION NEQ DMCOMPLETE THEN
       PROCESS_THE_MESSAGE;

   END DO_EMPLOYEE;

QUERY_RESULT := OPEN UPDATE PROJEMP;
IF QUERY_RESULT THEN
   BEGIN
   PROCESS_THE_MESSAGE;
   ABORT_GRACEFULLY;
   END;

BEGINTRANSACTION;

SELECT PEQ FROM PROJ_EMPLOYEE
       (PEQ_NAME = NAME;
        % SOC_SEC_NO need not be specified
        DEPT = DEPT_IN.DEPT_TITLE;
        SELECT AQ FROM ASSIGNMENT_RECORD
               (AQ_START_DATE = START_DATE;
                AQ_END_DATE   = END_DATE;
                AQ_RATING     = RATING))
       WHERE SOME(ASSIGNMENT_RECORD.END_DATE) <  DEADLINE;
DO
   DO_EMPLOYEE
UNTIL QUERY_RESULT;

ENDTRANSACTION;

CLOSE PROJEMP;
END.
```

Example 3: Listing Subprojects

The following program lists the subprojects for a specific project, including subproject of subprojects. The program uses transitive closure and the related set statements.

```
BEGIN
SEMANTIC DATABASE PROJEMP:
         (EMPLOYEE,MANAGER,PROJ_EMPLOYEE,INTERIM_MANAGER,
          PROJECT,DEPARTMENT,ASSIGNMENT,PERSON);

TYPE UNPACKED DMRECORD PEQ_REC_TYPE
              (EBCDIC ARRAY PQ_TITLE[0:29]);
TYPE UNPACKED DMRECORD SQ_REC_TYPE
              (EBCDIC ARRAY SQ_TITLE[0:29]);

PEQ_REC_TYPE  PEQ_REC;
SQ_REC_TYPE   SQ_REC;

QUERY PEQ(PEQ_REC),
      SQ(SQ_REC);

BOOLEAN ALL_DONE, QUERY_RESULT;
INTEGER COUNTER;
ARRAY OUT_TEXT[0:26],
      LANG_ARRAY[0:5];

DEFINE ABORT_GRACEFULLY =
       MYSELF.STATUS := -1#;

PROCEDURE PROCESS_THE_MESSAGE;
 %        -------------------
   BEGIN
    DMEXCEPTIONMSG(LANG_ARRAY, OUT_TEXT);
    MSG_LENGTH := OUT_TEXT[0];
    WRITE(RMT,MSG_LENGTH,POINTER(OUT_TEXT[1],8));
    END PROCESS_THE_MESSAGE;

PROCEDURE GET_SUBPROJECT;
 %        --------------
   BEGIN
    WRITE(RMT,<A30>,SQ_REC.SQ_TITLE);
    SETTOCHILD(SQ);
    COUNTER := * + 1;

QUERY_RESULT := RETRIEVE(SQ,SQ_REC);
    IF QUERY_RESULT THEN
       IF REAL(QUERY_RESULT).DMEXCEPTION = DMCOMPLETE THEN
          DO
             BEGIN
             SETTOPARENT(SQ);
             COUNTER := *-1;
```

```
              IF COUNTER LEQ 0 THEN
                ALL_DONE := TRUE
              ELSE
                 BEGIN
                 QUERY_RESULT := RETRIEVE(SQ,SQ_REC);
                 IF REAL(QUERY_RESULT).DMEXCEPTION NEQ DMCOMPLETE THEN
                    ALL_DONE := TRUE;
                 END
              END
           UNTIL ALL_DONE OR NOT QUERY_RESULT
        ELSE
           ALL_DONE := TRUE;
    END GET_SUBPROJECT;

QUERY_RESULT := OPEN UPDATE PROJEMP;
IF QUERY_RESULT THEN
   BEGIN
   PROCESS_THE_MESSAGE;
   ABORT_GRACEFULLY;
   END;


BEGINTRANSACTION;

SELECT PQ FROM PROJECT
       (PQ_TITLE = PROJECT_TITLE;
        SELECT SQ FROM TRANSITIVE(SUBPROJECT)
               (SQ_TITLE = PROJECT_TITLE));
        WHERE PROJECT.PROJECT_TITLE = "Master Project"

QUERY_RESULT := RETRIEVE(PQ,PQ_REC);
IF QUERY_RESULT THEN
   BEGIN
   WRITE(RMT,<A15>,"No such project");
   ALL_DONE := TRUE;
   END
ELSE
```

```
   BEGIN
   QUERY_RESULT := RETRIEVE(SQ,SQ_REC);
   IF QUERY_RESULT THEN
      BEGIN
      WRITE(RMT,<A14>,"No subprojects");
      ALL_DONE := TRUE;
      END;
   END;
IF NOT ALL_DONE THEN
   DO
      GET_SUBPROJECT
   UNTIL ALL_DONE;

ENDTRANSACTION;
CLOSE PROJEMP;
END.
```

Example 4: Using COMS with a SIM Database

This program, called ONLINESAIL, tracks sailboat races and updates the SIM database SIMSAILDB by using the following features of the COMS direct-window interface:

- Declared COMS input and output headers
- Trancodes and Module Function Indexes (MFIs)
- Recovery

The direct window, the headers, the trancodes, and an agenda must be defined to COMS to enable the program to run.

