UNISYS

Product Information Announcement

o New Release ● Revision o Update o New Mail Code

Title

MCP/AS COBOL ANSI-85 Programming Reference Manual, Volume 2: Product Interfaces (8600 1526–202)

This announces a revision of the *MCP/AS COBOL ANSI-85 Programming Reference Manual Volume 2: Product Interfaces* for HMP NX 4.0 and SSR 45.1, dated June 1998.

Previous title: *ClearPath HMP NX and A Series COBOL ANSI-85 Programming Reference Manual Volume 2: Product Interfaces*

This manual has been revised to include the following changes:

- Section 2: Addition of service function mnemonics to Table 2–5
- Section 3: Clarification of the syntax for the DMSII SET statement
- Section 3: Addition of double–octet characters in identifiers
- Section 4: Deletion of paragraph, re: changing $DICTIONARY after IDENTIFICATION DIVISION
- Sections 4, 5 and 6: Addition of compiler control images after...FROM DICTIONARY...invocations
- Section 6: Modification of the FORM-KEY function line (COMS_OUT_AGENDA)

Various other technical and editorial changes were made to improve the quality and usability of this manual.

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: AS194	System:	MCP/AS
		Release:	HMP 4.0 and SSR 45.1
		Date:	June 1998
		Part number:	8600 1526–202

MCP/AS

UNISYS

COBOL ANSI-85

Programming Reference Manual

Volume 2 Product Interfaces

HMP 4.0 and SSR 45.1 June 1998

Printed in USA
Priced Item 8600 1526–202

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of a duly executed agreement to purchase or lease equipment or to license software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED – Use, reproduction, or disclosure is restricted by DFARS 252.227–7013 and 252.211–7015/FAR 52.227–14 & 52.227-19 for commercial computer software.

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 4. Using the ADDS Program Interface

Section 5. Using the SDF Plus Program Interface

Contents

Tables

Examples

About This Manual

Purpose

This manual is the second volume of a two-volume reference set. The *COBOL ANSI-85 Programming Reference Manual*, *Volume 1: Basic Implementation* provides the syntax and general usage of standard elements of Unisys COBOL ANSI-85. This manual, Volume 2, contains information on using Unisys COBOL ANSI-85 to write application programs that interface with the following products:

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

SDF is a member of the InterPro (Interactive Productivity) family of products. ADDS and SIM are members of the InfoExec (Information Executive) family of products.

Scope

For each product, this manual presents information on

- The purposes of COBOL85 interfaces with a product
- The product features and functions that an interface can manipulate
- The uses of the language extensions of a product
- The means by which each language extension is used
- The information necessary for writing COBOL85 programs that need to use the capabilities of COMS and DMSII

Unisys COBOL ANSI-85 is implemented for use on A Series systems. It is based on, and compatible with, the American National Standard Programming Language COBOL ANSI X3.23-1985.

Volume 2 briefly describes how to access Open/OLTP from COBOL85 and provides references to the Open/OLTP documentation.

Audience

The information in this manual is intended for application programmers and systems analysts who are programming in COBOL85 and require the capabilities of one or more Unisys products. The secondary audience consists of technical support personnel and information systems management.

Prerequisites

To use this manual, you should be familiar with COBOL85, the product or products being used, and the programming concepts for the products.

How to Use This Manual

The information in this volume complements Volume 1 of the *COBOL ANSI-85 Programming Reference Manual* and the manuals for the COMS and DMSII products. For more information about these products, refer to the documentation for each product. (See "Related Product Information" later in this section.) For information on using Unisys COBOL ANSI-85 not specific to these products, refer to Volume 1.

This manual discusses each product in a separate section, to allow you to access only the sections that apply to your needs. The first section is an introduction, containing tables that summarize the extensions used with each product.

COBOL Coding Examples

Many of the discussions in this volume use COBOL85 code to illustrate an aspect of a product interface. Most examples do not include line numbers; it is assumed that the first character of a line of source code is located in the appropriate column. Complete program examples have line numbers to provide continuity for programs that span several pages.

Document References

Throughout this manual, *Volume 1* refers to the *COBOL ANSI-85 Programming Reference Manual*, *Volume 1: Basic Implementation.* Unless otherwise specified, manuals referred to in the text are for A Series systems.

Acronyms

All acronyms used in this volume appear in the glossary, with their full spellings and definitions. Acronyms are spelled out the first time they occur in the manual.

COBOL Syntax

The syntax for the language extensions is described in COBOL notation; for a complete explanation of COBOL format notation, refer to Volume 1**.**

System Messages

This manual does not contain system messages. You can find system messages for COMS or DMSII in the respective product manuals; syntax error messages for COBOL85 appear in Volume 1.

Organization

This manual is organized into seven sections and two appendixes. The first section is an introduction to program interfaces. Each subsequent section describes a program interface for a specific product. The appendixes provide related information. There is an index at the end of this volume. A brief description of the contents of the manual follows

Section 1. Introduction to COBOL85 Program Interfaces

Section 1 describes the concept of a COBOL85 interface with a product. It also alphabetically lists the extensions for each product, with a summarized description of each extension.

Section 2. Using the COMS Program Interface

Section 2 explains the COMS interface, the extensions and their relationships to the product features, and information on using COMS with DMSII. This section includes the syntax and explanation for each extension, and program examples.

Section 3. Using the DMSII Program Interface

Section 3 describes the extensions developed for the DMSII interface that allow you to invoke a database, use data management statements and database items, and handle exception conditions. It includes the syntax, explanation, and program examples for each extension.

Section 4. Using the ADDS Program Interface

Section 4 describes the extensions developed for the ADDS program interface that enable you to manipulate data, define complex data structures, and update and report on entities or structures in the data dictionary. It includes the syntax, explanation, and program examples for each extension.

Section 5. Using the SDF Plus Program Interface

Section 5 describes the SDF Plus user interface management system that enables you to define a complete form-based user interface for an application program. This section describes the extensions that enable you to invoke form record library descriptions into your program as COBOL85 declarations, to send messages and receive transactions, and to send error transaction messages and send text messages. It includes the syntax, explanation, and program examples for each extension.

Section 6. Using the SDF Program Interface

Section 6 describes the COBOL85 user interface to SDF, which enables you to define a complete form-based user interface for an application program. This interface is designed primarily for users migrating from V Series COBOL74 to A Series COBOL85. Current A Series users will probably want to use SDF Plus, which is described in Section 5.

Section 7. TransIT Open/OLTP

This section provides a brief description of Open/OLTP. It also contains references to other manuals for further information about using COBOL74 to create Open/OLTP applications.

Appendix A. Reserved Words

Appendix A lists COBOL reserved words. A reserved word has a special meaning to COBOL and cannot be redefined by the programmer. New reserved words that have been developed for Unisys COBOL ANSI-85 are indicated with a double asterisk (**).

Appendix B. User-Defined Words

Appendix B lists variable words or terms that are required in clauses or statements, for which you must define names.

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.

COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation

This manual describes the basic features of the MCP/AS COBOL ANSI-85 programming language. This manual is written for programmers who are familiar with programming concepts.

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 Data and Structure Definition Language (DASDL) Programming Reference Manual

This manual provides instructions for defining and maintaining a Data Management System II (DMSII) database using DASDL. This manual is written for database administrators and staff.

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.

InfoExec Capabilities Manual

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

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.

System Software Utilities Operations Reference Manual

This manual provides information on the system utilities BARS, CARDLINE, COMPARE, DCAUDITOR, DCSTATUS, DUMPALL, DUMPANALYZER, FILECOPY, FILEDATA, HARDCOPY INTERACTIVEXREF, ISTUTILITY, LOGANALYZER, LOGGER, PATCH, PRINTCOPY RLTABLEGEN, SORT, XREFANALYZER, and the V Series conversion utilities. It also provides information on the PL/I Indexed Sequential Access Method (PLIISAM), KEYEDIO support, Peripheral Test Driver (PTD), and mathematical functions. This manual is written for applications programmers, system support personnel, and operators.

Task Attributes Programming Reference Manual

This manual describes all the 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.

TransIT Open/OLTP for A Series Programming Guide

This guide provides a conceptual and procedural overview of how to develop TransIT Open/OLTP application programs on ClearPath and A Series systems, by using ALGOL, C, COBOL74, and COBOL85 programming languages. Topics covered in this guide include overviews of the TX and XATMI application program interfaces and programming concepts related to the client/server model of communication. Extensive programming examples are also included.

Acknowledgement

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee in connection therewith.

The authors and copyright holders of the copyrighted material used herein have specifically authorized the use of this material, in whole or in part, in the COBOL specifications. These authors or copyright holders are the following:

- FLOW-MATIC, programming for the UNIVAC I and II, Data Automation Systems, copyrighted 1958, by Sperry Rand Corporation
- IBM Commercial Translator, form No. F 28-8013, copyrighted 1959 by IBM
- FACT, DSI 27 A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Section 1
Introduction to COBOL85 Program Interfaces

A program interface comprises the syntax, conventions, and protocols of a programming language that are used to manipulate a software product. Interfaces are developed to aid you in writing applications that use the full functionality of enterprise server products.

Using Program Interfaces for Specific Products

COBOL85 program interfaces for specific products are easy to use, flexible, and efficient. They make it easier to manipulate special product features by making those features available to an application program written in COBOL85. The COBOL85 program interfaces consist of the Unisys extensions to COBOL ANSI-85.

COBOL85 program interfaces have been developed for the following products:

- Communications Management System (COMS)
- Data Management System II (DMSII)
- Advanced Data Dictionary System (ADDS)
- Screen Design Facility Plus (SDF Plus)
- Screen Design Facility (SDF)
- TransIT Open/OLTP

Each program interface is described in a section of this manual. Each section includes suggested ways to implement some of the features in your application of the product, including any requirements or options for using a combination of interfaces.

Using Language Extensions for Specific Products

The extensions for Unisys COBOL ANSI-85 comprise two groups:

- Extensions used with all Unisys products
- Extensions used only with specific Unisys products

Volume 1 describes the extensions that are used with enterprise server systems. This volume explains extensions that are used only with specific Unisys products; however, the general extensions also appear in this volume to provide context and to illustrate their use with specific products.

The tables in this section summarize alphabetically the extensions used with COMS, DMSII, ADDS, SDF Plus, and SDF. Detailed discussions of these extensions are provided in the appropriate sections of this volume. The COBOL85 syntax for accessing Open/OLTP is described in Section 7.

COMS Extensions

The following extensions have been developed for the COMS interface. For detailed information about each extension, refer to Section 2, "Using the COMS Program Interface."

Extension	Explanation
COMS header declaration	Provides information about the message for input or output. (See "Using COMS Headers" in Section 2.)
ACCEPT MESSAGE COUNT statement	Tells how many messages are in a program application queue.
DISABLE statement	Closes a COMS direct window to a station or disconnects a station reached through a modem or a CP2000 station.
ENABLE statement	Initializes the interface between COMS and a program, and opens a COMS direct window to a station not currently attached to COMS.
RECEIVE statement	Makes a message and pertinent information available to the program from a queue maintained by the message control system.
SEND statement	Releases a message or message segment to one or more output queues maintained by the message control system.
VALUE clause	Accesses COMS service function routines and allows a mnemonic parameter to be passed to obtain a numeric result. The VALUE clause uses the CALL statement. (See "Using Parameters by Value" in Section 2.)

DMSII Extensions

The following extensions have been developed for the DMSII interface. For detailed information on each extension, refer to Section 3, "Using the DMSII Program Interface."

Extension	Explanation
ABORT-TRANSACTION statement	Discards updates made in a transaction after a BEGIN-TRANSACTION statement.
ASSIGN statement	Establishes a relationship between a record in one data set and a record in the same or another data set.
BEGIN-TRANSACTION statement	Places a program in transaction state. This statement is used only with audited data bases.
CANCEL TRANSACTION POINT statement	Discards all updates in a transaction to an intermediate transaction point or to the beginning of the transaction.
CLOSE statement	Closes a database unconditionally when further access is no longer required. A syncpoint is caused in the audit trail, and all locked records are freed.
COMPUTE statement	Assigns a value to a Boolean item in the current record of a data set.
CREATE statement	Initializes the user work area of a data set record.
Database (DB) declaration	Provides information about one or more databases during compilation.
Database equation	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.
Data management (DM) attributes	Allows read-only access to the count, record type, and population information in a record.
Data set reference entry	Specifies which structures are to be invoked from the declared data base

continued

Extension	Explanation
DELETE statement	Finds a specific record, and then locks and deletes it.
DMERROR Use procedure	Handles exception conditions.
DMSTATUS database status word	Indicates whether an exception condition has occurred and identifies the exception.
DMSTRUCTURE function	Determines the structure number of a data set, set, or subset programmatically. This data management structure function can be used to analyze exception condition results.
DMTERMINATE statement	Halts a program with a fault when an exception occurs that the program does not handle.
END-TRANSACTION statement	Takes a program out of 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.
FREE statement	Unlocks the current record.
GENERATE statement	Creates a subset in one operation. All subsets must be disjoint bit vectors.
IF statement	Tests an item to determine if it contains a NULL value.
INSERT statement	Places a record into a manual subset.
LOCK/MODIFY statement	Finds a record or structure and locks it against concurrent modification by another user.
ON EXCEPTION clause	Handles exception conditions. It is placed after certain data management statements.
OPEN statement	Opens a database for subsequent access and designates the access mode.
RECREATE statement	Partially initializes the user work area.

continued

Extension	Explanation
REMOVE statement	Finds a record, and then locks it and removes it from the subset.
SAVE TRANSACTION POINT statement	Provides an intermediate point in a transaction for audit.
SECURE statement	Locks a record in such a way that other programs can read the record but not update it.
Selection expressions	Identifies a certain record in a data set. Selection expressions are used with FIND, LOCK, MODIFY, and DELETE statements.
SET statement	Alters the current path or changes the value of an item in the current record.
STORE statement	Places a new or modified record into a data set.

ADDS Extensions

The following extensions have been developed for the ADDS interface. For detailed information about each extension, refer to Section 4, "Using the ADDS Program Interface."

Extension	Explanation
DICTIONARY clause	Specifies the function name of the dictionary library. The function name is the name equated to a library code file when using the SL (Support Library) system command.
DIRECTORY clause	Specifies the directory under which the entity is stored in the data dictionary.
FD statement	Invokes all file attributes of the file named in the SELECT statement. FD refers to file description.
FROM DICTIONARY clause	Enables you to obtain an entity from the dictionary.
INVOKE clause	Assigns an alias identifier to an entity name invoked from the dictionary.
PROGRAM-DIRECTORY clause	Specifies the directory of the program to be tracked.

continued

Extension	Explanation
PROGRAM-NAME clause	Specifies the name of the entity of type program that is to be tracked. This clause is needed if the entity tracking is defined in ADDS.
PROGRAM-VERSION clause	Specifies the version of the program to be tracked.
SD statement	Invokes all file attributes of the file named in the SELECT statement. SD refers to a sort-merge file description.
SELECT statement	Enables you to include files from the dictionary in your program.
VERSION clause	Identifies the 6-digit numeric literal version number of the record description.

SDF Plus Extensions

The following COBOL85 extensions are provided for use with the SDF Plus interface. Refer to Section 5 for detailed information about these extensions.

Extension	Explanation
DICTIONARY statement	Identifies the dictionary to be used during compilation.
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.
FROM DICTIONARY clause	Invokes an SDF Plus form record library from the dictionary.
GLOBAL clause	Allows subprograms to reference form record libraries declared in host programs.
READ FORM statement	Causes a form record to be read from a specified remote file and stored in a specified buffer.
REDEFINES clause	Enables multiple form record libraries to have the same record area.

continued

Extension	Explanation
SAME RECORD AREA clause	Enables all form record descriptions in the form record library to be invoked as redefinitions of the first form record description in the form record library.
SEPARATE RECORD AREA clause	Invokes each form record in the form record library as a separate data description with its own record area.
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.
WRITE FORM statement	Writes the contents of a form record to a specified remote file.
WRITE FORM TEXT statement	Causes the contents of a text array to be written to a remote file.

SDF Extensions

The following COBOL85 extensions are provided for use with the SDF interface. Refer to Section 6 for detailed information about each extension.

Extension	Explanation
DICTIONARY statement	Identifies the dictionary to be used during compilation.
FORM-KEY function	Enables access to an internal binary number of a form name for programmatic uses. This function is required for using SDF with COMS.
FROM DICTIONARY clause	Invokes a formlibrary from the dictionary.
READ FORM statement	Reads specific and self-identifying forms.
REDEFINES clause	Enables formlibraries invoked into a program to redefine the same record area as the first formlibrary.
SAME RECORD AREA clause	Enables all form descriptions in the formlibrary to be invoked as redefinitions of the first form in the formlibrary.
WRITE FORM statement	Writes forms from a station to a program.

Section 2
Using the COMS Program Interface

This section explains how to use extensions within an application program in order to communicate with COMS through direct windows.

The tasks presented in this section include

- Preparing a communication structure for routing or for describing information about the message.
- Declaring and using COMS headers when receiving and sending messages.
- Preparing to receive and send messages. This preparation includes linking to COMS and initializing a program.
- Using communication statements to receive and send messages.
- Using service functions. For an alphabetized list of the extensions developed for COMS, refer to "COMS Extensions" in Section 1.

What Does the COMS Program Interface Do?

The Communications Management System (COMS) is a general message control system (MCS) that controls online environments. It supports a network of users and handles a high volume of transactions from programs, stations, and remote files.

The program interface for COMS allows you to create online, interactive, and batch programs that take advantage of the features COMS offers for transaction processing through direct windows. The available features and functions depend on the version of COMS that is installed. With the full-featured version of COMS, the program interface allows the program to communicate with COMS using the following COMS functions:

- Message routing by transaction codes (trancodes) and agendas
- Processing message data through processing items
- Security checking of messages
- Service functions
- Dynamic opening of direct windows to terminals not attached to COMS, and dynamic communication by modem
- Synchronized recovery

You can write processing items using COBOL85 with an ALGOL shell. Instructions on using the shell, as well as general concepts for programming for COMS, are provided in the *Communications Management System (COMS) Programming Guide*.

Running DMSII with COMS

COMS can be used with Data Management System II (DMSII). When you run DMSII with COMS, use the following DMSII statements to allow a program interfacing with COMS to support synchronized transactions and recovery:

- ABORT-TRANSACTION
- BEGIN-TRANSACTION
- CLOSE
- DMTERMINATE
- END-TRANSACTION

For information about the syntax and use of these statements and the extensions developed for DMSII, refer to Section 3, “Using the DMSII Program Interface.”

Using Multiple COMS Language Support Libraries

The COBOL ANSI-85 compiler uses the COMS Language Support library to help it describe the various service functions and types of header information needed in your program. The Language Support library is used only during the compilation of programs, unlike the DCI library (which is also used when your programs are running).

Some sites require separate environments. For example, you may want to maintain your normal production environment separate from your test environment. To do this, you may need multiple versions of the COMS Language Support library.

If an environment requires multiple versions of the COMS Language Support library, you can modify the COBOL ANSI-85 compiler to use a different version of the COMS Language Support library. You can modify the library used by the compiler in either of two ways:

- Permanently modify the COBOL ANSI-85 compiler so that it is linked to a different support library
- Temporarily modify the support library name for a single compilation of a program

With either method, you must understand how to use the SL (Support Library) system command.

For the examples that follow in this discussion, assume the following support libraries are defined:

```
SL  COMSLANGSUPPORT = *SYSTEM/COMS/LANGUAGE/SUPPORT
SL  COMSLANGSUPPORT85 = *SYSTEM/COMS/LANGUAGE/SUPPORT/V41xxx
```

In these examples, COMSLANGSUPPORT and COMSLANGSUPPORT85 refer to the function name, and the *xxx* refers to the release level of the SYSTEM/COMS/LANGUAGE/SUPPORT/V41 code file.

Permanently Modifying the COBOL ANSI-85 Compiler

A permanent modification changes the function name of the COMS Language Support library being used by the COBOL ANSI-85 compiler for every compilation done using that compiler. To make a permanent modification, perform the following steps:

1. Copy the compiler under a different name. For example, use

```
COPY *SYSTEM/COBOL85 AS *SYSTEM/COBOL85/V41xxx
```

2. Use the WFL MODIFY command to make a permanent change to the compiler. In this example, use

```
WFL MODIFY *SYSTEM/COBOL85;
    LIBRARY COMSLANGSUPPORT
      ( LIBACCESS = BYFUNCTION,
        FUNCTIONNAME = COMSLANGSUPPORT85 );
```

3. Use the MC (Make Compiler) system command to re-create the compiler. In this example, use

```
MC *SYSTEM/COBOL85
```

The compiler *SYSTEM/COBOL85/V41xxx will now look for a COMS support library with a function name of COMSLANGSUPPORT85. The compiler *SYSTEM/COBOL85 will retain COMSLANGSUPPORT as the function name of its support library.

For information about the SL or MC system commands, refer to the *System Commands Operations Reference Manual*.

Temporarily Modifying the Support Library Name

A temporary modification changes the function name of the COMS Language Support library being used by the COBOL ANSI-85 compiler, but this change only affects a single compilation of a program. In this case, you would create a WFL job to compile the program and include the following library equate syntax:

```
COMPILER LIBRARY COMSLANGSUPPORT
         ( LIBACCESS = BYFUNCTION,
           FUNCTIONNAME = COMSLANGSUPPORT85 );
```

For more information about using a WFL job to compile a program, see the *Work Flow Language (WFL) Programming Reference Manual*.

Preparing the Communication Structure

The program must provide a communication structure for routing and for descriptive information about the message. You provide a communication structure by declaring each of the following in your program:

- An area for the message
- A COMS interface that directs input and output and provides an optional conversation area for user-defined information

Declaring a Message Area

To receive and send messages, you must declare a message area in the Data Division of the program. Always declare the message area as a 01-level record.

Declare the message area with a format and size that are appropriate to the data your program is to receive. If the message area is too small to contain all of the incoming text, COMS truncates the message.

After a message is received, the Text Length field in the header reflects the number of characters in the whole message text.

Example

Example 2–1 shows the declaration for a COMS message area. This declaration occurs in the Data Division.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COMSMSG.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COMS-MESSAGE-AREA.
    02 COMS-MESSAGE           PIC X(1920).
 COMMUNICATION SECTION.
```

Example 2–1. Declaring a COMS Message Area

Declaring a COMS Interface

You declare a COMS interface by using COMS headers for input or output. COMS headers are dynamically built at compilation time. The compiler requests the header structure from COMS and constructs the headers.

COMS headers offer the following advantages:

- COMS handles the location of fields in the headers; therefore, you do not need to know the memory structure.
- COMS identifiers access all fields within the header; therefore, there is no need to rename the fields in the queue structure.
- You do not have to change your program when new releases of COMS occur.
- Headers can be referenced by bound-in procedures.

Using COMS Headers

There are two types of COMS headers:

- Input headers used to receive messages
- Output headers used to send messages

The fields in the input and output headers can be used to receive or send values that provide information or instruction for various activities.

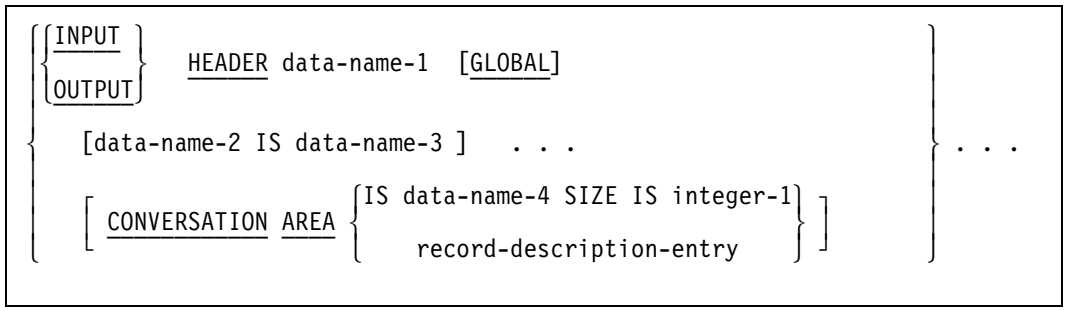
For detailed information on the use of the headers and fields, refer to the *COMS Programming Guide*.

Declaring COMS Headers

The following discussion explains the COBOL85 syntax, rules, and steps for declaring COMS headers in your program. Examples are also provided. For information about the fields of the input and output headers, refer to “COMS Input Header Fields” and “COMS Output Header Fields” later in this section.

Format

The general format for declaring input and output COMS headers is as follows:



Explanation

You can declare input and output headers in the Communication Section of the COBOL85 program. The only syntax items required are the INPUT or OUTPUT HEADER phrase and data-name-1. The preceding syntax invokes the description of the header and provides access to all the fields of the input header. If data-name-2 is not unique to the input header, you can access it with an OF qualification that mentions data-name-1. More information about using the OF qualification, see "Using Set and Data Set Names" in Section 3.

Input and output headers described with the GLOBAL clause are considered to be global headers. A global header can be referenced either from the program in which the global header is declared or from any other program that is contained in the program that declares the global header.

Data-name-2 is any identifier retrieved from COMS. The IS clause renames the identifier.

Data-name-3 is not equated; it replaces the name supplied by COMS. However, a field cannot be renamed with a name that already exists in the header. If renamed, data-name-2 must be referred to by data-name-3.

If you use the CONVERSATION AREA clause, you must also either have data-name-4 and a SIZE phrase, or else map it with the record-description-entry. If you use data-name-4 and the SIZE phrase, the conversation area is defined as a level 02 data description entry with the following format:

```
02 data-name-4  PIC X(integer-1)
```

The SIZE phrase defines the conversation area as a PIC X representation with the length indicated by integer-1.

If you use a record-description-entry to define the conversation area, it must start at level 02. The record-description-entry is added to the end of the header.

For more information on the CONVERSATION AREA clause, see "COMS Input Header Fields" and "COMS Output Header Fields" later in this section.

Example

Example 2–2 shows an example of the declarations for COMS input and output headers. To see declarations for headers within the context of a complete program, see Example 2-24, "COMS Sample Program with a DMSII Database," later in this section.

```
COMMUNICATION SECTION.
INPUT HEADER COMS-IN;
   PROGRAMDESG IS COMS-IN-PROGRAM.
   CONVERSATION AREA.
     02 CA.
         05 CA-1              PIC X(20).
         05 CA-2              PIC X(30).
OUTPUT HEADER COMS-OUT.
```

Example 2–2. Declaring COMS Input and Output Headers

Mapping COMS Data Types to COBOL85

Table 2–1 shows the COMS data types and the valid COBOL85 usage. For information on COMS types and COBOL85 usage for fields in the COMS headers, see "COMS Input Header Fields" and "COMS Output Header Fields" later in this section.

Table 2–1. COMS Data Types and COBOL85 Usage

COMS Type	COBOL85 Usage
Boolean	DMS Boolean
Designator	Real
Display	Display
Mnemonic	Binary
Record	Display
TIME(6)	Real

COMS data types include a COMS designator data type that is used only for specific fields and with service functions. COMS determines the kind of designator and required name from the value that is passed. The designator type is compatible with COBOL85 data items of type real. For more information about COMS designators used with service functions, see "Using COMS Designators," later in this section.

Boolean items are similar to DMSII Boolean items; they can be tested with an IF condition and set with a COMPUTE statement.

Note that when using logical tests against the COMS type TIME(6), you must redefine the type as a PIC X(6) DISPLAY item to test against it.

COMS Input Header Fields

The fields of the input header are COBOL85 attributes of the header. The Conversation Area field is not part of the header provided by COMS; it is an optional user-defined field that is associated with the input header. All other fields of the input header are defined by either COMS or COBOL85. The structure of COMS input headers is obtained from COMS at compilation.

COMS places values (designators and integers) in the input header fields when a RECEIVE, ACCEPT, DISABLE, or ENABLE statement is executed. For information on specific values used in the input header fields, refer to the *COMS Programming Guide*.

A service function translates a designator to a name representing a COMS entity. See "Using Service Functions" later in this section for more information.

COMS uses the input headers of incoming messages for the following tasks:

- Confirming message status
- Passing data in the Conversation Area field
- Detecting queued messages
- Determining message origin
- Processing transaction codes for routing
- Obtaining direct-window notifications

For more information about input headers and fields, refer to the *COMS Programming Guide*. For information about data types, see "Mapping COMS Data Types to COBOL85" and "Using COMS Designators" in this section.

Table 2–2 lists the input header fields in alphabetical order, with the corresponding COBOL85 field names, the COMS data types, and the COBOL85 usages. For an example of coding the input header fields, see the COMS sample program (lines 017004 through 017060) in Example 2–24.

Table 2–2. Input Header Fields

COMS Field Name	COBOL85 Field Name	COMS Type`	COBOL85 Usage
Agenda Designator	AGENDA	Designator	Real
Continuator Data Length	CONTDATALENGTH	Integer	Binary
Continuator Data Offset	CONTDATAOFFSET	Integer	Binary
Continuator Data Status	CONTDATASTATUS	Mnemonic	Binary
Continuator Entry Number	CONTENTRYNUM	Integer	Binary
Function Index	FUNCTIONINDEX	Mnemonic	Binary
Function Status	FUNCTIONSTATUS	Mnemonic	Binary
Message Count	MESSAGECOUNT	Integer	Binary
Program Designator	PROGRAMDESG	Designator	Real
Restart	RESTART	Designator	Real
Security Designator	SECURITYDESG	Designator	Real
Station Designator	STATION	Designator	Real
Status Value	STATUSVALUE	Mnemonic	Binary
Text Length	TEXTLENGTH	Integer	Binary
Timestamp	TIMESTAMP	TIME(6)	Real
Transparent	TRANSPARENT	Boolean	DMS Boolean
Usercode Designator	USERCODE	Designator	Real
User-Defined Conversation Area	User defined	User defined	User defined
VT Flag	VTFLAG	Boolean	DMS Boolean

COMS Output Header Fields

The fields of the output header are COBOL85 attributes of the header. The Conversation Area field is not a field in the header provided by COMS; it is an optional user-defined field that is associated with the output header. All other fields in the header are defined by either COMS or COBOL85. The structure of COMS output headers is obtained from COMS during compilation.

COMS uses the output header when sending messages. You place designators into the output header fields to route outgoing messages and describe their characteristics. To obtain designators, you call service functions to translate names representing COMS entities to designators. See "Calling Service Functions" later in this section for more information.

For information about values returned to the output header to indicate errors in destination routing, refer to the *COMS Programming Guide*.

COMS uses the output header fields for the following tasks:

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

For general information on output headers and fields, refer to the *COMS Programming Guide*. For information on data types, refer to "Mapping COMS Data Types to COBOL85" and "Using COMS Designators" in this section.

Table 2–3 lists the fields of the output header. It shows the field name in COMS and in COBOL85, the COMS data type, and the COBOL85 usage.

Table 2–3. Output Header Fields

COMS Field Name	COBOL85 Field Name	COMS Type	COBOL85 Usage
Agenda Designator	AGENDA	Designator	Real
Casual Output	CASUALOUTPUT	Boolean	DMS Boolean
Continuator Mode	CONTMODE	Mnemonic	Binary
Continuator Data Length	CONTDATALENGTH	Integer	Binary
Continuator Data Offset	CONTDATAOFFSET	Integer	Binary
Continuator Data Status	CONTDATASTATUS	Mnemonic	Binary
Continuator Entry Number	CONTENTRYNUM	Integer	Binary
Destination Count	DESTCOUNT	Integer	Binary
Destination Designator	DESTINATIONDESG	Designator	Real
Delivery Confirmation Flag	CONFIRMFLAG	Boolean	DMS Boolean
Delivery Confirmation Key	CONFIRMKEY	Display	Display
VT Flag	VTFLAG	Boolean	DMS Boolean
Map Alias Name	MAPALIAS	Display	Display
Next Input Agenda Designator	NEXTINPUTAGENDA	Designator	Real
Retain Transaction Mode	RETAINTRANSACTIONMODE	Boolean	DMS Boolean
Set Next Input Agenda	SETNEXTINPUTAGENDA	Boolean	DMS Boolean
Status Value	STATUSVALUE	Mnemonic	Binary
Text Length	TEXTLENGTH	Integer	Binary
Transparent	TRANSPARENT	Boolean	DMS Boolean
User-Defined Conversation Area	User Defined	User Defined	User Defined

Using the VT Flag of the Output Header

You can use the VT (virtual terminal) flag bit of the output header with a COMS direct window. The window can have a virtual terminal name when it is used within a CP2000 environment. The virtual terminal name describes to BNA how the direct window has formatted the output.

A direct-window program can set the VT flag before sending output messages, by using the following syntax:

```
COMPUTE VTFLAG OF OUTHDR = TRUE
```

On input, COMS returns the result in the VT Flag field of the input header. You can test the result directly using the IF statement. For example,

```
IF VTFLAG OF INHDR . . .
```

Requesting Delivery Confirmation on Output

Delivery confirmation is available for network support processor (NSP) and CP2000 stations. This feature of COMS lets a direct window program know when a station has received a particular message the window has sent.

To request delivery confirmation for an output message, place values in the following fields of the output header before executing the SEND statement:

- Use the COMPUTE statement to set the Delivery Confirmation Flag field. For example, enter

  ```
  COMPUTE CONFIRMFLAG OF OUTHDR = TRUE
  ```

- Identify a message individually by placing a unique value of your choice into the Delivery Confirmation Key field. When confirming delivery, COMS uses the default input agenda to return the unique value in the first three characters of the message area.

For more information on requesting delivery confirmation on output, see the *COMS Programming Guide*. Refer to “Using Communication Statements” later in this section for information on receiving and sending messages.

Preparing to Receive and Send Messages

Before you can receive or send messages, you must first link to COMS and initialize the program. The following information explains how to perform these tasks.

Linking an Application Program to COMS

To prepare to receive or send messages, you must link your application program to COMS. You do this by specifying access to the data communications interface (DCI) library. The DCI library serves as the programmatic interface with COMS. It enables programs to deal

with symbolic sources and destinations instead of peripherals, and thus avoids recompilation when the peripherals are changed or rearranged.

The compiler builds references to the DCI library whenever a program uses the ACCEPT, DISABLE, ENABLE, RECEIVE, or SEND statements; in fact, the DCI library must be present for a COBOL program to use these verbs. The library reference is built with the title DCILIBRARY and the library entry point name DCIENTRYPOINT. The DCIENTRYPOINT library entry point is an untyped procedure within the DCI library. (DCIENTRYPOINT is also known as a library "object." An object is any item that is declared in a program.)

You can access the DCI library in one of these ways:

- By function name of the COMS code file

  This method uses both the BYFUNCTION value of the LIBACCESS library attribute, and the FUNCTIONNAME library attribute.

- By initiator

  This method uses the BYINITIATOR value of the LIBACCESS library attribute.

- By title

  This method is used by default. It uses the BYTITLE value of the LIBACCESS library attribute and the TITLE library attribute.

It is recommended that you use the BYFUNCTION value of the LIBACCESS attribute to link your programs to COMS. If you do not use either of these methods, the compiler will use the BYTITLE library access method and look for a COMS code file called SYSTEM/COMS.

Once your program is linked, COMS then provides the functions of the DCI library.

The examples in the following discussions show the syntax for each method.

Linking by Function

COBOL allows you to set up your programs to link to the COMS library by function name to avoid modifying your programs every time the file title of the COMS code file changes. You do this by modifying several attributes of the COMS DCI library.

Example 2–3 illustrates the syntax that links to COMS if you have equated the COMS code file to the function name COMSSUPPORT by using the SL (Support Library) system command. However, it will be necessary to modify your program if you change the function name of the COMS code file.

Example

```
CHANGE ATTRIBUTE LIBACCESS OF "DCILIBRARY"
    TO BYFUNCTION.
CHANGE ATTRIBUTE FUNCTIONNAME OF "DCILIBRARY"
    TO "COMSSUPPORT".
```

Example 2–3. Linking a COMS Application Program by Function

The LIBACCESS library attribute statement specifies that the function name, not the object code file title, is used to access the library. The FUNCTIONNAME library attribute statement designates the function name of the library.

Linking by Initiator

COBOL allows you to set up your programs to link to the COMS library by using the BYINITIATOR value of the LIBACCESS attribute. The syntax is illustrated in Example 2–4. If you use the BYINITIATOR file attribute, you will not have to change your programs every time the file title or function name for the COMS DCI library code file changes. However, this linkage method is only valid for programs initiated by COMS.

Example

```
CHANGE ATTRIBUTE LIBACCESS OF "DCILIBRARY"
      TO BYINITIATOR.
```

Example 2–4. Linking a COMS Application Program by Initiator

For additional information on library attributes, refer to the discussion of libraries in Volume 1. Refer to the *System Software Utilities Manual* for a description of the library attributes.

Initializing an Interface Link

To initialize the interface between COMS and your program, include the following statement in the Procedure Division:

```
  ENABLE INPUT COMS-IN KEY "ONLINE".
```

Example

Example 2–5 shows the initialization statement used within the context of code that links an application program to COMS.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 SYSTEM-COMS                 PIC X(50).
.
.
.
PROCEDURE DIVISION.
.
.
.
START-UP-SECTION.
START-UP.
    CHANGE ATTRIBUTE LIBACCESS OF "DCILIBRARY"
      TO BYFUNCTION.
    CHANGE ATTRIBUTE FUNCTIONNAME OF "DCILIBRARY"
      TO "COMSSUPPORT".
    ENABLE INPUT COMS-IN KEY "ONLINE".
```

Example 2–5. Initializing a COMS Interface

Using Communication Statements

You code communication statements in the Procedure Division of the COBOL85 program. The statements include

- ACCEPT MESSAGE COUNT
- DISABLE
- ENABLE
- RECEIVE
- SEND

For general information about using communication statements in a program, refer to the *COMS Programming Guide*.

ACCEPT MESSAGE COUNT Statement

The ACCEPT MESSAGE COUNT statement makes available to the user the number of messages in the application queue of the program.

Format

The general format of the ACCEPT MESSAGE COUNT statement is as follows:

```
ACCEPT  COMS-header-name-1  MESSAGE COUNT
```

Explanation

COMS-header-name-1 is the name of the COMS input header.

The ACCEPT MESSAGE COUNT statement updates the Message Count field to indicate the number of messages present in the queue that COMS maintains for the program.

When the ACCEPT MESSAGE COUNT statement is executed, the Status Value and Message Count fields of the input header are appropriately updated.

Example

Example 2–6 shows an ACCEPT MESSAGE COUNT statement with a COMS-header-name-1 called COMS-IN.

```
.
.
.
COMMUNICATION SECTION.
INPUT HEADER COMS-IN.
OUTPUT HEADER COMS-OUT.
.
.
.
PROCEDURE DIVISION.
.
.
.
    ACCEPT COMS-IN MESSAGE COUNT.
```

Example 2–6. Updating the Message Count Field of the Input Header Message

DISABLE Statement

The DISABLE statement closes a direct window to a station or disconnects a station reached through a modem or a CP2000 station.

For information about dynamically detaching a station, refer to the *COMS Programming Guide*.

Format

```
DISABLE INPUT  [ TERMINAL ]

         COMS-header-name-1  [ WITH KEY  { identifier-1 } ]
                             [           { literal-1    } ]
```

Explanation

COMS-header-name-1 is the name of the COMS input header.

Literal-1 and identifier-1 must be nonnumeric.

For descriptions of the KEY values that COMS provides, refer to the *COMS Programming Guide*.

The DISABLE INPUT phrase provides a logical disconnection between the MCS and the specified sources or destinations. When this logical disconnection has already occurred or is handled by means external to the program, you do not need to include the DISABLE statement in the program. The DISABLE statement does not affect the logical path for the transfer of data between the COBOL85 programs and the MCS.

When the logical disconnection specified by the DISABLE statement has already occurred or is denied by the MCS, the Status Value field data item is updated.

If you use the TERMINAL option with the DISABLE INPUT phrase, only the Station field data item is meaningful.

Example

Example 2–7 shows uses of KEY values with the DISABLE statement.

```
DISABLE INPUT COMS-IN KEY "ONLINE".

DISABLE INPUT COMS-IN KEY "BATCH".

DISABLE INPUT TERMINAL HDR-IN KEY "NOWAIT".

MOVE "WAITNOTBUSY (HOSTNAME = AB10)" TO TEMP.
DISABLE INPUT TERMINAL HDR-IN KEY TEMP.
```

Example 2–7. Using KEY Values with the DISABLE Statement

ENABLE Statement

The ENABLE statement dynamically opens a direct window to a station not currently attached to COMS. You can check the status of station attachment by querying values in the fields of the COMS input header.

For information about the use of this statement to dynamically attach a station, refer to the *COMS Programming Guide*.

Format

```
ENABLE INPUT [ TERMINAL ]

         COMS-header-name-1  [ WITH KEY  { identifier-1 } ]
                             [           { literal-1    } ]
```

Explanation

COMS-header-name-1 is the name of a COMS output header.

Literal-1 or identifier-1 must be nonnumeric.

For descriptions of the KEY values that COMS provides, refer to the *COMS Programming Guide*.

The ENABLE INPUT phrase creates a logical connection between the MCS and the specified sources or destinations. If this logical connection already exists or is handled by means external to the program, you do not need to include the ENABLE statement in the program. The ENABLE statement does not affect the logical path for the transfer of data between the COBOL85 program and the MCS.

The TERMINAL option dynamically opens a direct window to a station that is not attached to COMS.

When the logical connection specified by the ENABLE statement already exists or is denied by the MCS, the Status Value field data item is updated.

Example

Example 2–8 illustrates uses of the KEY values with the ENABLE statement. The first sample can be seen within the context of a complete program, in line 019100 of the COMS sample program in Example 2–24 at the end of this section. For more information on the use of the ENABLE statement to establish communication with the COMS MCS, see "Initializing an Interface Link" earlier in this section.

```
ENABLE INPUT COMS-IN KEY "ONLINE".

ENABLE INPUT COMS-IN KEY "BATCH".

ENABLE INPUT TERMINAL HDR-IN KEY "NOWAIT".

MOVE "WAITNOTBUSY (HOSTNAME = AB10)" TO TEMP.
ENABLE INPUT TERMINAL HDR-IN KEY TEMP.
```

Example 2–8. Using KEY Values with the ENABLE Statement

RECEIVE Statement

The RECEIVE statement makes a message and pertinent information about the data available to the COBOL85 program from a queue maintained by the MCS. You can use the RECEIVE statement to execute a specific imperative-statement when you use the NO DATA syntax, as shown in the following format.

You can use the RECEIVE statement as many times as needed in the Procedure Division of your program. You can structure your program to receive messages from one or more stations or programs, but you cannot programmatically limit the reception of messages to selected stations on the network or to certain types of programs.

Before you can receive or send messages, however, you must link to COMS and initialize the program. For information, refer to "Preparing to Receive and Send Messages" earlier in this section.

Caution

Do not use a RECEIVE statement between a BEGIN-TRANSACTION statement and an END-TRANSACTION statement. Doing so violates the rules of synchronized recovery and you might lose some of the data in your database.

Format

```
RECEIVE  COMS-header-name-1     MESSAGE INTO identifier-1
                              [ NO DATA  imperative-statement ]
                              [ END-RECEIVE ]
```

Explanation

COMS-header-name-1 is the name of a COMS input header.

The message is transferred to the receiving character positions of the area referenced by identifier-1 and is aligned to the left without space fill.

If you specify the NO DATA phrase and the MCS makes data available in identifier-1, COMS transfers control to the next executable statement when you execute a RECEIVE statement.

An imperative-statement can consist of a NEXT SENTENCE phrase.

If the MCS does not make data available in the identifier-1 data item, one of the following actions takes place when you execute a RECEIVE statement:

- If you specify the NO DATA phrase, the RECEIVE statement is terminated, indicating that action is complete; the imperative-statement is executed.
- If you do not specify the NO DATA phrase, then the object program execution is suspended until data is made available in identifier-1 or until end-of-task (EOT) notification.

The following rules apply to the data transfer:

If the message size is . . .	**Then the message . . .**
The same size as identifier-1	Is stored in identifier-1.
Smaller than identifier-1	Is aligned to the leftmost character position of identifier-1 with no space fill.
Larger than identifier-1	Fills identifier-1 from left to right, starting with the leftmost character of the message. The rest of the message is truncated.

Each time the RECEIVE statement executes, the MCS appropriately updates the data items identified by the COMS input header.