```
BEGIN
 %       ONLINESAIL

REAL COMS_STATUS;
TYPE INPUTHEADER COMS_IN_TYPE (ARRAY CONVERSATION [0:59]);
COMS_IN_TYPE COMS_IN;
OUTPUTHEADER COMS_OUT;
FILE RMT (KIND=REMOTE);
LIBRARY DCILIBRARY;
SEMANTIC DATABASE SIMSAILDB:
         (RACE_CALENDAR, ENTRY);

TYPE DMRECORD RACE_REC_TYPE
     (EBCDIC ARRAY RACE_NAME [0:19];
      INTEGER RACE_ID;
      EBCDIC ARRAY RACE_DATE [0:5],
                   RACE_TIME [0:3],
                   RACE_LOCATION [0:19],
                   RACE_SPONSOR [0:9]);
RACE_REC_TYPE RACE_REC;

TYPE DMRECORD ENTRY_REC_TYPE
     (EBCDIC ARRAY ENTRY_BOAT_NAME [0:19],
                   ENTRY_BOAT_ID [0:5];
      INTEGER ENTRY_BOAT_RATING;
      EBCDIC ARRAY ENTRY_BOAT_HELSMAN [0:19],
                   ENTRY_AFF_Y_CLUB [0:14];
      INTEGER ENTRY_RACE_ID);
ENTRY_REC_TYPE ENTRY_REC;

QUERY ENTQ (ENTRY),
      RACEQ (RACE_CALENDAR);

EBCDIC ARRAY SCRATCH [0:255];
       ARRAY LANG_ARRAY [0:5];

INTEGER NUM_KEY,
        E_RACE,
        E_BOAT;
 DEFINE EOT_NOTICE = 99#,
       TEXT_LEN    = 113 #;
EBCDIC ARRAY MSG_TEXT[0 : TEXT_LEN-1];
DEFINE           MSG_TCODE = MSG_TEXT[0] #,
 %               MSG_FILLER
 %               MSG_CREATE_RACE
                     MSG_CR_ID = INTEGER( MSG_TEXT[7],6) #,
                     MSG_CR_NAME = MSG_TEXT[13] #,
                     MSG_CR_DATE = MSG_TEXT[33] #,
                     MSG_CR_TIME = MSG_TEXT[39] #,
                     MSG_CR_LOCATION = MSG_TEXT[43] #,
                     MSG_CR_SPONSOR = MSG_TEXT[63] #,
 %                   FILLER
 %               MSG_ADD_ENTRY REDEFINES MSG_CREATE_RACE
                     MSG_AE_RACE_ID = INTEGER( MSG_TEXT[7],6) #,
                     MSG_AE_ID = MSG_TEXT[13] #,
                     MSG_AE_NAME = MSG_TEXT[19] #,
                     MSG_AE_RATING = INTEGER( MSG_TEXT[39],3) #,
                     MSG_AE_OWNER = MSG_TEXT[43] #,
                     MSG_AE_CLUB = MSG_TEXT[63] #,
 %                   FILLER
 %               MSG_DELETE_ENTRY REDEFINES MSG_CREATE_RACE
                     MSG_DE_RACE_ID = INTEGER( MSG_TEXT[7],6) #,
                     MSG_DE_ID = MSG_TEXT[13] #,
 %                   FILLER
                 MSG_STATUS = MSG_TEXT[83] #;

EBCDIC ARRAY WS_FAMILY [0:39];
       ARRAY WS_MSG [0:28];
DEFINE MSG_1 = WS_MSG [0]#,
       MSG_2 = WS_MSG [12]#;


BOOLEAN B;

PROCEDURE SEND_MSG;
   %  Send the message back to the originating station.  Do
   %  not specify an output agenda.  Make sure to test
   %  the result of the SEND operation.
   BEGIN
   COMS_OUT.DESTCOUNT := 1;
   COMS_OUT.DESTINATIONDESG := 0;
   COMS_OUT.STATUSVALUE := 0;
   COMS_STATUS := SEND(COMS_OUT, TEXT_LEN, MSG_TEXT);
   IF NOT(COMS_STATUS = 0 OR COMS_STATUS = 92) THEN
      DISPLAY("Online Program SEND Err: " !! STRING8(C
   END SEND_MSG;
PROCEDURE SIM_ERR_RTN;
   %  Get the error message from SIM.  It can be up to 176 bytes.
   BEGIN
   WRITE(RMT,<"SIM Error: Race=",I6," Boat='
  DMEXCEPTIONMSG (LANG_ARRAY, WS_MSG);
```

```
      WRITE(RMT,78,MSG_1);
      WRITE(RMT,78,MSG_2);
      END SIM_ERR_RTN;