Example

Example 2–9 shows part of an application routine that was written to receive a message from COMS and place it in the Working-Storage Section. An example of the RECEIVE statement within the context of a complete program is provided in the COMS sample program in Example 2–24, beginning at line 019700.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
01 MSG-IN-TEXT              PIC X(1920).
.
.
.
COMMUNICATION SECTION.
INPUT HEADER COMS-IN.
OUTPUT HEADER COMS-OUT.
.
.
.
PROCEDURE DIVISION.
.
.
.
RECEIVE-MSG-FROM-COMS.
    RECEIVE COMS-IN MESSAGE INTO MSG-IN-TEXT.
      IF STATUSVALUE OF COMS-IN = 99
         GO TO EOJ-ROUTINE.
*      Process the message.
      .
      .
      .
      GO-TO-RECEIVE-MSG-FROM-COMS.
EOJ-ROUTINE.
     STOP RUN.
```

Example 2–9. Placing a COMS Message in the Working-Storage Section

SEND Statement

The SEND statement releases a message, message segment, or portion of a message to one or more output queues maintained by the MCS. Before you can send or receive messages, however, you must link to COMS and initialize the program. For information on these functions, refer to "Preparing to Receive and Send Messages" earlier in this section.

Caution

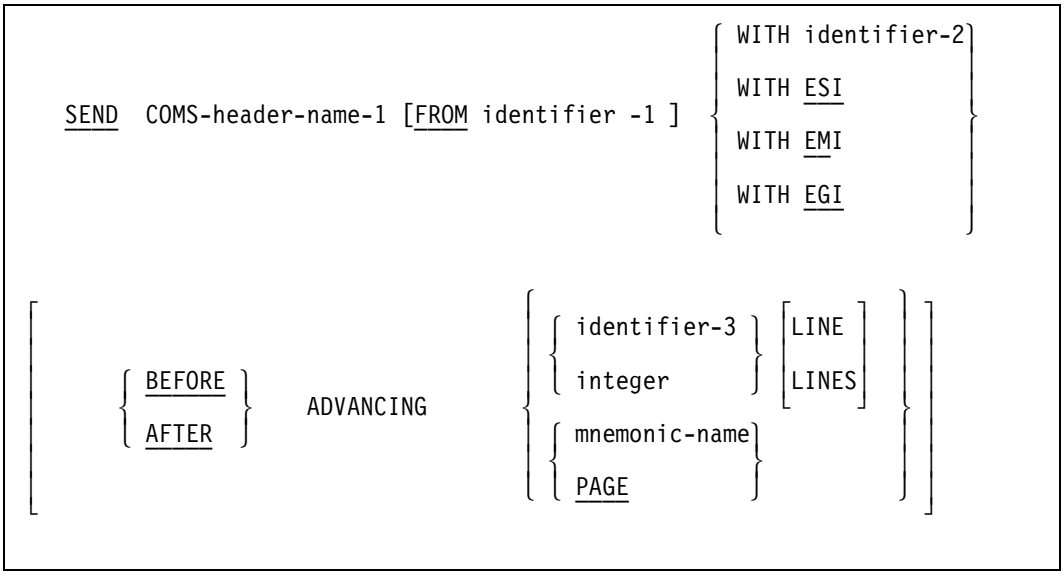
Do not use a SEND statement between a BEGIN-TRANSACTION statement and an END-TRANSACTION statement. Doing so violates the rules of synchronized recovery and you might lose some of the data in your database.

There are two formats for the SEND statement. Format 1 is for nonsegmented output only, and Format 2 is for nonsegmented or segmented output.

Format 1 - Nonsegmented Output Only

```
SEND  COMS-header-name-1  FROM identifier-1
```

Format 2 - Segmented or Nonsegmented Output



Explanation for Format 1 - Nonsegmented Output Only

Format 1 is for nonsegmented output only.

COMS-header-name-1 is the name of a COMS output header.

Identifier-1 is the data-name of the area where the MCS looks for data to be sent.

The message or message segment is moved to the sending character positions of the area of identifier-1 and aligned to the left with space fill.

When you execute a SEND statement, the MCS interprets the value in the Text Length field of the output header as the number of leftmost character positions of identifier-1 from which data is to be transferred.

If the value of Text Length is 0 (zero), no characters of identifier-1 are transferred. The value of Text Length cannot be outside the range 0 through the size of identifier-1. If the value is outside the range, the message is truncated to the size of identifier-1 and the Status Value field of the output header is set to 0.

When a SEND statement executes, the MCS updates the Status Value field.

The effect of special control characters within identifier-1 is undefined.

Explanation for Format 2 - Segmented or Nonsegmented Output

Format 2 allows either segmented or nonsegmented output. You can use the WITH option of the SEND statement to select nonsegmented output or a type of segmented output. The ESI, EMI, and EGI options are for segmentation. These options are explained in the following subsection.

Identifier-2 must reference a 1-character integer without an operational sign. For example,

```
PIC S9(11) USAGE BINARY
```

If identifier-2 is 0 (zero), it indicates nonsegmented output.

Identifier-3, if used, must be the name of an elementary integer item.

Integer, and the value of identifier-3, can be 0 (zero).

If you use a mnemonic-name phrase, it is identified with a particular feature specified in the SPECIAL-NAMES paragraph in the Environment Division.

Segmenting Options

There are three segmenting options: the end-of-segment indicator (ESI), the end-of-message indicator (EMI), and the end-of-group indicator (EGI). COMS recognizes these indicators and establishes the appropriate linkage to maintain control over groups, messages, and segments. For example, the following statement sends output immediately after values have been moved to the required fields of the output header:

```
SEND <output header name> FROM <message area> WITH EMI.
```

The contents of identifier-2 indicate that the contents of identifier-1 are to have an associated ESI, EMI, or EGI according to the schedule in Table 2–4.

Table 2–4. Transmission Indicator Schedule

Identifier-2	Identifier-1	Explanation
0	No indicator	No indicator
1	ESI	Message segment complete
2	EMI	Message complete
3	EGI	Group of messages complete

Any character other than 1, 2, or 3 is interpreted as 0 (zero). If identifier-2 is a number other than 1, 2, or 3 and if identifier-1 is not specified, the data is transferred and no error is indicated.

The hierarchy of ending indicators (major to minor) is EGI, EMI, and ESI. An EGI need not be preceded by an ESI or an EMI, and an EMI need not be preceded by an ESI.

A single execution of a Format 2 SEND statement never releases to the MCS more than the single message or single message segment that is indicated by the contents of the data item referenced by identifier-2 or by the specified ESI, EMI, or EGI. However, the MCS does not transmit any portion of a message to a communications device until the entire message is placed in the output queue.

During the execution of the run unit, the disposition of a portion of a message not terminated by an EMI or EGI is undefined. Thus, the message does not logically exist for the MCS and cannot be sent to a destination.

After the execution of a STOP RUN statement, the system removes any portion of a message transferred from the run unit as a result of a SEND statement, but not terminated by an EMI or EGI. Thus, no portion of the message is sent.

Advancing Options

The ADVANCING phrase enables you to control the vertical positioning of each message or message segment on an output device where this control is possible. If vertical positioning is not applicable on the device, COMS ignores the specified or implied vertical positioning.

On a device where vertical positioning applies and the ADVANCING phrase is not specified, automatic advancing occurs as if you had specified BEFORE ADVANCING 1 LINE.

You can use the BEFORE ADVANCING and AFTER ADVANCING options to specify whether the text of an output message should be written to the output device before or after the device advances to the next page, or before or after the device advances a specified number of lines. If you specify neither of these options, it is assumed that you are specifying lines. For example, the following code instructs the device to write a message after advancing one line:

```
SEND COMS-OUT FROM MSG-OUT-TEXT AFTER ADVANCING 1.
```

Although COMS supplies a default setting for carriage control, a processing item can alter carriage control before an output message reaches its destination. For instructions about this procedure, refer to the *COMS Programming Guide*.

If you specify identifier-3 and identifier-3 is 0 (zero), the MCS ignores the ADVANCING phrase.

If you implicitly or explicitly specify the ADVANCING phrase to a device on which you can control the vertical positioning, the following rules apply:

- If you use the BEFORE ADVANCING phrase, the output device writes the message or message segment before it repositions the message vertically according to the rules for identifier-3, integer, and mnemonic-name.

  In the following SEND statement, the AFTER ADVANCING phrase instructs the output device to advance before values have been moved into the appropriate fields of the output headers:

  ```
  SEND COMS-OUT FROM COMS-OUT-AREA WITH EMI
      BEFORE ADVANCING 2 LINES.
  ```

- If you use the AFTER ADVANCING phrase, the output device writes the message or message segment after it repositions the message vertically according to the rules for identifier-3, integer, and mnemonic-name. These rules are
  - If you specify identifier-3 or integer, the output device repositions characters vertically downward by a number of lines equal to the value of identifier-3 or integer.
  - If you specify a mnemonic-name, the output device positions characters according to the rules for that device.

- If you specify PAGE, the output device writes characters either before or after the device is repositioned to the next page (depending on the phrase used). For example, in the following SEND statement, the BEFORE ADVANCING phrase instructs the output device to advance after values have been moved into the appropriate fields of the output header:

```
SEND COMS-OUT FROM COMS-OUT-AREA WITH EMI
     AFTER ADVANCING PAGE.
```

  If you specify PAGE, but PAGE has no meaning in conjunction with a specific device, then the output device advances as if you had specified a BEFORE ADVANCING 1 LINE or AFTER ADVANCING 1 LINE phrase.

Example

Example 2–10 shows SEND statements that specify segmented output using the WITH ESI and WITH EGI options. The options are specified to hold output until the message is complete.

```
.
.
.
WORKING-STORAGE SECTION.
     01 MESSAGE-1  PIC X(100).
     01 MESSAGE-2  PIC X(100).
     01 MESSAGE-3  PIC X(100).
.
.
.
COMMUNICATION SECTION.
INPUT HEADER INHDR.
OUTPUT HEADER OUTHDR.
.
.
.
PROCEDURE DIVISION.
.
.
.
    MOVE OUTPUT-SIZE1 TO TEXTLENGTH OF OUTHDR.
    MOVE 0 TO STATUSVALUE OF OUTHDR.
    SEND OUTHDR FROM MESSAGE-1 WITH ESI.

    MOVE OUTPUT-SIZE2 TO TEXTLENGTH OF OUTHDR.
    MOVE 0 TO STATUSVALUE OF OUTHDR.
    SEND OUTHDR FROM MESSAGE-2 WITH ESI.

    MOVE OUTPUT-SIZE3 TO TEXTLENGTH OF OUTHDR.
    MOVE 1 TO DESTCOUNT.
    MOVE STATIONDESG OF INHDR TO DESTINATIONDESG OF OUTHDR.
    MOVE 0 TO STATUSVALUE OF OUTHDR.
    SEND OUTHDR FROM MESSAGE-3 WITH EGI.
```

Example 2–10. Using SEND Statements with ESI and EGI Options

Using Service Functions

This discussion of service functions includes explanations of the following:

- Using COMS designators
- Identifying information with service function mnemonics
- Calling service functions
- Passing parameters to service functions

This discussion also includes an explanation of each service function. COMS uses the following service functions:

- CONVERT_TIMESTAMP
- GET_DESIGNATOR_ARRAY_USING_DESIGNATOR
- GET_DESIGNATOR_USING_DESIGNATOR
- GET_DESIGNATOR_USING_NAME
- GET_INTEGER_ARRAY_USING_DESIGNATOR
- GET_INTEGER_USING_DESIGNATOR
- GET_NAME_USING_DESIGNATOR
- GET_REAL_ARRAY
- GET_STRING_USING_DESIGNATOR
- STATION_TABLE_ADD
- STATION_TABLE_INITIALIZE
- STATION_TABLE_SEARCH
- TEST_DESIGNATORS

In COBOL85, you can use hyphens (-) rather than underscores (_) in the names of service functions. The compiler automatically translates hyphens to underscores for use with COMS.

For a complete discussion of the COMS service functions and their use, refer to the *COMS Programming Guide*.

Using COMS Designators

Service functions use numeric designators that are part of an internal code understood by COMS. You obtain designators from COMS headers or from service functions that allow you to translate names to designators. Refer to “Mapping COMS Data Types to COBOL85,” earlier in this section, for information on the COMS designator data type and its use with COBOL85. See the *COMS Programming Guide* for information about COMS designators.

Identifying Information with Service Function Mnemonics

The COMS entities have designators that can be used in service function calls. Table 2–5 lists the service function mnemonics that you can use to identify particular data items. The data items indicate the kinds of information that can be requested in a program.

Table 2–5. Service Functions Mnemonics

Data Item	Mnemonic
Agenda	AGENDA
Database	DATABASE
Device designator	DEVICE
Device list	DEVICE-LIST
Host Name	HOST-NAME
Installation data	INSTALLATION-DATA INSTALLATION-DATA-LINK INSTALLATION-STRING-1 INSTALLATION-STRING-2 INSTALLATION-STRING-3 INSTALLATION-STRING-4 INSTALLATION-HEX-1 INSTALLATION-HEX-2 INSTALLATION-INTEGER-ALL INSTALLATION-INTEGER-1 INSTALLATION-INTEGER-2 INSTALLATION-INTEGER-3 INSTALLATION-INTEGER-4
Library	LIBRARY
Message date in format MMDDYY	DATE
Message time in format HHMMSS	TIME
Processing item	PROCESSING-ITEM
Processing item list	PROCESSING-ITEM-LIST
Program	PROGRAM

Table 2–5. Service Functions Mnemonics

Data Item	Mnemonic
Program: current input queue depth	QUEUE-DEPTH
Program: mix numbers for active copies	MIXNUMBERS
Program: response time for last transaction	LAST-RESPONSE
Program: aggregate response time	AGGREGATE-RESPONSE
Program: security designator	SECURITY
Program: total number of input messages handled	MESSAGE-COUNT
Security category	SECURITY-CATEGORY
Security category list	CATEGORY-LIST
Station	STATION
Station list	STATION-LIST
Station: logical station number (LSN)	LSN
Station: screen size	SCREENSZ
Station: session number	SESSION
Station: security designator	SECURITY
Station: virtual terminal	VIRTUAL-TERMINAL VIRTTERM
Statistics	STATISTICS
Transaction code	TRANCODE
Window	WINDOW
Window: maximum number of users	MAXIMUM-USER-COUNT
Window: current number of users	CURRENT-USER-COUNT
Window list	WINDOW-LIST
Usercode	USERCODE

Calling Service Functions

You can call the COMS service functions with application programs and processing items. When you call a service function, do the following:

- Use the CALL statement syntax for calling library procedures in COBOL85.
- Pass the integer parameters (using unscaled integer values) by name rather than by value.

Using the CALL statement

Format

```
CALL  literal-1
          USING  { identifier-1 } . . .
          GIVING   identifier-2
          [ END-CALL ]
```

Explanation

Literal-1 is a nonnumeric literal. It contains the service function name, followed by the qualifying library name. The only name allowed is DCILIBRARY.

Identifier-1 is a set of input and output parameters passed to the procedure described in literal-1. The passing of parameters by value is described later in this section.

Identifier-2 is the result of a service function call.

The END-CALL phrase delimits the scope of the CALL statement.

Do not perform arithmetic computations on the values returned from the procedure calls. You can move the values to other locations of a compatible type within your program, and pass them as parameters when calling other library objects. For information on compatible types, refer to “Mapping COMS Data Types to COBOL85” earlier in this section.

Example

Example 2–11 shows a use of the CALL statement to pass an agenda designator to obtain an agenda name. You must also declare the agenda designator and agenda name used in the example in the Working-Storage Section of the program.

The service function result value indicates the result of the service function call. The result value is returned to the parameter specified in the GIVING clause of the CALL statement. Refer to the *COMS Programming Guide* for information about the service function result values. Refer to “Passing Parameters to Service Functions,” later in this section, for information on valid parameters and examples of the CALL syntax.

For more information on the GET_NAME_USING_DESIGNATOR service function, and for an example of program code used for the CALL statement with this service function, see "GET_NAME_USING_DESIGNATOR Service Function" later in this section.

```
CALL "GET-NAME-USING-DESIGNATOR OF DCILIBRARY"
        USING <agenda designator>,
              <agenda name>
        GIVING <service function result value>
```

Example 2–11. Calling a COMS Service Function with a CALL Statement

Using Parameters by Value

The VALUE clause is used with the CALL statement for COMS service functions in which a mnemonic is passed for a numeric result.

Format

```
[ VALUE mnemonic-1 ]
```

Explanation

The keyword VALUE indicates that a service function mnemonic is being passed by value to the service function. Mnemonic-1 is the mnemonic parameter. Refer to "Identifying Information with Service Function Mnemonics," earlier in this section, for information on valid mnemonics.

Note: *If the library being called is DCILIBRARY, you must use hyphens (-) in the mnemonic names. The hyphens are automatically translated to underscores (_) by the compiler for use with COMS.*

Example

Example 2–12 shows an example of using the CALL statement with the VALUE parameter. The LSN-NUM variable contains the value of the LSN service function mnemonic. SF-RSLT receives the result of the procedure call. If the result of the procedure call is successful, SF-RSLT contains a 0 (zero); otherwise, SF-RSLT receives an error code that the user accesses in another part of the program.

Note that the GET_INTEGER_USING_DESIGNATOR service function has hyphens (-) between the words in its name because the DCILIBRARY library is the library called.

For information on valid parameters and an example of program code using the GET_INTEGER_USING_DESIGNATOR service function, see "GET_INTEGER_USING_DESIGNATOR Service Function" later in this section.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 HDR-STATION       REAL.
77 LSN-NUM           PIC S9(11) USAGE BINARY.
77 SF-RSLT           PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
CALL "GET-INTEGER-USING-DESIGNATOR OF DCILIBRARY"
          USING HDR-STATION
                VALUE LSN
                LSN-NUM
          GIVING SF-RSLT.
```

Example 2–12. Using the VALUE Parameter When Calling a Service Function

Passing Parameters to Service Functions

The following parameters can be passed to service functions:

- Designators
- Mnemonics or real values
- Arrays

For general information about passing service function parameters, refer to the *COMS Programming Guide*. For information on service function mnemonics used in COBOL85, refer to "Identifying Information with Service Function Mnemonics" earlier in this section.

You must declare all parameters to be passed, except mnemonics, in the Working-Storage Section of your COBOL85 program.

You cannot pass header fields as parameters. To pass fields, you must first move them to a declared temporary parameter, and then pass the values. The temporary parameter must be declared as a real value.

The service function declarations and valid parameters for passing service functions in a COBOL85 program are explained in the following discussion. Note the following general characteristics:

- An entity name is a data item with DISPLAY usage.
- A mnemonic is a COMS mnemonic representing an entity or a designator.
- An array is an EBCDIC or integer array.
- A designator is a data item of type real. Refer to the coded examples of declarations of designator, integer, and real tables that are provided at the end of the discussion of each service function.

CONVERT_TIMESTAMP Service Function

The CONVERT_TIMESTAMP service function converts the COMS timestamp TIME(6) to the date or time in EBCDIC arrays. For information on the COMS TIME(6) type and COBOL85 usage, refer to “Mapping COMS Data Types to COBOL85” earlier in this section.

The input parameter is a real value that represents the timestamp.

The allowable mnemonics represent the requested function. They include

- DATE, which returns MMDDYY
- TIME, which returns HHMMSS

The result parameter is an EBCDIC array in which the time or date is returned.

Example

Example 2–13 shows an example of coding for the CONVERT_TIMESTAMP service function.

```
.
.
.
 WORKING-STORAGE SECTION.
.
.
.
77 WS-TIMESTAMP                  REAL.
01 WS-TIME                       PIC X(6).
77 SF-RSLT                       PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
   CALL "CONVERT-TIMESTAMP OF DCILIBRARY"
        USING WS-TIMESTAMP
              VALUE TIME
              WS-TIME
        GIVING SF-RSLT.
```

Example 2–13. Using the CONVERT_TIMESTAMP Service Function

GET_DESIGNATOR_ARRAY_USING_DESIGNATOR Service Function

The GET_DESIGNATOR_ARRAY_USING_DESIGNATOR service function obtains a designator array from the structure represented by the designator.

The allowable input parameter is the STATION LIST designator that represents the structure. STATION LIST returns an array with stations.

The result parameter is an integer that represents the number of designators returned in the array. The array is a real array containing the designators.

Example

Example 2–14 shows an example of coding for the GET_DESIGNATOR_ARRAY_USING_DESIGNATOR service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 WS-TABLE-DESG                    REAL.
77 WS-DESG-TABLE-MAX-INDEX          PIC S9(11) USAGE BINARY.
01 WS-DESG-TABLE                    REAL.
   05 WS-D-TABLE                    REAL OCCURS 10 TIMES.

77 SF-RSLT                          PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
   CALL "GET-DESIGNATOR-ARRAY-USING-DESIGNATOR OF DCILIBRARY"
        USING WS-TABLE-DESG
              WS-DESG-TABLE-MAX-INDEX
              WS-DESG-TABLE
        GIVING SF-RSLT.
```

Example 2–14. Using the GET_DESIGNATOR_ARRAY_USING_DESIGNATOR Service Function

GET_DESIGNATOR_USING_DESIGNATOR Service Function

The GET_DESIGNATOR_USING_DESIGNATOR service function obtains a specific designator from the structure represented by the designator.

The input parameter is a designator that represents the structure.

The mnemonic for the input parameter describes the requested integer array. The mnemonics allowed for the designators are

Designator	Mnemonic
All	INSTALLATION_DATA_LINK
Station, usercode, program	SECURITY
Station	DEVICE

The result parameter is a designator.

Example

Example 2–15 provides an example of coding for the GET_DESIGNATOR_USING_DESIGNATOR service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 WS-DESG                    REAL.
77 WS-DESG-RSLT               REAL.
77 SF-RSLT                    PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
CALL "GET-DESIGNATOR-USING-DESIGNATOR OF DCILIBRARY"
       USING WS-DESG
             VALUE INSTALLATION-DATA-LINK
             WS-DESG-RSLT
       GIVING SF-RSLT.
```

Example 2–15. Using the GET_DESIGNATOR_USING_DESIGNATOR Service Function

GET_DESIGNATOR_USING_NAME Service Function

The GET_DESIGNATOR_USING_NAME service function converts the COMS name variable into a designator.

The input parameter is an entity name. The string for the entity name of agenda, transaction code (trancode), and installation data includes the window name if the program calling the service function is running in another window or is outside of COMS. For example, the following input passes the entity name:

```
<agenda name> of <window name>
```

For installation data, the installation data with a window value equal to the ALL entity (the default value) is used if no window is specified, and if the window in which the program is running does not have an entity of the same name.

The mnemonic for the input parameter is the entity type for the required name.

The result parameter is a designator.

Example

Example 2–16 shows an example of coding for the GET_DESIGNATOR_USING_NAME service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
01 WS-NAME                      PIC X(30).
77 WS-DESG                      REAL.
77 SF-RSLT                      PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
   CALL "GET-DESIGNATOR-USING-NAME OF DCILIBRARY"
        USING WS-NAME
              VALUE STATION-LIST
              WS-DESG
        GIVING SF-RSLT.
```

Example 2–16. Using the GET_DESIGNATOR_USING_NAME Service Function

GET_ERRORTEXT_USING_NUMBER Service Function

The GET_ERRORTEXT_USING_NUMBER service function converts an XATMI function error code into text representing an error message.

The input parameter is an integer representing an XATMI function result value.

The output parameters are the following:

- An integer representing the length of the text string returned
- A text string representing an error message

If the error text is available in multiple languages, the language attribute of the calling program determines the language in which the error message is returned.

Example

Example 2–17 shows coding for the GET_ERRORTEXT_USING_NUMBER service. SF-RSLT receives the result of the call to the service function. If the result of the call is successful, SF-RSLT contains a 0 (zero); otherwise, SF-RSLT contains an error code.