   PROCEDURE CREATE_RACE;
      %  Enter a new race in the database.
      BEGIN
      E_RACE := MSG_CR_ID;
      B := BEGINTRANSACTION;
      IF B THEN
         BEGIN
         SIM_ERR_RTN;
         SEND_MSG;
         END
      ELSE
         BEGIN
         B := INSERT RACE_CALENDAR
                (ASSIGN (RACE_NAME, STRING(MSG_CR_NAME,20));
                 ASSIGN (RACE_ID, E_RACE);
                 ASSIGN (RACE_DATE, STRING(MSG_CR_DATE,6));
                 ASSIGN (RACE_TIME, STRING(MSG_CR_TIME,4));
                 ASSIGN (RACE_LOCATION, STRING(MSG_CR_LOCATION,20));
                 ASSIGN (RACE_SPONSOR, STRING(MSG_CR_SPONSOR,10)));
         IF B THEN
            REPLACE MSG_STATUS BY "Store Error", '
            REPLACE MSG_STATUS BY "Race Added", '
         IF B THEN
            SIM_ERR_RTN
         ELSE
            SEND_MSG;
         END;
      END CREATE_RACE;

   PROCEDURE ADD_ENTRY;
      %  Enter a boat in a race.  Check to see if the race exists.
      %   If a DM error occurs, it indicates a duplicate entry.
      BEGIN
      NUM_KEY := MSG_AE_RACE_ID;
      E_RACE := MSG_AE_RACE_ID;
      E_BOAT := MSG_AE_RATING;

     SELECT RACEQ FROM RACE_CALENDAR
          WHERE RACE_ID = NUM_KEY;
      B := RETRIEVE (RACEQ);
      DISCARD (RACEQ);
      IF B THEN
         BEGIN
        REPLACE MSG_STATUS BY "Race Not Found", '
         END
      ELSE
         BEGIN
         B := BEGINTRANSACTION;
```

```
            IF B THEN
               SIM_ERR_RTN
            ELSE
               BEGIN
               B := INSERT ENTRY
                       (ASSIGN (ENTRY_BOAT_NAME, STRING(MSG_AE_NAME,20));
                        ASSIGN (ENTRY_BOAT_ID, STRING(MSG_AE_ID,6));
                        ASSIGN (ENTRY_BOAT_RATING, E_BOAT);
                        ASSIGN (ENTRY_BOAT_HELMSMAN, STRING(MSG_AE_OWNER,20));
                        ASSIGN (ENTRY_RACE_ID, E_RACE));
               IF B THEN
                  BEGIN
                  SIM_ERR_RTN;
                  REPLACE MSG_STATUS BY "Insert Error", '
               ELSE
                  BEGIN
                  REPLACE MSG_STATUS BY "Boat Added", '
                  IF B THEN
                     SIM_ERR_RTN;
                  END;
               END;
            SEND_MSG;
            END;
         END ADD_ENTRY;

      PROCEDURE DELETE_ENTRY;
         %  Delete a boat from a race.  First check to see if the boat is
         %  entered.  (SIM always returns an OK result so be sure to check.)
         %  If the boat is entered, delete it.
         BEGIN
         NUM_KEY := MSG_DE_RACE_ID;

        SELECT ENTQ FROM ENTRY
             WHERE ENTRY_RACE_ID = NUM_KEY AND
                   ENTRY_BOAT_ID = STRING(MSG_DE_ID,6);
         B := RETRIEVE (ENTQ);
         DISCARD (ENTQ);
         IF B THEN
            BEGIN
            REPLACE MSG_STATUS BY "Boat Entry Not Found", '<
            SEND_MSG;
            END
         ELSE
            BEGIN
            B := BEGINTRANSACTION;
            IF B THEN
               SIM_ERR_RTN
            ELSE
               BEGIN
               B := DELETE ENTRY WHERE ENTRY_RACE_ID = NUM_KEY AND
                                       ENTRY_BOAT_ID = STRING(MSG_DE_ID,6);
               IF B THEN
            BEGIN
            SIM_ERR_RTN;
            REPLACE MSG_STATUS BY "Found But Not Deleted'
         ELSE
            BEGIN
            REPLACE MSG_STATUS BY "Boat Deleted", '
            IF B THEN
               SIM_ERR_RTN;
            END;
         END;
      SEND_MSG;
      END;
   END DELETE_ENTRY;

PROCEDURE CHECK_COMS_INPUT_ERRORS;
   %  Check for COMS control messages.
   BEGIN
   CASE COMS_STATUS OF
        BEGIN
    93: REPLACE MSG_STATUS BY "MSG Causes Abort, Do Not Retry'
    20:
   100:
   101:
   102: REPLACE MSG_STATUS BY "Error in STA Attach/Detachment'
     0:
    92:
    99:
    ELSE:; % A good message, recovery message, or EOT notification.
        END;
   IF COMS_IN.FUNCTIONSTATUS < 0 THEN
      BEGIN
      REPLACE MSG_STATUS BY "Negative Function Code",
      SEND_MSG;
      END;
   END CHECK_COMS_INPUT_ERRORS;

PROCEDURE CLOSE_DOWN;
   %  Close the database.
   BEGIN
   CLOSE SIMSAILDB;
   END;

PROCEDURE PROCESS_TRANSACTION;
   %  Since the transaction type is based on the function index, make
   %  sure the function index is within range.
   BEGIN
   CASE COMS_IN.FUNCTIONINDEX OF
        BEGIN
```