```
.
.
.
WORKING-STORAGE SECTION.
.
77 WS-ERROR                              PIC S9(11)  USAGE BINARY.
77 WS-TEXT-LENGTH                        PIC S9(11)  USAGE BINARY
01 WS-ERROR-TEXT                         PIC X(90).
77 SF-RSLT                               PIC S9(11)  USAGE BINARY
.
.
PROCEDURE DIVISION
.
.
   CALL "GET-ERRORTEXT-USING-NUMBER OF DCILIBRARY"
      USING WS-ERROR
            WS-TEXT-LENGTH
            WS-ERROR-TEXT
      GIVING SF-RSLT.
```

Example 2–17. Using the GET_ERRORTEXT_USING_NUMBER Service Function

GET_INTEGER_ARRAY_USING_DESIGNATOR Service Function

The GET_INTEGER_ARRAY_USING_DESIGNATOR service function obtains an array of integers from the structure represented by the designator.

The input parameter is a designator that represents the structure.

The mnemonic for the input parameter describes the requested integer array. The mnemonics allowed for designators are

Designator	Mnemonic
All	INSTALLATION_INTEGER_ALL
Program	MIXNUMBERS

The result parameters are the following:

- An integer representing the number of integers returned in the array
- An integer array containing the returned information

Example

Example 2–18 shows an example of coding for the GET_INTEGER_ARRAY_USING_DESIGNATOR service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 WS-DESG                    REAL.
77 WS-INT-TABLE-MAX-INDEX     PIC S9(11) USAGE BINARY.
77 SF-RSLT                    PIC S9(11) USAGE BINARY.
01 WS-INT-TABLE               USAGE BINARY.
   03 WS-INT-TABLE-DETAIL     PIC S9(11) OCCURS 10 TIMES.
.
.
.
PROCEDURE DIVISION.
.
.
.
   CALL "GET-INTEGER-ARRAY-USING-DESIGNATOR OF DCILIBRARY"
        USING WS-DESG
              VALUE INSTALLATION-INTEGER-ALL
              WS-INT-TABLE-MAX-INDEX
              WS-INT-TABLE
        GIVING SF-RSLT.
```

Example 2–18. Using the GET_INTEGER_ARRAY_USING_DESIGNATOR Service Function

GET_INTEGER_USING_DESIGNATOR Service Function

The GET_INTEGER_USING_DESIGNATOR service function obtains a specific integer from the structure represented by the designator.

The input parameter is a designator representing the structure.

The mnemonic for the input parameter describes the requested integer. Allowable mnemonics and designators are

Designator	Mnemonic
All	INSTALLATION_INTEGER_1, 2, 3, 4
Station	LSN, VIRTTERM, SCREENSZ, SESSION
Window	MAXIMUM_USER_COUNT and CURRENT_USER_COUNT
Program	QUEUE_DEPTH, MESSAGE_COUNT, LAST_RESPONSE, and AGGREGATE_RESPONSE

The result parameter is an integer.

Example

Example 2–19 provides an example of coding for the GET_INTEGER_USING_DESIGNATOR service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 WS-DESG                          REAL.
77 WS-INT                           PIC S9(11) USAGE BINARY.
77 SF-RSLT                          PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
   CALL "GET-INTEGER-USING-DESIGNATOR OF DCILIBRARY"
        USING WS-DESG
              VALUE INSTALLATION-INTEGER-1
              WS-INT
        GIVING SF-RSLT.
```

Example 2–19. Using the GET_INTEGER_USING_DESIGNATOR Service Function

GET_NAME_USING_DESIGNATOR Service Function

The GET_NAME_USING_DESIGNATOR service function converts a COMS designator to the COMS designator name.

The input parameter is a designator. All designators are allowed.

The result parameter is an entity name.

Example

Example 2–20 shows an example of coding for the GET_NAME_USING_DESIGNATOR service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 WS-DESG                   REAL.
01 WS-NAME                   PIC X(30).
77 SF-RSLT                   PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
   CALL "GET-NAME-USING-DESIGNATOR OF DCILIBRARY"
        USING WS-DESG
              WS-NAME
        GIVING SF-RSLT.
```

Example 2–20. Using the GET_NAME_USING_DESIGNATOR Service Function

GET_REAL_ARRAY Service Function

The GET_REAL_ARRAY service function obtains a structure of data with no connection to any entity.

The input parameter is a mnemonic representing the requested function. The only allowable mnemonic is STATISTICS, which returns a table. Each table entry has the following six input parameter items. (The input parameter items that show DCI library in parentheses are passed from DCI library programs only, not from remote files.)

- Entity designator
- Type of entity:

Entry Code	Entity Represented
1	DCI library program
2	Remote file interface
3	MCS window

- Queue depth (DCI library)
- Number of transactions
- Last transaction response in milliseconds (DCI library)
- Aggregate response in milliseconds (DCI library)

The result parameters are the following:

- An integer representing the total number of elements returned in the array
- An array containing the information returned

Example

Example 2–21 shows an example of coding for the GET_REAL_ARRAY service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 WS-REAL-TABLE-MAX-INDEX         PIC S9(11) USAGE BINARY.
01 WS-REAL-TABLE                   REAL.
   05 WS-R-TABLE                   REAL OCCURS 500 TIMES.
77 SF-RSLT                         PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
   CALL "GET-REAL-ARRAY OF DCILIBRARY"
        USING VALUE STATISTICS
              WS-REAL-TABLE-MAX-INDEX
               WS-REAL-TABLE
         GIVING SF-RSLT.
```

Example 2–21. Using the GET_REAL_ARRAY Service Function

GET_STRING_USING_DESIGNATOR Service Function

The GET_STRING_USING_DESIGNATOR service function obtains an EBCDIC string from the structure represented by the designator.

The input parameter is an entity designator that represents the structure.

The entity mnemonic describes the requested integer vector. All designators are allowable. The allowable mnemonics are INSTALLATION_STRING_1, 2, 3, 4 and INSTALLATION_HEX_1, 2.

The result parameters are the following:

- An integer that indicates the number of valid characters in the string
- An EBCDIC array indicating the returned string

Example

Example 2–22 shows an example of coding for the GET_STRING_USING_DESIGNATOR service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77  WS-DESG                  REAL.
77  WS-STRING-MAX-CHAR       PIC S9(11) USAGE BINARY.
01  WS-STRING                PIC X(30).
77  SF-RSLT                  PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
      MOVE SPACES TO WS-STRING.
      CALL "GET-STRING-USING-DESIGNATOR OF DCILIBRARY"
           USING WS-DESG
                 VALUE INSTALLATION-STRING-1
                 WS-STRING-MAX-CHAR
                 WS-STRING
           GIVING SF-RSLT.
```

Example 2–22. Using the GET_STRING_USING_DESIGNATOR Service Function

STATION_TABLE_ADD Service Function

The STATION_TABLE_ADD service functions allows you to add a station designator to the list (table) of stations. The table is controlled by the transaction processor.

The input parameters are as follows:

- A real array that contains the table of station designators
- A station designator that is to be added to the table

The output parameter is an index value representing the location of the station in the station table.

An example of the use of this service function is provided in Example 2–22, at the end of the explanation of the STATION_TABLE_SEARCH service function.

STATION_TABLE_INITIALIZE Service Function

The STATION_TABLE_INITIALIZE service function initializes the table into which the station index values are placed. The table is implemented as a hash table. It is recommended that the size of the table is based on the number of stations that may be added to the table (or the number of stations that exist).

The input parameters are as follows:

- A real table of station designators
- A table modulus

The table modulus is used to determine the density and access time of the table. If, for example, you have a table with a sparse population and you desire very fast access time, select a modulus that is twice the maximum number of table entries. If the table has a compact population and slower access is acceptable, use a modulus with half the maximum number of entries.

An example of the use of this service function is provided in Example 2–22, at the end of the explanation of the STATION_TABLE_SEARCH service function.

STATION_TABLE_SEARCH Service Function

The STATION_TABLE_SEARCH service function searches through a table and locates a specific station designator.

The input parameters are as follows:

- The name of the real table of designators to be searched
- The name of the station designator to be found

The output parameter is the index value at the point in the table that the designator was found. If the station designator was not found, a value of 0 (zero) is returned.

Example

Example 2–23 shows the declarations and statements for the station table service functions. After the execution of the code in the example, STATION-SEARCH-RESULT will contain the index of the station designator in the table.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
01 STATION-TABLE                REAL.
   02 STATION-DESIGNATOR        REAL OCCURS 100.
77 STATION-SEARCH-RESULT        PIC S9(11) BINARY.
77 STATION-SEARCH-DESIGNATOR    REAL.
77 STATION-SEARCH-MODULUS       PIC S9(11) BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
    CALL "STATION-TABLE-INITIALIZE OF DCILIBRARY"
          USING STATION-TABLE, STATION-SEARCH-MODULUS.
    MOVE STATION OF COMS-IN TO STATION-SEARCH-DESIGNATOR.
    CALL "STATION-TABLE-SEARCH OF DCILIBRARY"
          USING STATION-TABLE, STATION-SEARCH-DESIGNATOR
          GIVING STATION-SEARCH-RESULT.
    IF STATION-SEARCH-RESULT IS EQUAL TO 0
        CALL "STATION-TABLE-ADD OF DCILIBRARY"
              USING STATION-TABLE, STATION-SEARCH-DESIGNATOR
              GIVING STATION-RESULT.
```

Example 2–23. Using the STATION TABLE Service Functions

TEST_DESIGNATORS Service Function

The TEST_DESIGNATORS service function tests whether a designator is part of a structure represented by another designator.

The input parameters are two designators representing structures. The order of the designators does not matter. Allowable designator combinations include an array of designators using either the mnemonics DEVICE and DEVICE_LIST or the mnemonics SECURITY and SECURITY_CATEGORY.

Example

Example 2–24 shows an example of coding for the TEST_DESIGNATORS service function.

```
.
.
.
WORKING-STORAGE SECTION.
.
.
.
77 WS-DESG                          REAL.
77 WS-DESG-RSLT                     REAL.
77 SF-RSLT                          PIC S9(11) USAGE BINARY.
.
.
.
PROCEDURE DIVISION.
.
.
.
    CALL "TEST-DESIGNATORS-OF-DCILIBRARY"
         USING WS-DESG
               WS-DESG-RSLT
         GIVING SF-RSLT.
```

Example 2–24. Using the TEST_DESIGNATORS Service Functions

COMS Sample Program with a DMSII Database

This sample program, called SAILOLPROG, tracks sailboat races and updates a DMSII database by using features of the COMS direct-window interface. The program exemplifies the programming techniques used in writing transaction processors that allow synchronized recovery.

The SAILOLPROG program maintains the SAILDB database. The program contains three transactions. Each transaction has a unique trancode and a unique module function index (MFI). The CRERAC trancode creates a race entry in the database. ADDENT adds a boat entry to a race. The race must exist for the transaction to be completed. DELENT deletes a boat from a race.

COMS Features Used in the Sample Program

The following features of the COMS direct-window interface are used in the program:

- Declared COMS input and output headers
- Trancodes
- Synchronized recovery

Information on synchronized recovery when COMS is used with DMSII is provided in the explanation of the END-CLOSE phrase under “CLOSE Statement” in Section 3.

The SAILOLPROG program runs in a COMS environment that has been configured to include a DMSII database called SAILDB, and the following COMS entities:

- A direct window called SAIL
- An agenda called SAILAGINOL
- The following three trancodes:
  - CRERAC(MFI=1)
  - ADDENT(MFI=2)
  - DELENT(MFI=3)

All entities must be defined to COMS to allow the program to run.

Data Sets in the Database

The database SAILDB contains three data sets. The data set RACE-CALENDAR contains one record for every race. The data set 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. The data set RDS is the restart data set (RDS) with the appropriate COMS required fields. The DMSII option RDSSTORE is not set for the database.

Using the Sample Program

The SAILOLPROG program is shown in Example 2–25. This representation of the program contains comment lines to indicate what the program is doing at each step.

All transactions in the program are two-phase transactions. In phase 1, all records are locked. In phase 2, the data is stored in the database and only the END-TRANSACTION statement unlocks records. All transactions instruct COMS to audit the input message when the END-TRANSACTION statement takes the program out of transaction state

```
010200 IDENTIFICATION DIVISION.
010300 PROGRAM-ID. ONLINESAIL.
010400 ENVIRONMENT DIVISION.
010500 CONFIGURATION SECTION.
010600     SOURCE-COMPUTER. Micro A.
010700     OBJECT-COMPUTER. Micro A.
010800 INPUT-OUTPUT SECTION.
010900 FILE-CONTROL.
011000    SELECT RMT ASSIGN TO REMOTE ACCESS SEQUENTIAL.
011100 DATA DIVISION.
011200 FILE SECTION.
011300 FD RMT.
011400 01 REM-REC                    PIC X(72).
011500 DATA-BASE SECTION.
011600     DB SAILDB  ALL.
011700 WORKING-STORAGE SECTION.
011800 77  SCRATCH                   PIC X(256).
011900
012000*****************************************************************
012100*  MESSAGE AREA DECLARATIONS                                    *
012200*****************************************************************
012300
012400 01  MSG-TEXT.
012500     03 MSG-TCODE              PIC X(6).
```

```
012600     03 MSG-FILLER              PIC X.
012700     03 MSG-CREATE-RACE.
012800         05 MSG-CR-ID            PIC 9(6).
012900         05 MSG-CR-NAME          PIC X(20).
013000         05 MSG-CR-DATE          PIC X(6).
013100         05 MSG-CR-TIME          PIC X(4).
013200         05 MSG-CR-LOCATION      PIC X(20).
013300         05 MSG-CR-SPONSOR       PIC X(10).
013350         05 FILLER               PIC X(10).
013400     03 MSG-ADD-ENTRY REDEFINES MSG-CREATE-RACE.
013500         05 MSG-AE-RACE-ID       PIC 9(6).
013600         05 MSG-AE-ID            PIC X(6).
013700         05 MSG-AE-NAME          PIC X(20).
013800         05 MSG-AE-RATING        PIC 9(3).
013900         05 MSG-AE-OWNER         PIC X(20).
014000         05 MSG-AE-CLUB          PIC X(15).
014100         05 FILLER               PIC X(6).
014200     03 MSG-DELETE-ENTRY REDEFINES MSG-CREATE-RACE.
014300         05 MSG-DE-RACE-ID       PIC 9(6).
014400         05 MSG-DE-ID            PIC X(6).
014500         05 FILLER               PIC X(64).
014600     03 MSG-STATUS               PIC X(30).
014700
014710*****************************************************************
014720*  COMS INTERFACE DECLARATIONS                                  *
014730*****************************************************************
014800 COMMUNICATION SECTION.
017000
017002 INPUT HEADER COMS-IN;
017004   PROGRAMDESG            IS COMS-IN-PROGRAM;
017006   FUNCTIONSTATUS         IS COMS-IN-FUNCTION-STATUS;
017008   FUNCTIONINDEX          IS COMS-IN-FUNCTION-INDEX;
017010   USERCODE               IS COMS-IN-USERCODE;
017012   SECURITYDESG           IS COMS-IN-SECURITY-DESG;
017014   TRANSPARENT            IS COMS-IN-TRANSPARENT;
017016   VTFLAG                 IS CONS-IN-VT-FLAG;
017018   TIMESTAMP              IS COMS-IN-TIMESTAMP;
017020   STATION                IS COMS-IN-STATION;
017022   TEXTLENGTH             IS COMS-IN-TEXT-LENGTH;
017024   STATUSVALUE            IS COMS-IN-STATUS-KEY;
017026   MESSAGECOUNT           IS COMS-IN-MSG-COUNT;
017028   RESTART                IS COMS-IN-RST-LOC;
017030   AGENDA                 IS COMS-IN-AGENDA;
017031   CONVERSATION AREA.
017032     02 CA PIC X(60).
017034
017036 OUTPUT HEADER COMS-OUT;
017038   DESTCOUNT              IS COMS-OUT-COUNT;
```

```
017040   TEXTLENGTH             IS COMS-OUT-TEXT-LENGTH;
017042   STATUSVALUE            IS COMS-OUT-STATUS-KEY;
017044   TRANSPARENT            IS COMS-OUT-TRANSPARENT;
017046   VTFLAG                 IS COMS-OUT-VT-FLAG;
017048   CONFIRMFLAG            IS COMS-OUT-CONFIRM-FLAG;
017050   CONFIRMKEY             IS COMS-OUT-CONFIRM-KEY;
017052   DESTINATIONDESG        IS COMS-OUT-DESTINATION;
017054   NEXTINPUTAGENDA        IS COMS-OUT-NEXT-INPUT-AGENDA;
017055   CASUALOUTPUT           IS COMS-OUT-CASUAL-OUTPUT;
017056   SETNEXTINPUTAGENDA     IS COMS-OUT-SET-NEXT-INPUT-AGENDA;
017058   RETAINTRANSACTIONMODE  IS COMS-OUT-SAVE-TRANS-MODE;
017060   AGENDA                 IS COMS-OUT-AGENDA.
017100*****************************************************************
017200 PROCEDURE DIVISION.
017300*****************************************************************
017400 DECLARATIVES.
017500 DMERROR-SECTION SECTION.
017600     USE ON DMERROR.
017700 DMERROR-PARA.
017800
017900 END DECLARATIVES.
018000
018100 005-MAIN SECTION.
018110*    Link application program to COMS.
018200 005-MAIN-SN.
018500     CHANGE ATTRIBUTE LIBACCESS OF
018600          "DCILIBRARY" TO BYFUNCTION.
018700     CHANGE ATTRIBUTE FUNCTIONNAME OF
018800          "DCILIBRARY" TO "COMSSUPPORT".
018900     OPEN UPDATE SAILDB.
019000     IF DMSTATUS(DMERROR) CALL SYSTEM DMTERMINATE.
019010*    Initialize interface to COMS.
019100     ENABLE INPUT COMS-IN KEY "ONLINE".
019200     CREATE RDS.
019420     PERFORM 007-PROCESS-COMS-INPUT
019440       UNTIL COMS-IN-STATUS-KEY = 99.
019500 005-MAIN-EXIT.
019510     PERFORM 910-CLOSEDOWN.
019520     STOP RUN.
019530
019540*****************************************************************
019550 007-PROCESS-COMS-INPUT SECTION.
019560*****************************************************************
019562*    Get the next message from COMS.  If it is a 99, go to
019564*    end of task (EOT); otherwise, make sure it is a valid
019566*    message before processing it.
```

```
019568*
019570 007-PROCESS-CI-SN.
019600     MOVE SPACES TO MSG-TEXT.
019700     RECEIVE COMS-IN MESSAGE INTO MSG-TEXT.
019800     IF COMS-IN-STATUS-KEY NOT = 99
020000       PERFORM 920-CHECK-COMS-INPUT-ERRORS
020100       IF ( COMS-IN-STATUS-KEY =   0 OR 92)  AND
020200            COMS-IN-FUNCTION-STATUS NOT < 0 THEN
020300         PERFORM 100-PROCESS-TRANSACTION.
020400 007-PROCESS-CI-EXIT.
020450     EXIT.
020500
021000*****************************************************************
021100 100-PROCESS-TRANSACTION  SECTION.
021200*****************************************************************
021220*    Since the transaction type is programmatically based on MFI,
021240*    make sure it is within range before doing the GO TO.
021260
021300 100-PROCESS-TRANS-SN.
021400     IF MSG-TCODE = "CRERAC"
021500       PERFORM 200-CREATE-RACE
021600     ELSE
021700     IF MSG-TCODE = "ADDENT"
021800       PERFORM 300-ADD-ENTRY
021900     ELSE
022000     IF MSG-TCODE = "DELENT"
022100       PERFORM 400-DELETE-ENTRY
022200     ELSE
022300       MOVE "INVALID TRANS CODE" TO MSG-STATUS
022320       PERFORM 900-SEND-MSG.
022340
022400 100-PROCESS-TRANS-EXIT.
022500     EXIT.
022600
022700*****************************************************************
022800 200-CREATE-RACE  SECTION.
022900*****************************************************************
023000 200-CREATE-RACE-SN.
023020*    Enter a new race record in the database.
023030*    Because the transaction is done in online mode,
023040*    save the restart data set (RDS) in the conversation area
023050*    only.  If there is an ABORT on BEGIN-TRANSACTION or
023060*    END-TRANSACTION, return to the RECEIVE statement.
023080
023100     CREATE RACE-CALENDAR.
023200     MOVE MSG-CR-NAME     TO RACE-NAME.
023300     MOVE MSG-CR-ID       TO RACE-ID.
```

```
023400     MOVE MSG-CR-DATE     TO RACE-DATE.
023500     MOVE MSG-CR-TIME     TO RACE-TIME.
023600     MOVE MSG-CR-LOCATION TO RACE-LOCATION.
023700     MOVE MSG-CR-SPONSOR  TO RACE-SPONSOR.
023800
023950     BEGIN-TRANSACTION COMS-IN                NO-AUDIT RDS
024000        ON EXCEPTION
024100        IF DMSTATUS(ABORT)
024150        THEN
024200*         Return to the RECEIVE statement.
024300          GO TO 200-CREATE-RACE-EXIT
024400        ELSE
024500          CALL SYSTEM DMTERMINATE.
024700     STORE RACE-CALENDAR.
024800     IF DMSTATUS(DMERROR)
024900       MOVE "STORE ERROR" TO MSG-STATUS
025000     ELSE
025100       MOVE "RACE ADDED" TO MSG-STATUS.
025200     END-TRANSACTION COMS-OUT AUDIT RDS
025300       ON EXCEPTION
025400       IF DMSTATUS(ABORT)
025500         GO TO 200-CREATE-RACE-EXIT
025600       ELSE
025700         CALL SYSTEM DMTERMINATE.
025800     PERFORM 900-SEND-MSG
025900 200-CREATE-RACE-EXIT.
026000     EXIT.
026100
026200*****************************************************************
026300 300-ADD-ENTRY   SECTION.
026400*****************************************************************
026500 300-ADD-ENTRY-SN.
026520*    Enter a boat in a race. The restart requirements are the same
026540*    as for the above transaction.
026560
026600     FIND RACE-SET AT RACE-ID = MSG-AE-RACE-ID
026700       ON EXCEPTION
026800         IF DMSTATUS(NOTFOUND)
026900           MOVE "RACE DOES NOT EXIST" TO MSG-STATUS
027000           PERFORM 900-SEND-MSG
027100           GO TO 300-ADD-ENTRY-EXIT
027200         ELSE
027300           CALL SYSTEM DMTERMINATE.
027400
027500     CREATE ENTRY.
```

```
027600     MOVE MSG-AE-NAME      TO ENTRY-BOAT-NAME.
027700     MOVE MSG-AE-ID        TO ENTRY-BOAT-ID.
027800     MOVE MSG-AE-RATING    TO ENTRY-BOAT-RATING.
027900     MOVE MSG-AE-OWNER     TO ENTRY-BOAT-OWNER.
028000     MOVE MSG-AE-CLUB      TO ENTRY-AFF-Y-CLUB.
028100     MOVE MSG-AE-RACE-ID   TO ENTRY-RACE-ID.
028200
028350     BEGIN-TRANSACTION COMS-IN               NO-AUDIT RDS
028400        ON EXCEPTION
028500        IF DMSTATUS(ABORT)
028550        THEN
028600*         Return to the RECEIVE statement.
028700          GO TO 300-ADD-ENTRY-EXIT
028800        ELSE
028900          CALL SYSTEM DMTERMINATE.
029100     STORE ENTRY.
029200     IF DMSTATUS(DMERROR)
029300       MOVE "STORE ERROR" TO MSG-STATUS
029400     ELSE
029500       MOVE "BOAT ADDED" TO MSG-STATUS.
029600     END-TRANSACTION COMS-OUT AUDIT RDS
029700       ON EXCEPTION
029800       IF DMSTATUS(ABORT)
029900         GO TO 300-ADD-ENTRY-EXIT
030000       ELSE
030100         CALL SYSTEM DMTERMINATE.
030200     PERFORM 900-SEND-MSG.
030300 300-ADD-ENTRY-EXIT.
030400     EXIT.
030500
030600*****************************************************************
030700 400-DELETE-ENTRY  SECTION.
030800*****************************************************************
030900 400-DELETE-ENTRY-SN.
030920*    Delete a boat from a race.  The restart requirements are the
030940*    same as for the previous transaction.
030960
031000     LOCK ENTRY-RACE-SET AT
031190       ENTRY-RACE-ID = MSG-DE-RACE-ID AND
031200       ENTRY-BOAT-ID = MSG-DE-ID
031300       ON EXCEPTION
031400       IF DMSTATUS(NOTFOUND) THEN
031500         MOVE "BOAT ENTRY NOT FOUND" TO MSG-STATUS
031600         GO TO DE-SEND-MSG
```