```
 ELSE:BEGIN
        REPLACE MSG_STATUS BY
                "Invalid Trans Code", " '
        END;
   1:   CREATE_RACE;
   2:   ADD_ENTRY;
   3:   DELETE_ENTRY;
        END;
   END PROCESS_TRANSACTION;

PROCEDURE PROCESS_COMS_INPUT;
   %  Gets the next message from COMS.  If the status returned is an
   %  EOT_NOTICE, go to EOT.  Otherwise,  make sure that it is a valid
   %  message before processing it.
   BEGIN
   REPLACE MSG_TEXT BY " " FOR TEXT_LEN;
   COMS_STATUS := RECEIVE(COMS_IN, MSG_TEXT);
   IF COMS_STATUS NEQ EOT_NOTICE THEN
      BEGIN
      CHECK_COMS_INPUT_ERRORS;
      IF COMS_STATUS = 0 OR COMS_STATUS = 92 AND
         (COMS_IN.FUNCTIONSTATUS NEQ 0)            THEN
         PROCESS_TRANSACTION;
      END;
   END;

REPLACE SCRATCH BY MYSELF.EXCEPTIONTASK.EXCEPTIONTASK.NAME;
DCILIBRARY.LIBACCESS := VALUE(BYTITLE);
DCILIBRARY.TITLE := STRING(SCRATCH[0],256);

B := OPEN UPDATE SIMSAILDB;
IF B THEN
   BEGIN
   DMEXCEPTIONMSG(LANG_ARRAY,WS_MSG);
   WRITE(RMT,78,MSG_1);
   END;

REPLACE WS_FAMILY BY MYSELF.EXCEPTIONTASK.EXCEPTIONTASK.NAME;
DCILIBRARY.TITLE := STRING(WS_FAMILY[0],40);
REPLACE WS_FAMILY BY MYSELF.FAMILY;
REPLACE MYSELF.FAMILY BY "DISK = DISK ONLY.";
ENABLE(COMS_IN,"ONLINE");
REPLACE MYSELF.FAMILY BY WS_FAMILY;

DO
  PROCESS_COMS_INPUT
UNTIL COMS_STATUS = EOT_NOTICE;

CLOSE_DOWN;
END.
```

Section 8
Using TransIT Open/OLTP

TransIT Open/OLTP for enterprise servers enables an application program to update multiple DMSII databases in a coordinated fashion. That is, Open/OLTP ensures that either all updates are committed or all are rolled back. Pre-Open/OLTP (traditional online transaction processing) software enables an application to update multiple databases, but it is the responsibility of the application program to coordinate the updates and the recoveries of those databases.

Open/OLTP also implements a client/server model. Clients invoke services but do not update the databases directly. Service providers can provide multiple services where a service is used to update a DMSII database. The client/server model applies to COMS online programs only because the implementation of services is provided by COMS.

Open/OLTP is based on the X/Open Distributed Transaction Processing (DTP) model, which is specified in standards developed by the X/Open Company, Ltd.

Example

The following example shows the logic for a client:

```
Open databases.

Start global transaction.
Call Service 1 to debit savings account.
Call Service 2 to credit mutual fund account.
If services completed successfully then
      Commit global transaction
Else
      Rollback global transaction.

Close databases.
```

You can access Open/OTLP through ALGOL by calling library entry points that are exported by the Open/OLTP software. These entry points may be called in the same manner as any other library. For a description of the X/Open TX and XATMI interfaces for ALGOL, refer to the *TransIT Open/OLTP for A Series Programming Guide*. This guide explains how to use the include file (supplied in ALGOL) to access the TX and XATMI interfaces. This guide also provides sample programs written in ALGOL that call the library interfaces for using Open/OLTP.

More information about Transit Open/OLTP

For further information regarding TransIT Open/OLTP consult the *TransIT Open/OLTP for A Series Programming Guide.*

Appendix A
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 A–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 part of a constant appears in boldface, 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 appears in boldface, 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 part of it appears in boldface.

```
—— 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> ── ) ──────────────────────────┤
```

Sample Input	**Explanation**
LOCK (FILE4)	LOCK is a constant and cannot be altered. Because no part of the word appears in boldface, 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 ──┘
```

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 ──┤
                                       ├─ + ───┤
                                       └─ – ───┘
```

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>

```
                  ┌←──────────────────── , ──────────────────┐
── ENTITY REFERENCE ──┴─<entity ref ID>── ( ──<class ID>── ) ─┴─────────┤
```

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

```
── PS ── MODIFY ──────────────────────────────────────────────────→

   ┌←──────────────────── , ────────────────────┐
→──┬┴─┬─<request number>─────────────────────────┴┬──────────────────→
   │  └─<request number>── ─ ──<request number>──┘ │
   └─ ALL ─┬───────────────────────────────────────┤
           └─ EXCEPTIONS ──────────────────────────┘

→──┬───────────────────────────────────────────────────────────────┤
   │ ┌←───────────────── , ───────────────┐ │
   └─┴─┬──────┬─<file attribute phrase>──┬┴─┘
       │└ ─ ─┘                           │
       ├──────┬─<print modifier phrase>──┘
       └ ─ ─ ─┘
```

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.

Appendix B
Extended ALGOL Reserved Words

A <reserved word> in Extended ALGOL has the same syntax as an identifier. The reserved words are divided into three types. In the following explanation, each type is discussed separately and the reserved words for that specific type are listed. An alphabetical listing of all reserved words can be found at the end of this appendix.

Type 1 Reserved Words

Listed below are type 1 reserved words. A reserved word of type 1 can never be declared as an identifier; that is, it has a predefined meaning that cannot be changed. For example, because LIST is a type 1 reserved word, the declaration

```
ARRAY LIST[0:999]
```

is flagged with a syntax error.

ALPHA	FILE	REFERENCE
ARRAY	FOR	STEP
BEGIN	FORMAT	SWITCH
BOOLEAN	GO	TASK
COMMENT	IF	THEN
CONTINUE	INTEGER	TRANSLATETABLE
DIRECT	LABEL	TRUE
DO	LIST	TRUTHSET
DOUBLE	LONG	UNTIL
END	OWN	VALUE
ELSE	POINTER	WHILE
EVENT	PROCEDURE	ZIP
FALSE	REAL	

Type 2 Reserved Words

Listed below are type 2 reserved words. A reserved word of type 2 can be redeclared as an identifier; it then loses its predefined meaning in the scope of that declaration. For example, because IN is a type 2 reserved word, the declaration

```
FILE IN(KIND = READER)
```

is legal, but in the scope of the declaration, the statement

```
SCAN P WHILE IN ALPHA
```

is flagged with a syntax error on the word "IN".

If a type 2 reserved word is used as a variable in a program but is not declared as a variable, then the error message that results is not the expected "UNDECLARED IDENTIFIER". Instead, it might be "NO STATEMENT CAN START WITH THIS".

ABORTTRANSACTION	CHANGEFILE	DGAMMA
ABS	CHECKPOINT	DICTIONARY
ACCEPT	CHECKSUM	DIGITS
AFTER	CLN	DIMP
ALL	CLOSE	DINTEGER
AND	COLLATING	DISABLE
APPLYINSERT	COMPILETIME	DISCARD
APPLYMODIFY	COMPLEX	DISPLAY
ARCCOS	CONJUGATE	DIV
ARCSIN	COS	DLGAMMA
ARCTAN	COSH	DLN
ARCTAN2	COTAN	DLOG
ARRAYSEARCH	CSIN	DMABS
ASCII	CSQRT	DMAVG
ATANH	CURRENT	DMAX
ATTACH	DABS	DMCHR
AVAILABLE	DAND	DMCONTAINS
BCL	DARCCOS	DMCOUNT
BEFORE	DARCSIN	DMEQUIV
BINARY	DARCTAN	DMEXCEPTIONINFO
BREAKPOINT	DARCTAN2	DMEXCEPTIONMSG
BY	DCOS	DMEXCLUDES
CABS	DCOSH	DMEXISTS
CALL	DEALLOCATE	DMEXT
CALLING	DECIMAL	DMFUNCTION
CANCEL	DEFINE	DMIN
CANCELTRPOINT	DELINKLIBRARY	DMISA
CASE	DELTA	DMLENGTH
CAT	DEQV	DMMATCH
CAUSE	DERF	DMMAX
CAUSEANDRESET	DERFC	DMMIN

continued

continued

CCOS	DETACH	DMNEXTEXCEPTION
CEXP	DEXP	DMPOS
DMPRED	FIRSTWORD	NOCR
DMRECORD	FIX	NOLF
DMRPT	FORMAL	NONE
DMSORT	FORWARD	NORMALIZE
DMSUCC	FDMPOS	FREE
DMSUM	FREEZE	NOT
DMTRUNC	GAMMA	NUMBERIC
DMUPDATECOUNT	GEQ	OF
DNABS	GTR	OFFSET
DNOT	DMSUCC	ON
DNOTWAIT	HAPPENED	ONES
DOR	HEAD	OPEN
DROP	HEX	OR
DSCALELEFT	IMAG	ORDER
DSCALERIGHT	IMP	ORDERING
DSCALERIGHTT	IN	OUTPUTHEADER
DSIN	INCLUDE	OUTPUTMESSAGE
DSINH	INPUTHEADER	PICTURE
DSQRT	INTEGERT	POTC
DTAN	INTERRUPT	POTH
DTANH	INVERSE	POTL
DUMP	IS	PROCESS
EBCDIC	INST	PROCESSID
EGI	LB	PROCURE
EMI	LENGTH	PROGRAMDUMP
EMPTY	LEQ	LENGTH
EMPTY4	LIBERATE	PURGE
EMPTY7	LIBRARY	QUERY
EMPTY8	LINE	RANDOM
ENABLE	LINENUMBER	RB
ENTER	LINKLIBRARY	READ
ENTITY	LISTLOOKUP	READLOCK
EQL	LN	RECEIVE
EQV	LNGAMMA	RECORD
EQV_EQL	LOCK	REFERENCE
EQV_GEO	LOG	REMAININGCHARS
EQV_GTR	LSS	REMOVEFILE
EQV_LEQ	MASKSEARCH	REPEAT
EQV_LSS	MAX	REPLACE
EQV_NEQ	MERGE	RESET
ERF	MESSAGECOUNT	RESIZE
ERFC	MESSAGESEARCHER	RETRIEVE
ESI	MIN	REWIND
EXCHANGE	MON	RUN
EXCLUDE	MODIFY	SAVETRPOINT
EXISTS	MONITOR	SCALELEFT
EXP	MUX	SCALERIGHT

continued

EXPORT
EXTERNAL
FILL
FIRST
FIRSTONE
SEEK
SELECT
SEMANTIC SEND
SET
SETACTUALNAME
SETTOCHILD
SETTOPARENT
SIGN
SIN
SINGLE
SINH
SIZE
SKIP
SOME
SORT
SPACE
SQRT
MYJOB
MYSELF
NABS
NEQ
NO
STACKER
STARTINSERT
STARTMODIFY
STATION
STOP
STRING
STRING4
STRING7
STRING8
SUBFILE
SUBROLE
TAIL
TAKE
TAN
TANH
TERMINAL
SCALERIGHTF
SCALERIGHTT
SCAN
SDIGITS
SECONDWORD
THRU
TIME
TIMELIMIT
TIMES
TO
TRANSITIVE
TRANSLATE
TYPE
USING
WAIT
WAITANDRESET
WHEN
WHERE
WITH
WORDS
WRITE

Type 3 Reserved Words

Listed below are type 3 reserved words. A reserved word of type 3 is context-sensitive. It can be redeclared as an identifier, and if it is used where the syntax calls for that reserved word, it carries the predefined meaning; otherwise, it carries the user-declared meaning. The different meanings for the type 3 reserved word STATUS are illustrated in the following example.