```
031700       ELSE
031800         CALL SYSTEM DMTERMINATE.
031900
032050     BEGIN-TRANSACTION COMS-IN                NO-AUDIT RDS
032100        ON EXCEPTION
032200        IF DMSTATUS(ABORT)
032250        THEN
032300*         Return to the RECEIVE statement.
032400          GO TO 400-DELETE-ENTRY-EXIT
032500        ELSE
032600          CALL SYSTEM DMTERMINATE.
032800     DELETE ENTRY.
032900     IF DMSTATUS(DMERROR)
033000       MOVE "FOUND BUT NOT DELETED" TO MSG-STATUS
033100     ELSE
033200       MOVE "BOAT DELETED" TO MSG-STATUS.
033300     END-TRANSACTION COMS-OUT AUDIT RDS
033400       ON EXCEPTION
033500       IF DMSTATUS(ABORT)
033600         GO TO 400-DELETE-ENTRY-EXIT
033700       ELSE
033800         CALL SYSTEM DMTERMINATE.
033900 DE-SEND-MSG.
034000     PERFORM 900-SEND-MSG.
034100 400-DELETE-ENTRY-EXIT.
034200     EXIT.
034300
034400*****************************************************************
034500 900-SEND-MSG  SECTION.
034600*****************************************************************
034700 900-SEND-MSG-SN.
034720*    Send the message back to the originating station.
034740*    Do not specify an output agenda.  Make sure to
034760*    test the result of the SEND.
034780*
034800     MOVE 1               TO COMS-OUT-COUNT.
034950     MOVE 0               TO COMS-OUT-DESTINATION.
035000     MOVE 0               TO COMS-OUT-STATUS-KEY.
035100     MOVE 106             TO COMS-OUT-TEXT-LENGTH.
035200     SEND COMS-OUT FROM MSG-TEXT.
035300     IF COMS-OUT-STATUS-KEY = 0 OR 92
035400       NEXT SENTENCE
035500     ELSE
035600       DISPLAY "ONLINE PROGRAM SEND ERROR: " COMS-OUT-STATUS-KEY.
035700 900-SEND-MSG-EXIT.
035800     EXIT.
035900*****************************************************************

036000 910-CLOSEDOWN  SECTION.
036100*****************************************************************
036200 910-CLOSEDOWN-SN.
036220*    Close the database.
```

```
036700     CLOSE SAILDB.
036800 910-CLOSEDOWN-EXIT.
036900     EXIT.
037000*****************************************************************
037100 920-CHECK-COMS-INPUT-ERRORS  SECTION.
037200*****************************************************************
037300 920-CHECK-CIE-SN.
037320*    Check for COMS control messages.
037340
037400     IF ( COMS-IN-STATUS-KEY =   0 OR 92 OR 99 )
037450
037500*       These codes signify a good message, a recovery message,
037550*       and an EOT notification, respectively.
037575*
037600        NEXT SENTENCE
037650
037700     ELSE
037800     IF COMS-IN-STATUS-KEY = 93
037900       MOVE "MSG CAUSES ABORT, PLS DONT RETRY" TO MSG-STATUS
038000       PERFORM 900-SEND-MSG
038100     ELSE
038150
038200*      The COMS control message is 20, 100, 101, or 102, which
038300*      means the application is manipulating the dynamic
038400*      attachment or detachment of stations and receives an error.
038450
038500       MOVE "ERROR IN STA ATTACH/DETACHMENT" TO MSG-STATUS
038600       PERFORM 900-SEND-MSG.
038700
038800     IF COMS-IN-FUNCTION-STATUS < 0 THEN
038850
038900*      This means that the application ID is tied to a default
039000*      input agenda.  MSG-TEXT probably does not contain a valid
039010*      transaction.
039020
039100       MOVE "NEGATIVE FUNCTION CODE " TO MSG-STATUS
039200       PERFORM 900-SEND-MSG THRU 900-SEND-MSG-EXIT.
039300
039400 920-CHECK-CIE-EXIT.
039500     EXIT.
039600
```

Example 2–25. COMS Sample Program with a DMSII Database.

Section 3
Using the DMSII Program Interface

Data Management System II (DMSII) is used to invoke a database and maintain relationships among the various data elements in the database.

This section explains how to use the extensions developed for the DMSII program interface. The DMSII extensions allow you to

- Identify, qualify, and reference database items.
- Declare and invoke a database.
- Invoke data sets.
- Use database equation operations to specify and manipulate database titles, and to override compiled titles.
- Use selection expressions to identify a particular record in a data set.
- Use data management attributes for read-only access to count, record, and population items.
- Manipulate data through data management statements.
- Process exceptions.

For an alphabetized list of the extensions developed for DMSII, refer to the list of DMSII extensions in Section 1, "Introduction to COBOL85 Program Interfaces." Refer to the *DMSII Application Program Interfaces Programming Guide* for information on general programming considerations and concepts.

DMSII can be used with the Communications Management System (COMS). For more information, refer to Section 2, "Using the COMS Program Interface."

Using Database Items

This discussion describes the naming conventions for data items in a database (database items) and explains how to reference them. A data record from a database is accessed directly by a COBOL85 program.

Naming Database Components

Data and Structure Definition Language (DASDL) naming conventions for database components follow COBOL85 rules. More specifically, some item or structure names can require qualification and some can contain hyphens (-). Whenever syntax specifies the names of database components, these names can be fully qualified names and can contain hyphens.

Data item names can be 1 to 30 characters long or 1 to 14 double–byte characters long. Structure names (dataset, set, and subset names) can be 1 to 17 characters long. When ALIAS names are specified for structures in the DASDL source file, they can be 1 to 30 characters long or 1 to 14 double–byte characters long. ALIAS names can be specified in a COBOL85 program to invoke the structure associated with the ALIAS name. The logical database name and the database name can be 1 to 17 EBCDIC characters long.

Using Set and Data Set Names

You must qualify set and data set names that are used to find records if the names are not unique. You can declare a variable name with the same name as a database item if the item can be qualified.

If you invoke a data set more than once (using internal names), you must qualify any reference to an item or set within that data set. To qualify the reference, use the internal name associated with the invocation you want. If you use improper or insufficient qualification, you receive a syntax error. For example, assume a database declared as follows:

```
DB PAYROLL.
01 EMP = EMPLOYEES.
01 MGR = EMPLOYEES.
```

Assume that the data set called EMPLOYEES contains a set named EMP-NBR-SET. You must qualify any references to this set or items in the data set with an internal name, either EMP or MGR. The following statements show examples of these qualified references.

```
FIND NEXT EMP-NBR-SET OF EMP.

FIND EMP-NBR-SET OF MGR AT EMP-NBR = EMP-MGR-NBR OF EMP.

MOVE EMP-NAME OF EMP TO PR-NAME.

MOVE EMP-NAME OF MGR TO PR-MGR-NAME.
```

You can use any number of group item names for qualification, as long as the result is unique. The general format of the statement used to qualify a name is as follows:

Format

```
identifier-1  [ OF identifier-2 ] . . .
```

Explanation

Identifier-1 and identifier-2 are DASDL identifiers.

Examples

The following examples show code in which name qualification is needed or in which a successful or unsuccessful attempt has been made to provide qualification. The applicable DASDL descriptions are provided.

Qualifying Valid and Invalid Names

Example 3–1 applies to the following DASDL description:

```
DASDL (compiled as DBASE):
     D1 DATA SET (
        A NUMBER (5););
```

Example 3–1 shows an example in which the A declared at level 77 is invalid because it cannot be qualified. However, the A declared at the level 03 is valid because it can be qualified.

```
*Invalid DATA DIVISION entry:
  77 A PIC . . .
*Valid DATA DIVISION entry:
  01 Q.
   03 A . . .
```

Example 3–1. Qualifying DMSII Valid and Invalid Names

Using Names Requiring Qualification

Example 3–2 applies to the following DASDL description:

```
DASDL (compiled as DBASE):
 D1 DATA SET (
    A NUMBER (5);
    B . . .
    .
    .
    .        );
 S1 SET OF D1 KEY A;
```

Example 3–2 shows an example in which S1 and A require qualification.

```
DB DBASE.
 01 D1.
 01 DA=D1.

FIND S1 OF D1 AT A=V.
 MOVE A OF D1 TO LA.
```

Example 3–2. Using DMSII Names Requiring Qualification

Referencing Database Items

You can invoke all or part of a database in the Data-Base Section of your program. When the description is invoked, the compiler generates the interfaces needed to allocate the proper record areas when the database is opened.

The record area for a data set is established in two parts: One part contains control items, and the other contains data items. You set up the part that contains data items like a 01-level Working-Storage Section entry. This setup enables you to use the data manipulation statements to move database items, including groups.

Group moves are always considered alphanumeric moves. The arrangement of the data item record area also enables you to use MOVE CORRESPONDING statements. For more information on MOVE statements, refer to Volume 1.

Note that aggregate items are read-only. For more information about aggregate items, refer to the *DMSII Data and Structure Definition Language (DASDL) Programming Reference Manual*.

If you use variable-format records in your programs, a group move at the 01 level fills the receiving area without regard to the individual items contained within either the sending or receiving area. Using variable-format records can therefore cause unexpected values to be stored in the receiving area. For MOVE CORRESPONDING statements, only items in the fixed portion of the record are candidates for the move.

Examples

Examples 3–3 through 3–5 reference database items that contain compiler-produced listings.

Group Move

Example 3–3 illustrates a group move involving database items. The items T, CT, L, E, and S are control items and are not affected by moves to or from D.

The record area for D is the following:

```
01  D
     02  A
     02  B
     02  C
```

E1 and E2 are items of the record area for E rather than D; therefore, they are not affected by moves to or from D.

```
01  D  DATA SET (#1).
     02  T RECORDTYPE.
     02  CT COUNT.
     02  A PIC X(6) DISPLAY.
     02  B PIC 9(6) COMP.
     02  C PIC 9(6) COMP.
     02  L REFERENCE TO E.
     02  E DATA SET (#2).
         03  E1 . . .
         03  E2 . . .
     02  S SET(#4,MANUAL) OF E.
```

Example 3–3. Moving a DMSII Group of Database Items

Receiving Fields of a MOVE CORRESPONDING Statement

Example 3–4 describes the effect of a MOVE CORRESPONDING statement that involves database items. The items contained in the record depend on the value of T as follows:

- If T equals 0, then the record area contains T, A, and B.
- If T equals 1, then the record area contains T, A, B, and X.
- If T equals 2, then the record area contains T, A, B, and Y.

In this example, because A and B are in the fixed portions of the record, they are the only candidates for a MOVE CORRESPONDING statement on D.

The items X and Y are never moved as a result of a MOVE CORRESPONDING statement. T is not affected by the MOVE CORRESPONDING statement because it is not in a fixed portion of the record.

```
01  D
     02  T RECORDTYPE.
     02  A PIC X(6) DISPLAY.
     02  B PIC 9(6) COMP.
* FORMAT TYPE 1.
     02  X PIC 9(6) COMP.
* FORMAT TYPE 2.
     02  Y PIC 9(11) COMP.
```

Example 3–4. Receiving Fields of a MOVE CORRESPONDING Statement

Creating an Invalid DMSII Index

Example 3–5 applies to the following DASDL description:

```
DASDL:
  C  COMPACT DATASET
    (N    NUMBER (1);
     O    NUMBER (5) OCCURS 9 TIMES DEPENDING ON N);
```

Example 3–5 shows two lines of COBOL85 code that use MOVE statements. For both statements, the variable N in the DASDL description equals 5. The first line executes successfully. The second line, however, creates an invalid index error because the program attempts to access an occurrence of an OCCURS DEPENDING ON item that is larger than the current value of the DEPENDING ON item.

```
MOVE 123 to O(3).
MOVE 456 to O(7).
```

Example 3–5. Creating an Invalid DMSII Index

Declaring a Database

The Data-Base Section of a COBOL85 program supplies the COBOL85 compiler with a description of all or selected portions of one or more databases. You place the Data-Base Section in the Data Division after the File Section and before the Working-Storage Section.

The database declaration supplies information that identifies a given database. The compiler lists all the invoked descriptions of record formats, items, sets, subsets, and keys.

The general format for a database declaration is as follows:

Format

```
DB [ data-name-1  { INVOKE } ]   [ data-name-2 OF ] data-name-3
                  {   =    }

 [ GLOBAL ]
 [ COMMON ]

 [ ALL ]
 [ VALUE OF TITLE IS  literal-1 ]
```

Explanation

INVOKE and the equal sign (=) are synonyms.

Data-name-1 specifies the internal name of the database, data sets, sets, or subsets within the program. When you use the INVOKE clause in your program, you must use the internal names of renamed structures in all subsequent references to them. A structure using an alias identifier can be a string literal enclosed in quotation marks (“).

You can invoke a database, data set, set, or subset more than once; however, you must use the external name to reference only one invocation of each structure. Data-name-1 must be used to provide unique names for all other invocations of the structures. The default internal name is the external name of the structure.

You can establish multiple record areas, set paths, or both by specifying data-name-1 with a data set reference or set reference. In this way, several records of a single data set can be manipulated simultaneously.

Data-name-2 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. You specify selective invocations like physical databases; however, you can select only those structures that are included in the logical database.

Data-name-3 is the name of the database to be invoked. You can use data-name-3 as a qualifier of a data set, set, or subset. If you use the INVOKE clause, data-name-3 can be a string literal enclosed in quotation marks.

The COMMON clause declares a database to be common, enabling a separately compiled procedure or program to reference the database declared in your COBOL85 program. The database reference in the separately compiled procedure or program must exactly match the database reference in your COBOL85 program. In addition, the various components to be bound together must be compiled against the same database description file. Failure to observe these restrictions results in syntax errors when you attempt to combine the programs by using Binder.

The GLOBAL clause declares a database to be global. A global database is available to every program that declares it.

The ALL clause specifies that all structures of the specified database are to be used.

The only mnemonic allowed for the file attribute name in the VALUE OF clause is TITLE. For more information about the VALUE OF clause, refer to Volume 1.

For a database, the operating system constructs the control file title from the title specified in the declaration. The default title is the name plus the control file usercode and family name, if any, from the description file. Refer to the *DMSII Data and Structure Definition Language (DASDL) Programming Reference Manual* for a discussion of control and description files.

Examples

The following examples show different formats you can use to declare the database:

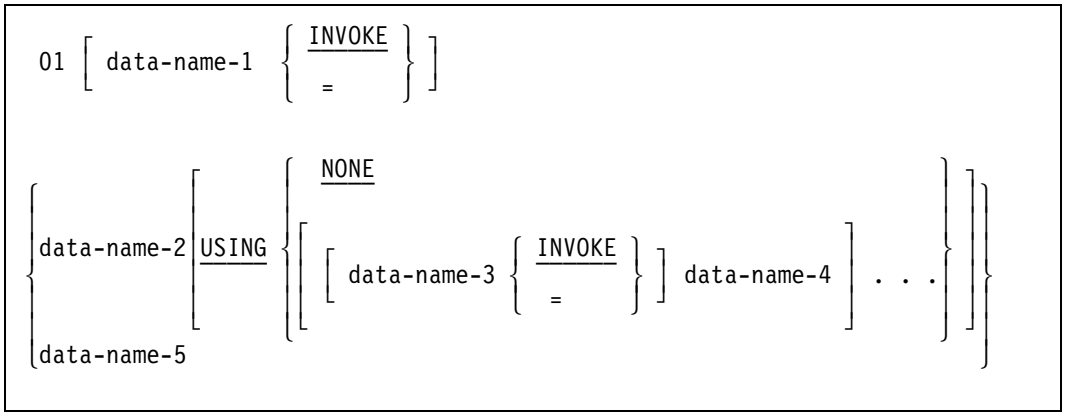
```
DB DATABASE-1.

DB MY-DB-2 = DATABASE-2.

DB DATABASE-3 GLOBAL.

DB DATABASE-4 VALUE OF TITLE IS "(XYZ)DATABASE/4".

DB DATABASE-5 ALL.

DB DATABASE-6.
 01 DATASET-1.
 01 DATASET-2.
```

Invoking Data Sets

The data set reference specifies the structures that are to be invoked from the declared database. If you do not specify a particular data set to be invoked, all structures are invoked implicitly.

The general format for the data set reference is as follows:

Format



Explanation

The data set reference must be written at the 01 level. INVOKE and the equal sign (=) are synonyms.

If you use the data set reference, only the data sets that you specify are invoked. If you do not use the data set reference, you implicitly invoke all structures of the specified database.

You cannot invoke only the global data from a database: you must invoke at least one structure. You can invoke the structure explicitly in the data set reference, or implicitly by default or by using the ALL clause in the database (DB) statement.

Data-name-1 is the internal name of the data set or set.

Data-name-2 must be the name of the data set to be used. You can use data-name-2 as a qualifier of its embedded structures. If the INVOKE clause is used, data-name-2 can be a string literal enclosed in quotation marks (“).

Data-name-3, data-name-4, and data-name-5 must contain the name of a set.

The USING clause invokes specific sets from the data sets declared in the data set reference. You can use data-name-3 in your program to reference a synonym for the set specified by data-name-4. All subsequent references to the set previously specified as data-name-4 must use data-name-3. If you omit the USING clause, you invoke all sets. If you specify the USING clause, you invoke no sets (NONE) or only the specified sets.

When you use data-name-5, data-name-1 must be the name of a set. Data-name-5 becomes a set reference that is not implicitly associated with any particular record area. You must specify the data set name VIA option in the selection expression to load a record area using data-name-5. Additional information on the VIA option is provided in "Using Selection Expressions" later in this section.

You can explicitly invoke only disjoint structures. Embedded data sets, sets, and subsets are always implicitly invoked if their master data sets are implicitly or explicitly invoked. You must reference all implicitly invoked structures by their external names.

To use a path, you must invoke the disjoint data set. You establish a path by invoking a data set containing either of the following:

- An embedded set associated with a disjoint data set
- A link to another disjoint data set

Multiple invocations of a structure provide multiple record areas, 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 options provides better use of system resources.

You invoke remaps declared in the DASDL in the same way that you invoke conventional data sets.

Examples of Invoking Data Sets

The following examples show code that invokes a data set. Each example is preceded by an explanation. The following DASDL description applies to the examples:

```
DASDL (compiled as DB):
D   DATA SET (
     K   NUMBER (6);
     R   NUMBER (5);
     );
S1  SET OF D KEY K;
S2  SET OF D KEY R;
```

- The following example establishes one current record area for the data set D, one path for the set S1, and one path for the set S2. Executing the FIND S1, MODIFY S1, FIND S2, or MODIFY S2 statement automatically loads the data to the D record area.

```
01 D.  (S1 and S2 are invoked implicitly.)
```

- The following example 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 USING NONE option prevents a set from being associated with X. Thus, using the FIND S1 or FIND S2 statement loads the D record area. The FIND X VIA S1 or FIND X VIA S2 statement must be executed to load the X record area using a set.

```
01 D.  (S1 and S2 are invoked implicitly.)
01 X=D USING NONE.
```

- The following example shows how multiple current record areas and multiple current paths can be established. Using the FIND S1 OF D statement loads the D record area without disturbing the path S1 OF X, and using the FIND S1 OF X statement loads the X record area without disturbing the path S1 OF D. S1 must be qualified.

```
01 D.    (S1 and S2 are invoked implicitly.)
01 X=D.  (S1 and S2 are invoked implicitly.)
```

- The following example shows how to establish more current record areas than paths. In this example, three record areas (D, X, and Y) are established with only two paths (S1 OF D and S1 OF X). To load the Y record area, the program must execute the FIND Y VIA S1 OF D, FIND Y VIA S1 OF X, or FIND Y statement.

```
01 D   USING S1.  (S1 is invoked explicitly.)
01 X=D USING S1.  (S1 is invoked explicitly.)
01 Y=D USING NONE.
```

- The following example shows the USING clause syntax explicitly associating a set with a given work area. Using the FIND S1 statement loads the X record area, and using the FIND T statement loads the Y record area. Sets S1 and T both use the same key.

```
01 X=D USING S1
01 Y=D USING T=S1.
```

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

```
01 D.
01 SY=S1.
```

Example of Invoking Disjoint Data Sets with a Data Set Reference

This example of using data set references to invoke disjoint data sets applies to the following DASDL description:

```
DASDL (compiled as DBASE):
 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;
   );
```

In the following example, only data set D is specified. Although the data set references are not specified to invoke E and F, the paths are established by invoking the embedded set SE and the link item LINK.

```
01 D.
```

These paths, however, cannot be used unless you specify data set references for E and F to establish record areas for these paths, as shown in the following lines of code:

```
01 F.
01 E.
01 D.
```

Example of Designating Sets as Visible or Invisible to User Programs

Example 3–6 shows how sets can be designated as visible or invisible to user programs. The example applies to the following DASDL description:

```
DASDL (compiled as EXAMPLEDB):
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);
```

Example 3–6 shows the COBOL85 code for designating sets as visible or invisible to user programs. After you compile this code, the commented code appears in the listing. For logical database LDB2, the following sets are visible to the user program:

- Data set D1 and its set S1B (referenced as S1)
- Data set D2 and its set S2B (referenced as S2)

Sets S1A and S2A are invisible to the user program. LDB1 and LDB3 are invisible because they are not declared in the Data-Base Section.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  DBTEST.
 DATA DIVISION.
 DATA-BASE SECTION.
 DB LDB2 OF EXAMPLEDB ALL.
*  01 D1 STANDARD DATA SET(#2).
*        S1 SET(#4,AUTO) OF D1 KEYS ARE A,B,C.
*     02 A REAL.
*     02 B PIC 9(5) COMP.
*     02 C PIC X(10) DISPLAY.
*  01 D2 STANDARD DATA SET(#5).
*        S2 SET(#9,AUTO) OF D2 KEYS ARE X,Y,Z.
*     02 X FIELD SIZE IS 08 BITS.
*     02 Y PIC 99 COMP.
*     02 Z REAL.
*     02 E STANDARD DATA SET(#6).
*           SE SET(#7,AUTO) OF E KEY IS V1.
*        03 V1 REAL.
*        03 V2 PIC XX DISPLAY.
 PROCEDURE DIVISION.
 T.
    STOP RUN.
```

Example 3–6. Designating DMSII Sets as Visible or Invisible

Examples of Using the GLOBAL Option to Reference a Database

In Example 3–7, a separately compiled procedure, SEP/P, uses the GLOBAL option to reference the database declared in a COBOL85 program. The database reference in the separately compiled procedure or program must exactly match the corresponding database reference in the COBOL85 program, or an error occurs at bind time. This example applies to the following DASDL description:

```
  DASDL (compiled as TESTDB):
  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);
```

Example 3–7 shows the SEP/P procedure.

```
$ LEVEL=3
 .
 .
 .
 DATA-BASE SECTION.
 DB TESTDB GLOBAL ALL.
 PROCEDURE DIVISION.
 P1.
    SET NAMESET TO BEGINNING.
 PERFORM P2 UNTIL (DMSTATUS (NOTFOUND).
 P2.
 FIND NEXT NAMESET AT LAST = "SMITH" AND FIRST = "JOHN".
*   OTHER STATEMENTS
 ?BEGIN JOB BIND/GLOB;
  BIND GLOBDB WITH BINDER LIBRARY;
  BINDER DATA CARD
  HOST IS SEP/HOST;
  BIND P FROM SEP/P;
 ?END JOB.
```

Example 3–7. Using a Separately Compiled Procedure to Reference a Database with the GLOBAL Clause

Example 3–8 shows the COBOL85 program declarations for the host program and the corresponding database reference. The program is compiled as SEP/HOST.

```
  .
  .
  .
  DATA-BASE SECTION.
  DB TESTDB ALL.
  PROCEDURE DIVISION.
  DECLARATIVES.
  P SECTION.  USE EXTERNAL AS PROCEDURE.
  END DECLARATIVES.
  P1.
     OPEN UPDATE TESTDB.
     CALL P.
     CLOSE TESTDB.
     STOP RUN.
```

Example 3–8. Declaring a DMSII Host Program to Be Used with the GLOBAL Clause

Using a Database Equation Operation

Database equation is like file equation. It enables access to databases stored under other usercodes and on pack families not visible to a task. It enables you to change or manipulate the database title at run time.