```
BEGIN
   TASK T;
   REAL STATUS;
   % IN THE NEXT STATEMENT, "STATUS" IS A REAL VARIABLE
   STATUS := 4.5;
   % IN THE NEXT STATEMENT, "STATUS" IS A TASK ATTRIBUTE
   IF T.STATUS = VALUE(TERMINATED) THEN
      % IN THE NEXT STATEMENT, "STATUS" IS A REAL VARIABLE
      STATUS := 10.0;
END.
```

Type 3 reserved words include the following:

- File attribute names
- Task attribute names
- Library attribute names
- Direct array attribute names
- Mnemonics for attribute values

All file attributes, direct array attributes, and mnemonics described in the *File Attributes Programming Reference Manual* are type 3 reserved words in ALGOL. All task attributes and mnemonics described in the *Task Attributes Programming Reference Manual* are type 3 reserved words in ALGOL.

ACTUALNAME	BYTITLE	EXCEPTIONEVENT
ALL	CHARGECODE	EXCEPTIONTASK
ALPHA6	CLASS	EXPONENTOVERFLOW
ALPHA7	CODE	EXPONENTUNDERFLOW
ALPHA8	COMPILETYPE	FAMILY
ANYFAULT	COREESTIMATE	FILECARDS
ARRAYS	CRUNCH	FILES
AS	DBS	FUNCTIONNAME
ASCIITOBCL	DECIMALPOINTISCOMMA	HEXTOASCII
ASCIITOEBCDIC	DECLAREDPRIORITY	HEXTOBCL
ASCIITOHEX	DISCARD	HEXTOEBCDIC
BACKUPPREFIX	DISK	HISTORY
BASE	DISKPACK	INITIATOR
BCLTOASCII	EBCDICTOASCII	INTEGEROVERFLOW
BCLTOEBCDIC	EBCDICTOBCL	INTNAME

continued

continued

BCLTOHEX	EBCDICTOHEX	INVALIDADDRESS
BYFUNCTION	ELAPSEDTIME	INVALIDINDEX
INVALIDOP	ORGUNIT	STACKSIZE
INVALIDPROGRAM	OUT	STARTTIME
WORD		
JOBNUMBER	PACK	STATUS
LIBACCESS	PAGED	STOPPOINT
LIBPARAMETER	PARTNER	STRINGPROTECT
LIBRARIES	PERMANENT	SUBSPACES
LOCKED	PRIVATELIBRARIES	TADS
LOOP	PROCESSIOTIME	TARGETTIME
MAXCARDS	PROCESSTIME	TASKATTERR
MAXIOTIME	PROGRAMMEDOPERATOR	TASKFILE
MAXLINES	REEL	TASKVALUE
MAXPROCTIME	RESTART	TEMPORARY
MEMORYPARITY	RETAIN	TITLE
MEMORYPROJECT	SCANPARITY	TYPE
NAME	SIBS	USERCODE
OFFER	STACKNO	ZERODIVIDE
OPTION		

Reserved Words Alphabetical Listing

The following is an alphabetical list of reserved words for Extended ALGOL. The number in parentheses following each word indicates the type of the reserved word. For example, "FOR (1)" indicates that FOR is a type 1 reserved word.

ABORTTRANSACTION (2)
ABS (2)
ACCEPT (2)
ACTUALNAME (3)
AFTER (2)
ALL (2)
ALL (3)
ALPHA (1)
ALPHA6 (3)
ALPHA7 (3)
ALPHA8 (3)
AND (2)
ANYFAULT (3)
APPLYINSERT (2)
APPLYMODIFY (2)
ARCCOS (2)
ARCSIN (2)
ARCTAN (2)
ARCTAN2 (2)
ARRAY (1)
ARRAYS (3)
ARRAYSEARCH (2)
AS (3)
ASCII (2)
ASCIITOBCL (3)
ASCIITOEBCDIC (3)
ASCIITOHEX (3)
ATANH (2)
ATTACH (2)
AVAILABEL (2)
BACKUPPREFIX (3)
BASE (3)
BCL (2)
BCLTOASCII (3)
BCLTOEBCDIC (3)
BCLTOHEX (3)
BEFORE (2)
BEGIN (1)
BINARY (2)
BOOLEAN (1)
BREAKPOINT (2)
BY (2)
BYFUNCTION (3)

CALL (2)
CALLING (2)
CANCEL (2)
CANCELTRPOINT (2)
CASE (2)
CAT (2)
CAUSE (2)
CAUSEANDRESET (2)
CCOS (2)
CEXP (2)
CHANGEFILE (2)
CHARGECODE (3)
CHECKPOINT (2)
CHECKSUM (2)
CLASS (3)
CLN (2)
CLOSE (2)
CODE (3)
COLLATING (2)
COMMENT (1)
COMPILETIME (2)
COMPILETYPE (3)
COMPLEX (2)
CONJUGATE (2)
CONTINUE (1)
COREESTIMATE (3)
COS 2)
COSH (2)
COTAN (2)
CRUNCH (2)
CSIN (2)
CSIN (2)
CSQRT (2)
CURRENT (2)
DABS (2)
DAND (2)
DARCCOS (2)
DARCSIN (2)
DARCTAN (2)
DARCTAN2 (2)
DBS (3)
DCOS (2)
DCOSH (2)

DECLAREDPRIORITY (3)
DEFINE (2)
DELINKLIBRARY (2)
DELTA (2)
DEQV (2)
DERF (2)
DERFC (2)
DETACH (2)
DEXP (2)
DGAMMA (2)
DICTIONARY (2)
DIGITS (2)
DIMP (2)
DINTEGER (2)
DIRECT (1)
DISABLE (2)
DISCARD (2)
DISCARD (3)
DISK (3)
DISKPACK (3)
DISPLAY (2)
DIV (2)
DLGAMMA (2)
DLN (2)
DLOG (2)
DMABS (2)
DMAVG (2)
DMAX (2)
DMCHR (2)
DMCONTAINS (2)
DMCOUNT (2)
DMEQUIV (2)
DMEXCEPTIONINFO (2)
DMEXCEPTIONMSG (2)
DMEXCLUDES (2)
DMEXISTS (2)
DMEXT (2)
DMFUNCTION (2)
DMIN (2)
DMISA (2)
DMLENGTH (2)
DMMATCH (2)
DMMAX (2)

continued

continued

BYTITLE (3)
CABS (2)
DMNEXTEXCEPTION (2)
DMPOS (2)
DMPRED (2)
DMRECORD (2)
DMRPT (2)
DMSQRT (2)
DMSUCC (2)
DMSUM (2)
DMTRUNC (2)
DMUPDATECOUNT (2)
DNABS (2)
DNOT (2)
DO (1)
DONTWAIT (2)
DOR (2)
DOUBLE (1)
DROP (2)
DSCALELEFT (2)
DSCALERIGHT (2)
DSCALERIGHTT (2)
DSIN (2)
DSINH (2)
DSQRT (2)
DTAN (2)
DTANH (2)
DUMP (2)
EBCDIC (2)
EBCDICTOASCII (3)
EBCDICTOBCL (3)
EBCDICTOHEX (3)
EGI (2)
ELAPSEDTIME (3)
ELSE (1)
EMI (2)
EMPTY (2)
EMPTY4 (2)
EMPTY7 (2)
EMPTY8 (2)
ENABLE (2)
END (1)
ENTIER (2)
ENTITY (2)
EQL (2)
EQV (2)
EQV_EQL (2)
EQV_GEQ (2)

DEALLOCATE (2)
DECIMAL (2)
DECIMALPOINTISCOMMA (3)
ESI (2)
EVENT (1)
EXCEPTIONEVENT (3)
EXCEPTIONTASK (3)
EXCHANGE (2)
EXCLUDE (2)
EXISTS (2)
EXP (2)
EXPONENTOVERFLOW (3)
EXPONENTUNDERFLOW (3)
EXPORT (2)
EXTERNAL (2)
FALSE (1)
FAMILY (3)
FILE (1)
FILECARDS (3)
FILES (3)
FILL (2)
FIRST (2)
FIRSTONE (2)
FIRSTWORD (2)
FIX (2)
FOR (1)
FORMAL (2)
FORMAT (1)
FORWARD (2)
FREE (2)
FREEZE (2)
FUNCTIONNAME (3)
GAMMA (2)
GEQ (2)
GO (1)
GTR (2)
HAPPENED (2)
HEAD (2)
HEX (2)
HEXTOASCII (3)
HEXTOBCL (3)
HEXTOEBCDIC (3)
HISTORY (3)
IF (1)
IMAG (2)
IMP (2)
IN (2)
INCLUDE (2)

DMMIN (2)
INTNAME (3)
INVALIDADDRESS (3)
INVALIDINDEX (3)
INVALIDOP (3)
INVALIDPROGRAMWORD (3)
INVERSE (2)
IS (2)
ISNT (2)
JOBNUMBER (3)