Database equation differs from file equation in that a run-time error results if a COBOL85 program attempts to set or examine the TITLE attribute of the database while it is open.

There are three different ways to equate or manipulate database titles, and each of these operations is done at a different time. They are as follows:

- Specifying database titles in the Data Division when the program is compiled, using the DB statement.
- Using work flow language (WFL).

  WFL equation overrides database titles specified in a language declaration in the Data Division at compilation time.

- Specifying database titles at program execution, using the MOVE or CHANGE ATTRIBUTE statement in the Procedure Division.

  Modifying database titles at run-time overrides both WFL equation and user language specifications in the Data Division.

The reentrance capability of the Accessroutines is available only when the title of a database is specified at run time.

Specifying Database Titles at Program Execution

The MOVE statement and the CHANGE statement manipulate the database TITLE attribute during program execution. Refer to Volume 1 for information on these statements and on the TITLE attribute.

The general format of the MOVE and CHANGE statements is as follows:

Format

```
{ MOVE   }
{ CHANGE } ATTRIBUTE TITLE { OF } internal-name
                           { IN }
        TO alphanumeric-data-item
```

Example

In Example 3–9, the first OPEN statement opens LIVEDB. The data and control files of LIVEDB are stored under the disk directory of the user. The second OPEN statement invokes TESTDB. The files for TESTDB are stored on TESTPACK under the usercode UC.

Examples of the MOVE and CHANGE statements within the context of a complete program are provided in Example 2–25, "COMS Sample Program with a DMSII Database," in Section 2.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DBEQUATE
.
.
.
DATA-BASE SECTION.
  DB MYDB ALL
  VALUE OF TITLE IS "LIVEDB".
.
.
.
PROCEDURE DIVISION.
.
.
.
OPEN UPDATE MYDB.
.
.
.
CLOSE MYDB.
CHANGE ATTRIBUTE TITLE OF MYDB
    TO "(UC)TESTDB ON TESTPACK".
OPEN UPDATE MYDB.
.
.
.
CLOSE MYDB.
STOP RUN.
```

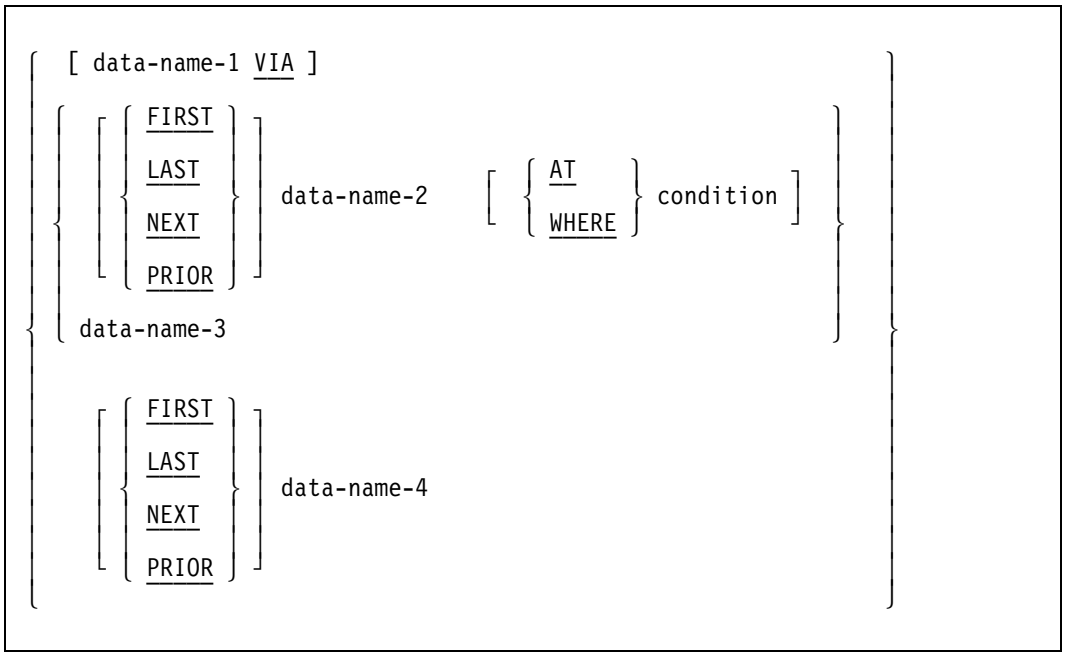
Example 3–9. Performing DMSII Database Equation Operations

Using Selection Expressions

A selection expression is used in FIND, LOCK, MODIFY, and DELETE statements to identify a particular record in a data set.

The general format for selection expressions is as follows:

Format



Explanation

Data-name-1 identifies the record area and the current path that is affected if the desired record is found. You can use this option for link items and for sets not implicitly associated with the data set.

Data-name-2 selects the record referred to by the set path. Data-name-2 must be a set or a subset. DMSII returns a NOTFOUND exception if the record has been deleted or if the path does not refer to a valid current record.

Data-name-3 specifies a link item defined in the DASDL. DMSII selects the record to which the link item refers and returns an exception if the link item is NULL.

Data-name-4 must be a data set name. Data-name-4 selects the record referred to by the data set path. A NOTFOUND exception is returned if the record has been deleted or if the path does not refer to a valid current record.

The FIRST clause selects the first record in the specified data set, set, or subset. FIRST is specified by default. If you specify a key condition, DMSII selects the first record that satisfies the key condition.

The LAST clause selects the last record in the specified data set, set, or subset. If you specify a key condition, DMSII selects the last record that satisfies the key condition.

The NEXT clause selects the next record relative to one of the following:

- The set path if you specify a set name or subset name
- The data set path if you specify a data set name

If you specify a key condition, DMSII selects the next record (relative to the current path) that satisfies the key condition.

The PRIOR clause selects the prior record relative to one of the following:

- The set path if you specify a set name or subset name
- The data set path if you specify a data set name

The AT or WHERE clause indicates that a key condition follows. AT and WHERE are synonyms.

The condition clause specifies values used to locate specific records in a data set referenced by a particular set or subset. The condition clause is also referred to as the key condition. If you specify a key condition, DMSII selects the prior record (relative to the current path) that satisfies the key condition.

A key condition is made up of DMSII data items, which must precede the following syntax elements:

- Relational operators
- The relational operator in a relational expression
- Data items and arithmetic expressions against which the DMSII key item is to be compared
- Left and right parentheses

Key conditions are a proper subset of condition expressions with the following additional limitations:

- The DMSII item used as the key must precede the relational operator.
- Abbreviated conditions are not allowed.

A key condition ultimately evaluates to TRUE or FALSE.

If the specified data item is not unique, the compiler provides implicit qualification through the set name or subset name. You can qualify the item by naming the data set that contains the item; however, the compiler handles this qualification as documentation only.

Examples

The following examples show selection expressions used in FIND statements. The first example locates a data set record using the set S where A is equal to 50 and B is equal to 50, or where A is equal to 50 and C is less than 90. The second example also locates a record using the set S where A is equal to the literal “MYNAME.”

```
FIND S AT A = 50 AND (B = 50 OR C < 90).

FIND S WHERE A = "MYNAME".
```

Using Data Management Attributes

Data management (DM) attributes are similar in COBOL85 to file and task attributes. DM attributes allow read-only access to the following:

- Count field of a record
- Record Type field of a record
- Current population of a structure name

Descriptions of the COUNT, RECORD TYPE, and POPULATION attributes are provided in the following text.

COUNT Attribute

The value of the COUNT attribute is the number of counted references pointing at the record in the Count field.

Because the ASSIGN statement updates the count item directly in the database, the value of the Count field can differ from the actual value in the database, even if the field is tested immediately after the record containing the Count field is made current.

Format

```
data-name-1  ( data-name-2 )
```

Explanation

Data-name-1 is the name of the data set.

Data-name-2 is a count name. The use of data-name-2 enables read-only access to the Count field of a record.

DMSII returns an exception when you attempt to delete a record and the count item is not 0 (zero).

Example

Example 3–10 provides the DASDL description for code that uses the COUNT attribute.

```
D DATA SET (
 A ALPHA (3);
 L IS IN E COUNTED;
 );

E DATA SET (
   C COUNT;
   N NUMBER (3);
   );
```

Example 3–10. Using a DASDL Description for the COUNT Attribute

The COBOL85 code for the DASDL description is as follows:

```
  IF E(C) = 0 DELETE D ON EXCEPTION PERFORM . . .
```

RECORD TYPE Attribute

The value of this attribute represents the type of record in the current record area.

Format

```
data-name-1  ( record-name-1 )
```

Explanation

Data-name-1 is the name of the data set.

The use of record-name-1 enables read-only access to the Record Type field of a record.

Example

Example 3–11 provides the DASDL description for code that uses of the RECORD TYPE attribute.

```
D DATA SET (
 T RECORD TYPE (2);
 A ALPHA (3);
 );

2: (
    B BOOLEAN;
    R REAL;
    N NUMBER (3);
            ) ;
```

Example 3–11. Using a DASDL Description for the RECORD TYPE Attribute

The COBOL85 code for this description is as follows:

```
IF D(T) = 2 GO TO . . .
```

POPULATION Attribute

The POPULATION attribute enables read-only access to the current population of the structure name. This value is often inaccurate, however, even if it is tested immediately after the record that contains it is made current, because other programs running concurrently on a multiprocessing system can cause the value of the population item in the database to change.

Format

```
data-name-1  ( data-name-2 )
```

Explanation

Data-name-1 is the name of the data set.

Data-name-2 is a population item.

Example

Example 3–12 provides the DASDL description for COBOL85 code that uses the POPULATION attribute. The operation in this example accesses not the population of D, but the population of the structure embedded in D to which EPOP refers.

```
D DATA SET ( . . .
EPOP POPULATION (100) OF E;
.
.
.
E DATA SET ( ...);
.
.
.
         );
```

Example 3–12. Using a DASDL Description for the POPULATION Attribute

The COBOL85 code for this description is as follows:

```
MOVE D (EPOP) TO X.
```

Manipulating Data in a Database

You can use the following data management statements to manipulate data in a database.

ABORT-TRANSACTION Statement

The ABORT-TRANSACTION statement discards any updates made in a transaction after a BEGIN-TRANSACTION statement, and removes a program from transaction state.

Format

```
ABORT-TRANSACTION  [ COMS-header-name-1 ]   data-name-1

[                  { imperative-statement-1   }  ]
|  ON EXCEPTION    { conditional-statement-1  }  |
[                  { NEXT SENTENCE            }  ]

[                      { imperative-statement-2   }  ]
|  NOT ON EXCEPTION     { conditional-statement-2  }  |
[                      { NEXT SENTENCE            }  ]

[ END-ABORT-TRANSACTION ]
```

Explanation

The ABORT-TRANSACTION statement backs out information that was updated after execution of the BEGIN-TRANSACTION statement, and removes the program from transaction state.

The optional COMS-header-name-1 phrase is used only with COMS. You can use COMS-header-name-1 to call the DCIENTRYPOINT of a data communications interface (DCI) library when your program detects an exception condition. This feature enables a program interfacing with COMS to support synchronized transactions and recovery.

COMS-header-name-1 specifies the COMS header. Your program calls the DCI library before it executes the exception-handling procedure. Refer to Section 2, “Using the COMS Program Interface,” for more information on COMS.

Data-name-1 is the name of a restart data set.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur. For more information about these clauses, refer to “DMSII Exceptions later in this section.

The END-ABORT-TRANSACTION phrase delimits the scope of the ABORT-TRANSACTION statement.

Example

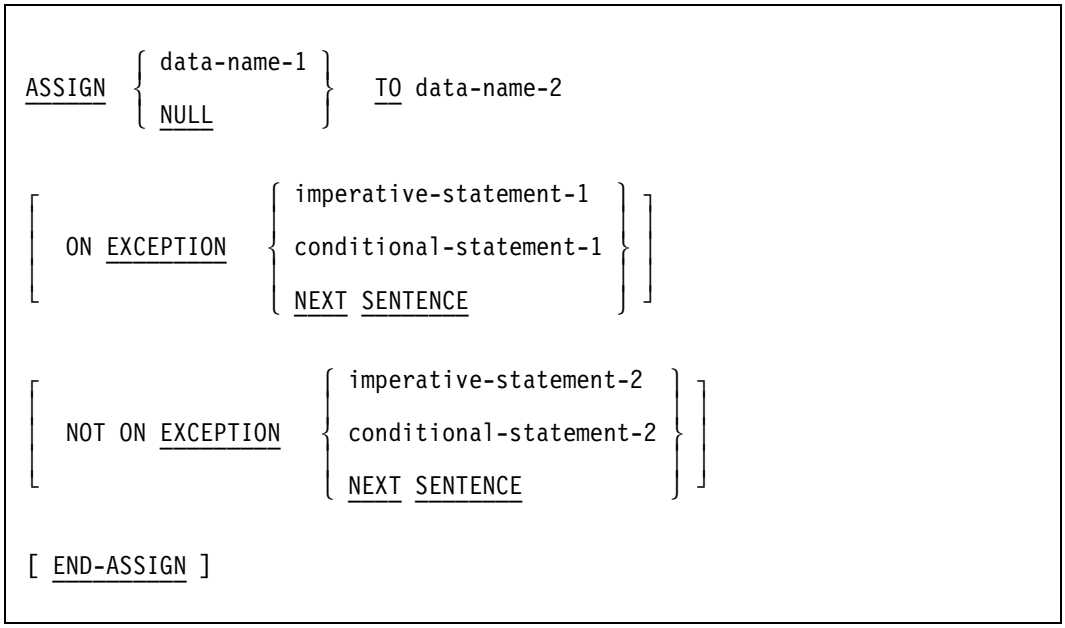
The following lines of code provide an example of the uses of the ABORT-TRANSACTION statement:

```
BEGIN-TRANSACTION NO-AUDIT RESTART-INFO
     ON EXCEPTION MOVE 6 TO ERROR-FLAG
                  PERFORM DM-ERROR-CHECK.
   .
   .
   .
ABORT-TRANSACTION RESTART-INFO
      ON EXCEPTION DISPLAY "ERROR IN ABORT TRANSACTION"
                           UPON CONSOLE
                  PERFORM DM-ERROR-CHECK.
```

ASSIGN Statement

The ASSIGN statement establishes the relationship between a record in a data set and a record in the same or another data set. The ASSIGN statement is effective immediately, so the second record does not need to be stored unless data items of this record have been modified.

Format



Explanation

If data-name-1 is a data set, you must declare it in the DASDL as the object data set of the link item data-name-2. Data-name-2 is a value that points to the current record in data-name-1.

The current path of the data set specified by data-name-1 must be valid, but the record need not be locked. Your program returns an exception if the data set path is not valid.

The NULL option severs the relationship between records by assigning a null value to data-name-2. If data-name-2 is already null, DMSII ignores this option. Executing a FIND, MODIFY, or LOCK statement on a null link item results in an exception.

If data-name-1 is a link item, it is assigned to data-name-2. You must declare data-name-1 in the DASDL according to the following requirements:

- It must have the same object data set as data-name-2.
- It must be the same type of link as data-name-2 (a counted link, a self-correcting link, a symbolic link, an unprotected link, or a verified link).

If the link item is a counted link, DMSII automatically updates the count item, even if the referenced record is locked by another program.

When the ASSIGN statement has been executed, data-name-2 points to either

- The current record in the data set specified by data-name-1, if data-name-1 is a data set
- The record to which data-name-1 points, if data-name-1 is a link item

Links can easily join unrelated records. However, they can also complicate the database as follows:

- Links must be maintained by a program. Other DMSII structures, such as automatic subsets, can do what links do but are maintained by the system.
- If you delete a record pointed to by several links, you might forget to remove all the links pointing to that record. As a result, the links would point to nothing.
- Links are one-way pointers to a record. Although you can find the record that a link is pointing to, you cannot easily find the record that is pointing to the linked record.

The current path of the data set that contains data-name-2 must be valid, and the record must be locked. Otherwise the program returns an exception.

If data-name-2 refers to a disjoint data set, data-name-2 can point to any record in the data set. If data-name-2 refers to an embedded data set, it can reference only certain records in the data set. In this case, the record being referenced must be owned by the record that contains data-name-2 or by an ancestor of the record that contains data-name-2. (An ancestor is the owner of the record, the owner of the owner, and so forth.)

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. If the program finds an exception, it terminates the ASSIGN statement, assigns a null value to data-name-2, and performs the instruction specified by the ON EXCEPTION clause.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-ASSIGN phrase delimits the scope of the ASSIGN statement.

Example

The following DASDL description used by the COBOL85 code in Example 3–13 is compiled with the name DBASE.

```
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;
```

Example 3–13 shows an example of the ASSIGN statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TAPE-FILE ASSIGN TO TAPE.
DATA DIVISION.
FILE SECTION.
FD  TAPE-FILE.
01  TAPE-REC.
    02  X PIC XXX.
    02  Y PIC 999.
DATA-BASE SECTION.
DB  DBASE ALL.
PROCEDURE DIVISION.
OPEN-INPUT-FILE.
    OPEN INPUT TAPE-FILE.
OPEN-DB.
    OPEN UPDATE DBASE.
START-PRG.
    READ TAPE-FILE AT END
          CLOSE TAPE-FILE
          CLOSE DBASE
          STOP RUN.
    FIND S AT A = X.
    FIND T AT N = Y.
    ASSIGN E TO L.
    FREE D.
    GO TO START-PRG.
```

Example 3–13. Using the ASSIGN Statement

An example of the ASSIGN statement within the context of a complete program is provided at line 011000 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

BEGIN-TRANSACTION Statement

The BEGIN-TRANSACTION statement places a program in transaction state. This statement can be used only with audited databases.

The BEGIN-TRANSACTION statement performs the following steps in order:

1. Captures the restart data set if the AUDIT clause is specified
2. Places a program in transaction state

Format

```
BEGIN-TRANSACTION  [ COMS-header-name-1  [ USING identifier-1 ] ]

    { AUDIT    }
    { NO-AUDIT } data-name-1

    [                 { imperative-statement-1  }  ]
    [ ON EXCEPTION    { conditional-statement-1 }  ]
    [                 { NEXT SENTENCE           }  ]

    [                     { imperative-statement-2  }  ]
    [ NOT ON EXCEPTION    { conditional-statement-2 }  ]
    [                     { NEXT SENTENCE           }  ]

[ END-BEGIN-TRANSACTION ]
```

Explanation

The optional COMS-header-name-1 phrase is used only with COMS. You can use COMS-header-name-1 to call the DCIENTRYPOINT of a DCI library when your program detects an exception condition. This feature enables a program interfacing with COMS to support synchronized transactions and recovery.

Your program calls the DCI library before it performs the exception-handling procedure. If your program does not detect an exception and you have employed the optional USING clause, your program calls the DCI library and passes the message area indicated by identifier-1 to the DCIENTRYPOINT.

COMS-header-name-1 specifies the input COMS header. Identifier-1 specifies the message area. For information on COMS, refer to Section 2, “Using the COMS Program Interface.”

The AUDIT clause captures the restart area. The path of the restart data set named is not altered when the restart record is stored. Either the AUDIT or NO-AUDIT clause must be specified.

The NO-AUDIT clause prevents the restart area from being captured.

Data-name-1 is the name of the restart data set you want to update.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. Your program returns an exception if you execute a BEGIN-TRANSACTION statement while the program is already in transaction state. If the program returns an exception, the program is not placed in transaction state. If the program returns an ABORT exception, all records that the program has locked are freed.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur. For more information about these clauses, refer to "DMSII Exceptions" later in this section.

The END-BEGIN-TRANSACTION phrase delimits the scope of the BEGIN-TRANSACTION statement.

Details

Any attempt to modify an audited database when the program is not in transaction state results in an audit error. The following data management verbs modify databases:

- ASSIGN
- DELETE
- GENERATE
- INSERT
- REMOVE
- STORE

Example

The following DASDL description used by the COBOL85 code in Example 3–14 is compiled with the name DBASE:

```
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;
```

Example 3–14 shows an example of the BEGIN-TRANSACTION statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TAPE-FILE ASSIGN TO TAPE.
DATA DIVISION.
FILE SECTION.
FD  TAPE-FILE.
01  TAPE-REC.
    02 X PIC 999.
    02 Y PIC XXX.
DATA-BASE SECTION.
DB  DBASE ALL.
WORKING-STORAGE SECTION.
01 CNT PIC 999.
PROCEDURE DIVISION.
OPEN-INPUT-FILE.
    OPEN INPUT TAPE-FILE.
OPEN-DB.
    OPEN UPDATE DBASE.
CREATE-D.
    CREATE D.
    ADD 1 TO CNT.
    MOVE CNT TO N.
    BEGIN-TRANSACTION  AUDIT R.
        STORE D.
    END-TRANSACTION NO-AUDIT R.
    IF CNT  < 100
        GO TO CREATE-D.
START-PRG.
    READ TAPE-FILE AT END
        CLOSE TAPE-FILE
        CLOSE DBASE
        STOP RUN.
    LOCK S AT N = X.
    BEGIN-TRANSACTION AUDIT R
        END-BEGIN-TRANSACTION.
        MOVE Y TO A.
        STORE D.
    END-TRANSACTION NO-AUDIT R.
    GO TO START-PRG.
```

Example 3–14. Using the BEGIN-TRANSACTION Statement

Examples of the BEGIN-TRANSACTION statement within the context of a complete program are provided in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

CANCEL TRANSACTION POINT Statement

The CANCEL TRANSACTION POINT statement discards all updates in a transaction back to an intermediate transaction point or to the beginning of the transaction without terminating the transaction state. The execution of the program continues with the statement following the CANCEL TRANSACTION POINT statement.

Format

```
CANCEL TRANSACTION POINT data-name-1  [ arithmetic-expression-1 ]

   [                  { imperative-statement-1  } ]
   [ ON EXCEPTION     { conditional-statement-1 } ]
   [                  { NEXT SENTENCE           } ]

   [                      { imperative-statement-2  } ]
   [ NOT ON EXCEPTION     { conditional-statement-2 } ]
   [                      { NEXT SENTENCE           } ]

[ END-CANCEL ]
```

Explanation

Data-name-1 is the name of a restart data set.

The CANCEL TRANSACTION POINT statement discards all database changes made between the current point in the transaction and the point specified by arithmetic-expression-1.

If you do not specify arithmetic-expression-1, DMSII discards all data updated since the BEGIN-TRANSACTION statement placed the program in transaction state. For details on arithmetic expressions, see Volume 1.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur. For more information about these clauses, refer to “DMSII Exceptions” later in this section.

The END-CANCEL phrase delimits the scope of the CANCEL TRANSACTION POINT statement.

Examples

The following lines of code provide examples of the uses of the CANCEL TRANSACTION POINT statement:

```
CANCEL TRANSACTION POINT MY-RESTART MAIN-SAVE-POINT.

CANCEL TRANSACTION POINT MY-RESTART.
```

CLOSE Statement

The CLOSE statement closes a database when your program requires no further access. The CLOSE statement is optional because the system closes any open database at the time the program terminates. A successfully closed database causes a syncpoint in the audit trail.

The CLOSE statement performs the following steps in order:

1. Closes the database
2. Frees all locked records

Format

```
CLOSE data-name-1
    [ ON EXCEPTION  { imperative-statement-1   } ]
                    { conditional-statement-1 }
                    { NEXT SENTENCE           }
    [ NOT ON EXCEPTION  { imperative-statement-2   } ]
                        { conditional-statement-2 }
                        { NEXT SENTENCE           }
[ END-CLOSE ]
```

Explanation

Data-name-1 specifies the database you want to close.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. Your program returns an exception if the specified database is not open.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about these clauses, refer to the “DMSII Exceptions” later in this section. Some specific information about exception handling is provided in the “Details” portion of this discussion.

Details

The CLOSE statement is the only statement in which the status word has meaning when no exception is indicated. Your program should, therefore, examine the status word after it closes a database and should take appropriate action, whether or not it received an exception. You can obtain an ABORT exception in this manner.

The CLOSE statement closes the database unconditionally, regardless of exceptions. If you use just the CLOSE syntax, the program is discontinued on any exceptions that raise the exception flag.

Your program does not return some exceptions when the CLOSE statement is used. To be sure your program detects any exceptions that occur during the execution of the CLOSE statement, do the following in the program code:

- Use the ON EXCEPTION clause to prevent the program from being discontinued if an exception flag is raised.
- Use an IF statement to check for exceptions that do not raise an exception flag.

The END-CLOSE phrase delimits the scope of the CLOSE statement.

If you are running COMS for synchronized recovery, it is recommended that you do not use the ON EXCEPTION clause. If DMSII detects a database error during the closing of a database, it should allow your program to terminate abnormally; otherwise, the database might abort recursively. If you use the ON EXCEPTION clause, you should ensure that your program calls the DMTERMINATE statement for those exceptions that your program does not handle. Use the following syntax, therefore, to close a database when you are using COMS with DMSII for synchronized recovery:

```
CLOSE DBASE.
```

Example

Example 3–15 shows the recommended syntax for the CLOSE statement when the ON EXCEPTION clause and the IF statement are used.

```
CLOSE MYDB
ON EXCEPTION
   DISPLAY "EXCEPTION WHILE CLOSING MYDB"
   CALL SYSTEM DMTERMINATE
   END-CLOSE.
IF DMSTATUS(DMERROR)
   OPEN MYDB
   GO TO ABORTED.
```

Example 3–15. Using the CLOSE Statement

An example of the CLOSE statement within the context of a complete program is provided at line 036700 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

COMPUTE Statement

The data management COMPUTE statement assigns a value to a Boolean item in the current record of a data set. The COMPUTE statement affects only the record area. The database is not affected until a subsequent STORE statement is executed.

No exceptions are associated with this statement.

Format

```
                          { condition }
COMPUTE data-name-1 =     { TRUE      }
                          { FALSE     }
```

Explanation

If you specify a condition, DMSII assigns the value of the condition to the specified Boolean item. The rules for the format of the condition are the same as the standard COBOL85 rules for the relation conditions.

The TRUE phrase assigns a TRUE value to the specified Boolean item.

The FALSE phrase assigns a FALSE value to the specified Boolean item.

Examples

The following lines of code provide two examples of the uses of the COMPUTE statement:

```
COMPUTE CLOSEFLAG = TRUE.

COMPUTE CHECKBALANCE = OLD-BALANCE + DEPOSIT EQUAL CURR-BALANCE
```

CREATE Statement

The CREATE statement initializes the user work area of a data set record.

The CREATE statement performs the following steps in order:

1. Frees the current record of the specified data set. Note that if the INDEPENDENTTRANS option in the DASDL is set, and the CREATE statement is issued during transaction state, the locked record is not freed until an END-TRANSACTION statement is executed.

   For more information on the INDEPENDENTTRANS option, refer to the DMSII DASDL Reference Manual.

2. Reads the specified expression to determine the format of the record to be created.
3. Initializes data items to one of the following:
   - The value of the INITIALVALUE option declared in the DASDL
   - The value of the NULL option declared in the DASDL
   - The default value of the NULL option, which is hexadecimal Fs

Format

```
CREATE data-name-1  [ ( expression ) ]

   [                  { imperative-statement-1   } ]
   [  ON EXCEPTION    { conditional-statement-1  } ]
   [                  { NEXT SENTENCE            } ]

   [                      { imperative-statement-2   } ]
   [  NOT ON EXCEPTION    { conditional-statement-2  } ]
   [                      { NEXT SENTENCE            } ]

[ END-CREATE ]
```

Explanation

Data-name-1 specifies the data set you want to initialize. The current path of the data set is unchanged until you execute a STORE statement.

The expression specifies the type of record you want to create. You must use an expression only to create a variable-format record; otherwise, the expression must not appear.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. Your program returns an exception if the expression does not represent a valid record type.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about these clauses, refer to “DMSII Exceptions” later in this section.

The END-CREATE phrase delimits the scope of the CREATE statement.

Details

You normally follow a CREATE statement with a STORE statement to place the newly created record into the data set. However, if you do not want to store the record, you can nullify the CREATE statement by executing a subsequent FREE statement or by using a FIND, LOCK, DELETE, CREATE, or RECREATE statement.

The CREATE statement only sets up a record area. If the record contains embedded structures, you must store the master record before you can create entries in the embedded structures. If you create only entries in the embedded structure (that is, if you do not alter items in the master), you need not store the master a second time.

Example

The following DASDL description used by the COBOL85 code in Example 3–16 is compiled with the name DBASE:

```
D DATA SET (
   A ALPHA (10) INITIALVALUE BLANKS;
   B BOOLEAN;
   N NUMBER (3) NULL 0;
   );
 S SET OF D KEY N;
```

Example 3–16 shows an example of the CREATE statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TAPE-FILE ASSIGN TO TAPE.
DATA DIVISION.
FILE SECTION.
FD  TAPE-FILE.
01  TAPE-REC.
    02  X PIC X(10).
    02  Y PIC 9.
    02  Z PIC 999.
DATA-BASE SECTION.
DB  DBASE ALL.
PROCEDURE DIVISION.
OPEN-INPUT-FILE.
    OPEN INPUT TAPE-FILE.
OPEN-DB.
    OPEN UPDATE DBASE.
START-PRG.
    READ TAPE-FILE AT END
       CLOSE TAPE-FILE
       CLOSE DBASE
       STOP RUN.
    CREATE D.
    MOVE X TO A.
    IF Y = 1
    COMPUTE B = TRUE.
    MOVE Z TO N.
    STORE D.
    GO TO START-PRG.
```

Example 3–16. Using the CREATE Statement

Examples of the CREATE statement within the context of a complete program are provided in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

DELETE Statement

The DELETE statement finds a record by a method identical to that of the FIND statement. However, the FIND statement transfers the record to the user work area associated with a data set or global data, whereas the DELETE statement performs the following steps in order:

1. Frees the current record, unless the selection expression is the name of the data set and the current record is locked. In this case, the locked status is not altered.
2. Alters the current path to point to the record specified by the selection expression and locks this record.
3. Transfers that record to the user work area.
4. Removes the record from all sets and automatic subsets, but not from manual subsets.
5. Removes the record from the data set.

If your program finds a record that cannot be deleted, your program returns an exception and terminates the DELETE statement, leaving the current path pointing to the record specified by the selection expression.

If you use a set selection expression and your program cannot find the record, an exception is returned, and the program changes and invalidates the set path. The selection expression refers to a location between the last key less than the condition and the first key greater than the condition. You can execute a set selection expression by using the NEXT or PRIOR clause from this location, provided keys greater than or 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.

Format

```
DELETE selection-expression-1

   [                  { imperative-statement-1   }  ]
   |  ON EXCEPTION    { conditional-statement-1  }  |
   [                  { NEXT SENTENCE            }  ]

   [                      { imperative-statement-2   }  ]
   |  NOT ON EXCEPTION    { conditional-statement-2  }  |
   [                      { NEXT SENTENCE            }  ]

[ END-DELETE ]
```

Explanation

Selection-expression-1 identifies the record you want to delete. Selection expressions are explained in "Using Selection Expressions" earlier in this section.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. Your program returns an exception and does not delete the record if one of the following conditions is true:

- The counted links are pointing to the record.
- The record contains a nonnull link or an embedded structure that contains entries.

Your program also returns an exception if the record exists in a manual subset. Refer to "REMOVE Statement" later in this section.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to "DMSII Exceptions" later in this section.

The END-DELETE phrase delimits the scope of the DELETE statement.

Details

When the DELETE statement is completed, the current paths still refer to the deleted record. Although a FIND statement on the current record results in a NOTFOUND exception, the FIND NEXT and FIND PRIOR statements yield valid results.

Example

The following DASDL description used by the COBOL85 code in Example 3–17 is compiled with the name DBASE:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
 S SET OF D KEY N;
```

Example 3–17 shows an example of coding for the DELETE statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TAPE-FILE ASSIGN TO TAPE.
DATA DIVISION.
FILE SECTION.
FD  TAPE-FILE.
01  TAPE-REC.
    02  X PIC 999.
DATA-BASE SECTION.
DB  DBASE ALL.
PROCEDURE DIVISION.
OPEN-INPUT-FILE.
    OPEN INPUT TAPE-FILE.
OPEN-DB.
    OPEN UPDATE DBASE.
START-PRG.
    READ TAPE-FILE AT END
        CLOSE TAPE-FILE
        CLOSE DBASE
        STOP RUN.
    DELETE S AT N = X.

    GO TO START-PRG.
```

Example 3–17. Using the DELETE Statement

An example of the DELETE statement within the context of a complete program is provided at line 032800 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

DMTERMINATE Statement

The DMTERMINATE statement terminates programs. When an exception occurs that the program does not handle, the DMTERMINATE statement terminates the program with a fault.

Format

```
CALL SYSTEM DMTERMINATE
```

Example

Example 3–18 shows an example of coding for the DMTERMINATE statement. An example of the DMTERMINATE statement within the context of a complete program is provided at line 019000 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

```
FIND FIRST D
 ON EXCEPTION
   DISPLAY "D IS EMPTY DATA SET"
   CALL SYSTEM DMTERMINATE.
```

Example 3–18. Using the DMTERMINATE Statement

END-TRANSACTION Statement

The END-TRANSACTION statement takes a program out of transaction state. You can use this statement only with audited databases. The END-TRANSACTION statement performs the following steps in order:

1. Captures the restart area if the AUDIT clause is specified
2. Forces a syncpoint if the SYNC option is specified
3. Implicitly frees all records of the database that the program has locked

If an exception occurs, this transaction is not applied to the database. Exceptions are discussed under the explanation for this transaction and later in this section under "DMSII Exceptions."

Format

```
END-TRANSACTION [ COMS-header-1  [ USING identifier-1 ] ]

    { AUDIT    } data-name-1    [ SYNC ]
    { NO-AUDIT }

[                  { imperative-statement-1   } ]
[  ON EXCEPTION    { conditional-statement-1  } ]
[                  { NEXT SENTENCE            } ]

[                      { imperative-statement-2   } ]
[  NOT ON EXCEPTION    { conditional-statement-2  } ]
[                      { NEXT SENTENCE            } ]

[ END-END-TRANSACTION ]
```

Explanation

The optional COMS-header-name phrase is used only with COMS. You can specify COMS-header-name-1 to call the DCIENTRYPOINT of a DCI library whenever you execute the statement. This feature enables a program interfacing with COMS to support synchronized transactions and recovery.

COMS-header-name specifies the COMS output header. For information on COMS, refer to Section 2, "Using the COMS Program Interface."

When your program detects an exception condition, your program calls the DCI library before it performs any exception-handling procedures.

If you employ the optional USING clause, your program calls the DCI library and passes the message area indicated by identifier-1 to the DCIENTRYPOINT.

The AUDIT clause captures the restart area. Storing the restart record does not alter the path of the restart data set. The NO-AUDIT clause prevents the restart area from being captured. You must specify either AUDIT or NO-AUDIT.

You can use the SYNC option to force a syncpoint.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. Your program returns an exception if you execute an END-TRANSACTION statement when the program is not in transaction state.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to "DMSII Exceptions" later in this section.

The END-END-TRANSACTION phrase delimits the scope of the END-TRANSACTION statement.

Example

The following DASDL description used by the COBOL85 code in Example 3–19 is compiled with the name DBASE:

```
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;
```

Example 3–19 shows two sections of code, each of which begins with a BEGIN-TRANSACTION statement and ends with an END-TRANSACTION statement. Both sections of code define a transaction. The transaction becomes an indivisible, logical unit. During processing, the transactions are audited for recovery. The AUDIT and NO-AUDIT phrases determine whether the restart record of the data set is captured.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TAPE-FILE ASSIGN TO TAPE.
```

```
DATA DIVISION.
  FILE SECTION.
  FD  TAPE-FILE.
  01  TAPE-REC.
      02  X PIC 999.
      02  Y PIC XXX.
  DATA-BASE SECTION.
  DB  DBASE ALL.
  WORKING-STORAGE SECTION.
  01 CNT PIC 999.
 PROCEDURE DIVISION.
  OPEN-INPUT-FILE.
      OPEN INPUT TAPE-FILE.
  OPEN-DB.
      OPEN UPDATE DBASE.
 CREATE-D.
      CREATE D.
      ADD 1 TO CNT.
      MOVE CNT TO N.
      BEGIN-TRANSACTION AUDIT R.
          STORE D.
      END-TRANSACTION AUDIT R.
      IF CNT  < 100
          GO TO CREATE-D.
 START-PRG.
      READ TAPE-FILE AT END
          CLOSE TAPE-FILE
          CLOSE DBASE
          STOP RUN.
  LOCK S AT N = X.
  BEGIN-TRANSACTION AUDIT R.
      MOVE Y TO A.
      STORE D.
  END-TRANSACTION  NO-AUDIT R
  END-END-TRANSACTION.
  GO TO START-PRG.
```

Example 3–19. Using the END-TRANSACTION Statement

Examples of the END-TRANSACTION statement within the context of a complete program are provided in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

FIND Statement

The FIND statement transfers a record to the user work area associated with a data set or global data. Additional information on the use of the FIND statement with the REBLOCK and READAHEAD options is provided in the *DMSII Application Programming Guide*.

The FIND statement performs the following steps in order:

1. Frees a locked record in the data set if you specify a data set in the FIND statement. Specifying a set in the FIND statement frees a locked record in the associated data set.
2. Alters the current path to point to the record specified by the selection expression or the database name.
3. Transfers that record to the user work area.

Using the FIND statement does not prevent other transactions from reading the record before the current update transaction is completed.

Format

```
{ FIND { selection-expression-1 }                                    }
{      { data-name-1            }                                    }
{                                                                    }
{ FIND KEY OF selection-expression-2                                 }

   [                  { imperative-statement-1  }                ]
   [ ON EXCEPTION     { conditional-statement-1 }                ]
   [                  { NEXT SENTENCE           }                ]

   [                      { imperative-statement-2  }            ]
   [ NOT ON EXCEPTION     { conditional-statement-2 }            ]
   [                      { NEXT SENTENCE           }            ]

[ END-FIND ]
```

Explanation

Selection-expression-1 specifies the record that you want to transfer to the user work area.

Data-name-1 specifies the global data record that you want to transfer to the user work area associated with the global data. If no global data is described in the DASDL, DMSII returns a syntax error.

The FIND KEY OF clause moves the key and any associated data (as specified in the DASDL) from the key entry to the user work area. Your program does not perform a physical read on the data set; consequently, the value and contents of all items in the record area that do not appear in the key entry retain whatever value they had before you executed the FIND KEY OF clause. The FIND statement does not affect the current path of the data set.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. Your program returns an exception if no record satisfies the selection expression.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

Details

If you use selection-expression-2 and your program fails to find the record, the program returns an exception and changes and invalidates the set path. The selection expression refers to a location in between the last key less than the condition and the first key greater than the condition. You can execute selection-expression-2 by using NEXT or PRIOR from this location, provided that the keys greater than or 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.

The END-FIND phrase delimits the scope of the FIND statement.

Examples

The following examples illustrate options for using the FIND statement. The first example shows the use of a set selection expression.

```
FIND FIRST OVER-65 AT DEPT-NO = 1019
ON EXCEPTION
   MOVE 0 TO POP-OVR-65 (1019).
```

This example shows the FIND statement used with a FIND KEY OF clause.

```
FIND KEY OF NAME-KEYS AT NAME = "FRED JONES".
```

An example of the FIND statement within the context of a complete program is provided at line 026600 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

FREE Statement

The FREE statement explicitly unlocks the current record or structure. A FREE statement executed on a record enables other programs to lock that record or structure.

Note that if you set the INDEPENDENTTRANS option in the DASDL for the database, the program ignores a FREE statement during transaction state. For more information on the INDEPENDENTTRANS option, refer to the *DMSII DASDL Reference Manual*.

You can execute a FREE statement after any operation. If the current record or structure is already free, or if no current record or structure is present, the program ignores the FREE statement.

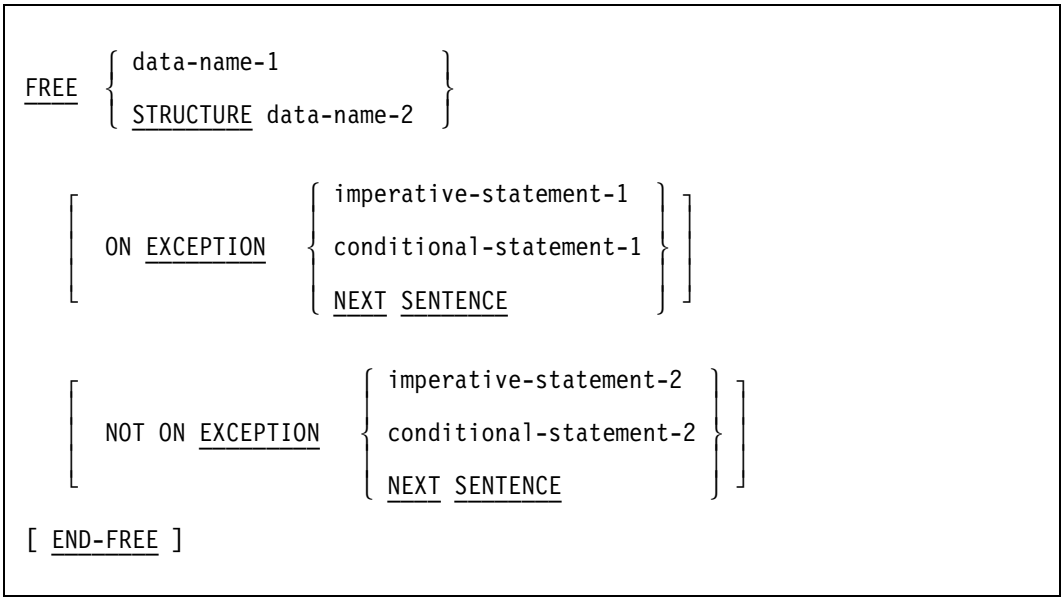
You can use the FREE statement to unlock a record or structure that you anticipate will not be implicitly freed for a long time.

The FREE statement is optional in some situations because the FIND, LOCK, MODIFY, and DELETE statements can free a record before they execute. Generally, an implicit FREE operation is performed, if needed, during any operation that establishes a new data set path.

FIND, LOCK, and MODIFY statements that use sets or subsets free the locked record only if a new record is retrieved. Other constructs that free data set records are

```
BEGIN-TRANSACTION          RECREATE
CREATE                     SET-TO-BEGINNING
END-TRANSACTION            SET-TO-ENDING
```

Format



Explanation

Data-name-1 specifies either the data set whose current record is to be unlocked or the global data record to be unlocked. The data set path and current record area remain unchanged. You can use the database name as a synonym to free the global data record.

Data-name-2 specifies the structure to be freed. The STRUCTURE option frees all records in the structure.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. If the program returns an exception, the state of the database remains unchanged.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-FREE phrase delimits the scope of the FREE statement.

Example

Example 3–20 shows an example of the FREE statement.