LABEL (1)
LB (2)
LENGTH (2)
LEQ (2)
LIBACCESS (3)
LIBERATE (2)
LIBPARAMETER (3)
LIBRARIES (3)
LIBRARY (2)
LINE (2)
LINENUMBER (2)
LINKLIBRARY (2)
LIST (1)
LISTLOOKUP (2)
LN (2)
LNGAMMA (2)
LOCK (2)
LOCKED (3)
LOG (2)
LONG (1)
LOOP (3)
LSS (2)
MASKSEARCH (2)
MAX (2)
MAXCARDS (3)
MAXIOTIME (3)
MAXLINES (3)
MAXPROCTIME (3)
MEMORYPARITY (3)
MEMORYPROJECT (3)
MERGE (2)
MESSAGECOUNT (2)
MESSAGESEARCHER (2)
MIN (2)
MOD (2)
MODIFY (2)
MONITOR (2)
MUX (2)

continued

EQV_GTR (2)
EQV_LEQ (2)
EQV_LSS (2)
EQV_NEQ (2)
ERF (2)
ERFC (2)
NEQ (2)
NO (2)
NOCR (2)
NOLF (2)
NONE (2)
NORMALIZE (2)
NOT (2)
NUMERIC (2)
OF (2)
OFFER (3)
OFFSET (2)
ON (2)
ONES (2)
OPEN (2)
OPTION (3)
OR (2)
ORDER (2)
ORDERING (2)
ORGUNIT (3)
OUT (3)
OUTPUTHEADER (2)
OUTPUTMESSAGE (2)
OWN (1)
PACK (3)
PAGED (3)
PARTNER (3)
PERMANENT (3)
PICTURE (2)
POINTER (1)
POTC (2)
POTH (2)
POTL (2)
PRIVATELIBRARIES (3)
PROCEDURE (1)
PROCESS (2)
PROCESSID (2)
PROCESSIOTIME (3)
PROCESSTIME (3)
PROCURE (2)
PROGRAMDUMP (2)
PROGRAMMEDOPERATOR (3)
PURGE (2)
QUERY (2)

INITIATOR (3)
INPUTHEADER (2)
INTEGER (1)
INTEGEROVERFLOW (3)
INTEGERT (2)
INTERRUPT (2)
REEL (3)
REFERENCE (1)
REFERENCE (2)
REMAININGCHARS (2)
REMOVEFILE (2)
REPEAT (2)
REPLACE (2)
RESET (2)
RESIZE (2)
RESTART (3)
RETAIN (3)
RETRIEVE (2)
REWIND (2)
RUN (2)
SAVETRPOINT (2)
SCALELEFT (2)
SCALERIGHT (2)
SCALERIGHTF (2)
SCALERIGHTT (2)
SCAN (2)
SCANPARITY (3)
SDIGITS (2)
SECONDWORD (2)
SEEK (2)
SELECT (2)
SEND (2)
SET (2)
SETACTUALNAME (2)
SETTOCHILD (2)
SETTOPARENT (2)
SIBS (3)
SIGN (2)
SIN (2)
SINGLE (2)
SINH (2)
SIZE (2)
SKIP (2)
SOME (2)
SORT (2)
SPACE (2)
SQRT (2)
STACKER (2)
STACKNO (3)

MYJOB (2)
MYSELF (2)
NABS (2)
NAME (3)
STOPPOINT (3)
STRING (2)
STRING4 (2)
STRING7 (2)
STRING8 (2)
STRINGPROJECT (3)
SUBFILE (2)
SUBROLE (2)
SUBSPACES (3)
SWITCH (1)
TADS (3)
TAIL (2)
TAKE (2)
TAN (2)
TANH (2)
TARGETTIME (3)
TASK (1)
TASKATTERR (3)
TASKFILE (3)
TASKVALUE (3)
TEMPORARY (3)
TERMINAL (2)
THEN (1)
THRU (2)
TIME (2)
TIMELIMIT (2)
TIMES (2)
TITLE (3)
TO (2)
TRANSITIVE (2)
TRANSLATE (2)
TRANSLATETABLE (1)
TRUE (1)
TRUTHSET (1)
TYPE (2)
TYPE (3)
UNTIL (1)
USERCODE (3)
USING (2)
VALUE (1)
WAIT (2)
WAITANDRESET (2)
WHEN (2)
WHERE (2)
WHILE (1)

continued

RANDOM (2)
RB (2)
READ (2)
READLOCK (2)
REAL (1)
RECEIVE (2)
RECORD (2)
STACKSIZE (3)
STARTINSERT (2)
STARTMODIFY (2)
STARTTIME (3)
STATION (2)
STATUS (3)
STEP (1)
STOP (2)
WITH (2)
WORDS (2)
WRITE (2)
ZERODIVIDE (3)
ZIP (1)

Index