```
LOCK NEXT S
ON EXCEPTION
   GO TO NO-MORE.
IF ITEM-1 NOT = VALID-VALUE
   FREE DS
   GO ERR.
```

Example 3–20. Using the FREE Statement

GENERATE Statement

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. Deletes all records from the subset if the subset is not empty
2. Assigns to the generated subset the records in another subset, a combination of the records in two other subsets, or null values

Note: *It is recommended that you coordinate any subset declaration with other users because subsets can be used concurrently and altered without your knowledge.*

Format

```
                          { NULL                                     }
                          {                 [ { AND }              ] }
GENERATE data-name-1 =    {                 [ { OR  }              ] }
                          { data-name-2     [ { +   }  data-name-3 ] }
                          {                 [ { -   }              ] }

   [                  { imperative-statement-1  } ]
   [  ON EXCEPTION    { conditional-statement-1 } ]
   [                  { NEXT SENTENCE           } ]

   [                      { imperative-statement-2  } ]
   [  NOT ON EXCEPTION    { conditional-statement-2 } ]
   [                      { NEXT SENTENCE           } ]

[ END-GENERATE ]
```

Explanation

Data-name-1 is the name of the subset you want to generate. Data-name-1 must refer to a manual subset and must be a disjoint bit vector.

The NULL option assigns a null value to the generated subset so that the subset remains empty.

Data-name-2 is the name of the subset you want to assign to data-name-1. The data-name-2 subset must be of the same data set as the data-name-1 subset and must be a disjoint bit vector.

Data-name-3 is the name of the subset you want to combine with data-name-2 and assigned to data-name-1. The data-name-3 subset must be of the same data set as the data-name-2 subset and must be a disjoint bit vector.

The AND operator assigns the intersection of data-name-2 and data-name-3 to data-name-1. The intersection is defined to be all the records in data-name-2 that are also in data-name-3.

The OR operator assigns the union of data-name-2 and data-name-3 to data-name-1. The union is defined to be all the records that are in either data-name-2 or data-name-3.

The plus (+) operator is the subset-exclusive OR. This operator assigns the records contained in either data-name-2 or data-name-3 (but not both) to data-name-1.

The minus (-) operator is the subset difference. This operator assigns the records contained in data-name-2 that are not in data-name-3 to data-name-1.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur. For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-GENERATE phrase delimits the scope of the GENERATE statement.

Example

The following DASDL description used by the COBOL85 code in Example 3–21 is compiled with the name DBASE:

```
DASDL (compiled as DBASE):
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;
```

Example 3–21 shows an example of coding for the GENERATE statement.

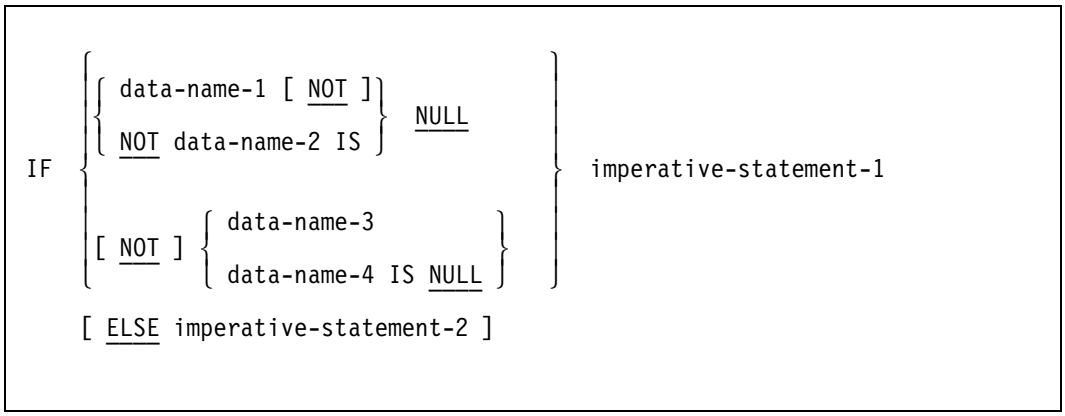
```
 IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TAPE-FILE ASSIGN TO TAPE.
 DATA DIVISION.
 FILE SECTION.
 FD  TAPE-FILE.
 01  TAPE-REC.
     02  S PIC XXX.
     02  T PIC 9.
     02  U PIC 999.
     02  V PIC 9(4).
 DATA-BASE SECTION.
 DB  DBASE ALL.
 PROCEDURE DIVISION.
 OPEN-INPUT-FILE.
     OPEN INPUT TAPE-FILE.
 OPEN-DB.
     OPEN UPDATE DBASE.
 START-PRG.
     READ TAPE-FILE AT END
     CLOSE TAPE-FILE
     GO TO GENERATE-SUBSET.
     CREATE D.
     MOVE S TO A.
     IF T = 1
         COMPUTE B = TRUE.
     MOVE U TO N.
     MOVE V TO R.
     STORE D.
     GO TO START-PRG.
 GENERATE-SUBSET.
     GENERATE Z = X AND Y.
     CLOSE DBASE.
     STOP RUN.
```

Example 3–21. Using the GENERATE Statement

IF Statement

The IF statement for DMSII tests an item to determine if it contains a NULL value.

Format



Explanation

Data-name-1 and data-name-2 are items you want to test. Data-name-3 specifies a Boolean item declared in the DASDL specification.

Your program executes imperative-statement-1 if the condition you are testing in the IF statement is satisfied. If the condition is not satisfied, imperative-statement-2 is executed.

The NULL option is the null value defined in the DASDL. The NULL clause specifies a condition that can also appear in combined conditions. Refer to Volume 1 for information on complex conditions.

Data-name-4 specifies a link item declared in the DASDL specification. The specified link item contains a null value if

- The link item does not point to a record.
- No current record is present for the data set that contains the link item. This condition occurs following OPEN, SET TO BEGINNING, or SET TO ENDING statements, or when the record containing the link item has been deleted.
- A version error would result from using a DMVERB against the structure into which the link item points.

  Both the structure in which link items are declared and the structure into which they point are accessed when the link items are tested. If either of the structures have been reorganized, the program can receive a version error.

  With declaration data sets, version errors are usually detected prior to the test for NULL, because the contents are considered NULL if there is no current record.

  However, a data set touched by a DMVERB returns a version error. If this data set is tested for NULL, the test is considered to be NULL. The data set cannot return a version error because the NULL test can only return a Boolean value.

The data-name-4 link item contains a nonnull value if it points to a record, even if that record has been deleted.

Data items declared in the DASDL, besides being used in the NULL test, can also be used in standard COBOL85 relation conditions, exactly like data items declared in a COBOL85 program.

Example

The following example illustrates the use of the NULL option with the IF statement:

```
IF THE-ITEM IS NULL
     PERFORM NEVER-USED.
```

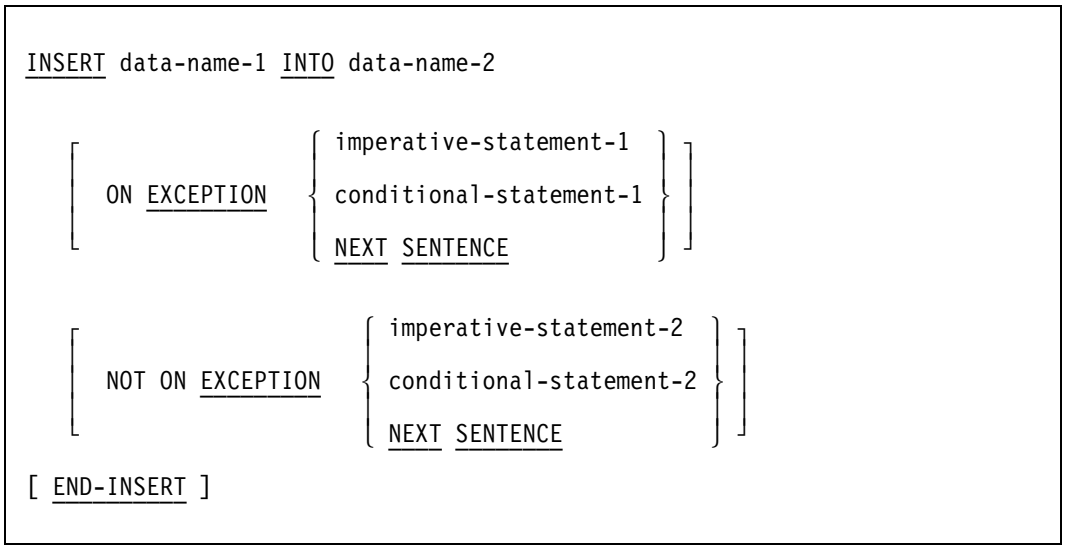
Examples of the IF statement within the context of a complete program are provided in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

INSERT Statement

The INSERT statement places a record into a manual subset. The statement performs the following steps in order:

1. Inserts the current record of the specified data set into the specified subset
2. Alters the set path for the specified subset to point to the inserted record

Format



Explanation

Data-name-1 identifies the data set whose current record you want to insert into the subset specified by data-name-2. Data-name-1 must be the object data set of the specified subset. The path of data-name-1 must refer to a valid record; otherwise, the program returns an exception.

Data-name-2 must specify a manual subset of the data set specified by data-name-1.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The program returns an exception if one of the following occurs:

- The subset you specified does not permit duplicates, and the record you want to insert has a key identical to that of a record currently in the specified subset.
- The specified subset is embedded in a data set, and the data set does not have a valid current record.
- The LOCK TO MODIFY DETAILS option was specified in the DASDL, and the current record is not locked.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-INSERT phrase delimits the scope of the INSERT statement.

Example

The following DASDL description used by the COBOL85 code in Example 3–22 is compiled with the name DBASE:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
  X SUBSET OF D BIT VECTOR;
```

Example 3–22 shows an example of coding for the INSERT statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
DATA DIVISION.
DATA-BASE SECTION.
DB  DBASE ALL.
PROCEDURE DIVISION.
OPEN-DB.
    OPEN UPDATE DBASE.
    SET D TO BEGINNING.
START-PRG.
    FIND NEXT D ON EXCEPTION
        CLOSE DBASE
        STOP RUN.
    IF N > 10
        INSERT D INTO X.
    GO TO START-PRG.
```

Example 3–22. Using the INSERT Statement

LOCK/MODIFY Statement

The LOCK statement finds a record in a manner identical to that of the FIND statement, except that a found record is locked against a concurrent modification by another user. LOCK and MODIFY are synonyms. This statement also provides the STRUCTURE option, which simultaneously locks all records in a structure.

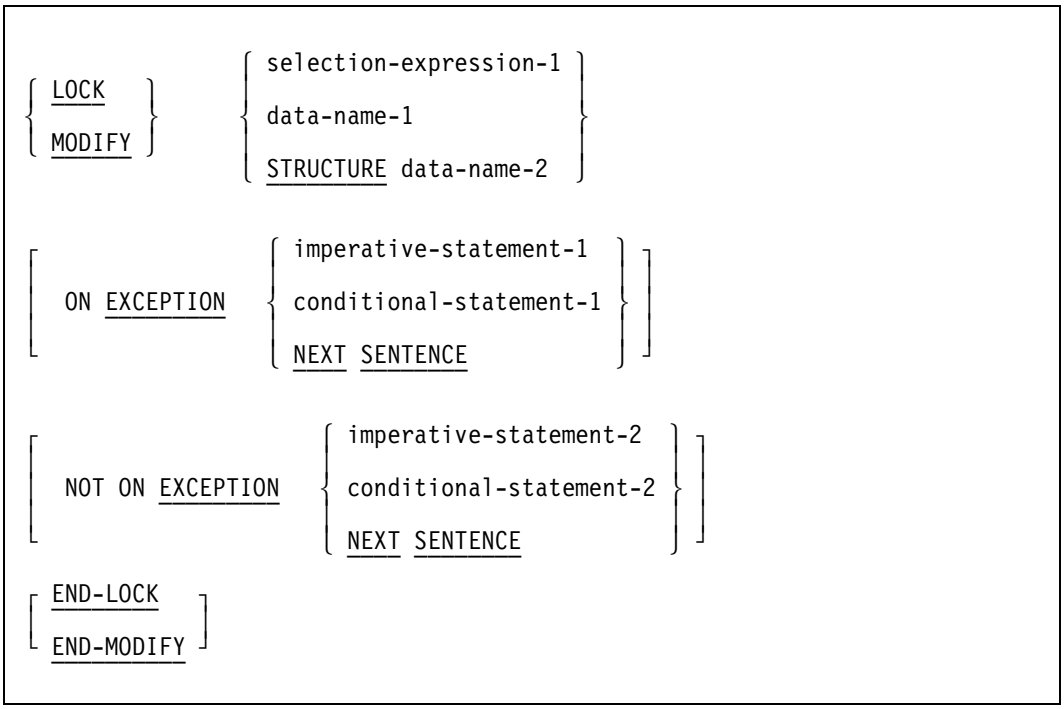
If the record to be locked has already been locked by another program, the system performs a contention analysis. The present program waits until the other program unlocks the record unless the wait would result in a deadlock. If a deadlock would result, the DMSII access routines unlock all records locked by the program that has the lowest priority of the programs involved in the deadlock, and returns a DEADLOCK exception to the program of lower priority whose records were unlocked.

No other user can lock or secure the record once it is locked; therefore, the record must be freed when locking is no longer required. You can free a record explicitly by using a FREE statement, or implicitly by executing a subsequent LOCK, FIND, DELETE, CREATE, or RECREATE statement on the same data set.

The LOCK/MODIFY statement performs the following steps in order:

1. Implicitly frees a locked record. However, if you have set the INDEPENDENTTRANS option in the DASDL, the LOCK/MODIFY statements do not free the locked record until you execute an END-TRANSACTION statement.
2. Alters the current path to point to the record specified by the selection expression or data name included in the statement.
3. Locks the specified record.
4. Transfers that record to the user work area.

Format



Explanation

Selection-expression-1 specifies the record you want to lock. Data-name-1 specifies the global data record you want to lock. If you specify the STRUCTURE option, data-name-2 must be a data set.

The STRUCTURE option locks or secures all records in the structure simultaneously. If other users have locked or secured the structure, or records in the structure, you must wait until those users free the records or the structure, or end their transactions. A deadlock occurs when other users attempt to lock or secure more records while you are locking the structure. Once you have locked a structure, you must continue to lock individual records. Each new lock implicitly frees the previous record, even if you have set the INDEPENDENTTRANS option in the DASDL. These freed records continue to be available only to the user who is securing the structure.

You cannot free structure locks with an END-TRANSACTION statement. You must use a FREE statement to free structure locks. Information is provided under “FREE Statement earlier is this section.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The program returns an exception if no record satisfies the selection expression. If the program returns an exception, the record is not freed. A DEADLOCK exception occurs if the program waits on a LOCK statement for a time longer than that specified in the MAXWAIT task attribute. For more information about the MAXWAIT attribute, refer to the *Task Attributes Programming Reference Manual*.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

An END-LOCK or an END-MODIFY phrase delimits the scope of the LOCK/MODIFY statement.

If a LOCK statement using a selection expression returns an exception, the program invalidates the current path of the specified set. 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.

Examples

The following line of code shows the LOCK statement used with the STRUCTURE option:

```
LOCK STRUCTURE VENDOR-DATA.
```

Example 3–23 shows the LOCK statement used with the ON EXCEPTION clause.

```
LOCK FIRST EMP AT DEPT-NO = 1019
ON EXCEPTION
   MOVE 0 TO POP-EMP (1019)
END-LOCK.
```

Example 3–23. Using the LOCK Statement with the ON EXCEPTION Clause

Example 3–24 shows the MODIFY statement used with the ON EXCEPTION clause.

```
MODIFY EMP AT EMP-NO = IN-SSN
 ON EXCEPTION
    MOVE INV-EMP-NO-ERR TO ERR-MSG
    PERFORM ERR-OUT.
```

Example 3–24. Using the MODIFY Statement with the ON EXCEPTION Clause

An example of the LOCK statement within the context of a complete program is provided at line 031000 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

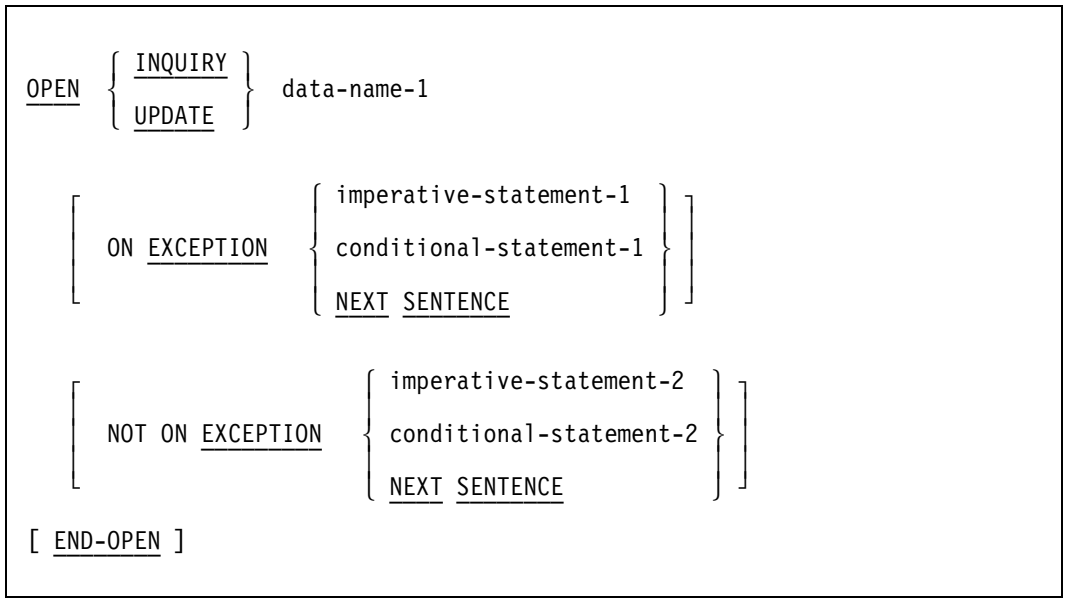
OPEN Statement

The OPEN statement opens a database for subsequent access and specifies the access mode. You must execute an OPEN statement before the database is first accessed; otherwise, the program terminates at run time with an invalid operator fault.

The OPEN statement performs the following steps in order:

1. Opens an existing database. If files required for invoked structures are not in the system directory, DMSII displays an informative message.
2. Performs an implicit CREATE operation on the restart data set.

Format



Explanation

The INQUIRY option enforces read-only access to the database specified by data-name-1. Use this option when you do not want to update the database. The UPDATE option enables you to modify the database specified by data-name-1. When you use the following verbs, the program returns an exception after opening the database with the INQUIRY option. You must specify UPDATE to use these verbs:

```
ASSIGN                      GENERATE
BEGIN-TRANSACTION           INSERT
DELETE                      REMOVE
END-TRANSACTION             STORE
```

DMSII does not open any audit files if OPEN INQUIRY has been specified by all programs that access the database.

Data-name-1 specifies the database to be opened.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The program returns an exception if the database is already open. If the program returns an exception, the state of the database remains unchanged.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-OPEN phrase delimits the scope of the OPEN statement.

Examples

The following example illustrates the use of the OPEN statement with the INQUIRY option:

```
  OPEN INQUIRY DBASE.
```

Example 3–25 shows an example of the OPEN statement with the INQUIRY option and an ON EXCEPTION clause.

```
OPEN INQUIRY MYDB
ON EXCEPTION
    DISPLAY "EXCEPTION OPENING MYDB"
    CALL SYSTEM DMTERMINATE
END-OPEN.
```

Example 3–25. Using the OPEN Statement with the INQUIRY Option

An example of the OPEN statement within the context of a complete program is provided at line 018900 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2 .

RECREATE Statement

The RECREATE statement partially initializes the user work area. Although it does not alter any data items, the RECREATE statement unconditionally sets control items such as links, sets, counts, and data sets to null values.

This statement performs the following steps in order:

1. Frees the current record of the specified data set
2. Reads any specified expression to determine the format of the record to be created
3. Unconditionally sets links, sets, counts, and data sets to null values

To re-create variable-format records, you must supply the same record type as that supplied in the original CREATE statement. If you do not, the subsequent STORE statement results in a DATAERROR subcategory 4. Refer to the *DMSII Application Programming Guide* for more information.

Format

```
RECREATE  data-name-1  [ ( expression ) ]

   [                  { imperative-statement-1   } ]
   [  ON EXCEPTION    { conditional-statement-1  } ]
   [                  { NEXT SENTENCE            } ]

   [                     { imperative-statement-2   } ]
   [  NOT ON EXCEPTION   { conditional-statement-2  } ]
   [                     { NEXT SENTENCE            } ]

[ END-RECREATE ]
```

Explanation

Data-name-1 is the name of the data set you want to initialize.

The expression specifies the value of the type of record you want to create. You must use an expression to create a variable-format record; otherwise, the expression must not appear.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The program returns an exception if the expression does not represent a valid record type.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-RECREATE phrase delimits the scope of the RECREATE statement.

Example

The following DASDL description is used by the COBOL85 code in Example 3–26. The description is compiled with the name DBASE:

```
D DATA SET (
   A ALPHA (3);
   N NUMBER (3);
   );
S SET OF D KEY N;
```

Example 3–26 shows an example of coding for the RECREATE statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
DATA DIVISION.
DATA-BASE SECTION.
DB  DBASE ALL.
PROCEDURE DIVISION.
START-PRG.
    OPEN UPDATE DBASE.
    CREATE D.
    MOVE "ABC" TO A.
    MOVE 1 TO N.
    STORE D.
RECREATE-D.
    RECREATE D.
    ADD 1 TO N.
    STORE D.
    IF N < 500
        GO TO RECREATE-D
    ELSE
        CLOSE DBASE
        STOP RUN.
```

Example 3–26. Using the RECREATE Statement

REMOVE Statement

The REMOVE statement is similar to the FIND statement except that a found record is locked and then removed from the specified subset.

The REMOVE statement performs the following steps in this order:

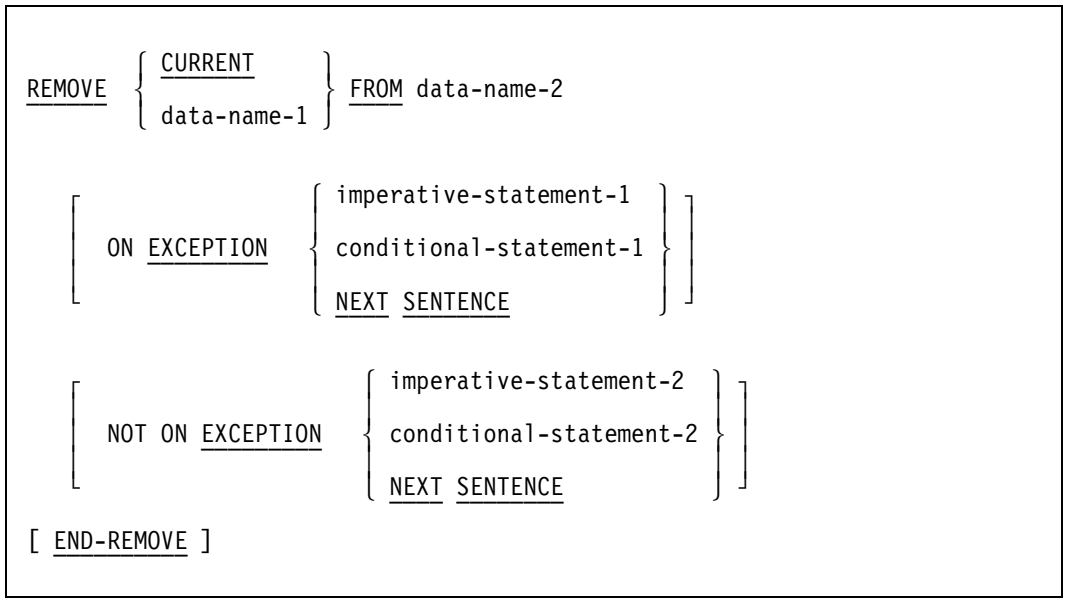
1. Frees the current record
2. Alters the current path to point to the record specified by the CURRENT phrase or the data set name
3. Locks the previously found record
4. Removes the record from the specified subset

If the program returns an exception after step 2, the current path is invalid.

If the program returns an exception after step 3, the operation terminates, leaving the current path pointing to the record specified by CURRENT or by data-name-1.

When the REMOVE statement is completed, the current paths still refer to the deleted record. As a result, a FIND statement on the current record results in a NOTFOUND exception, although FIND NEXT and FIND PRIOR statements give valid results.

Format



Explanation

The CURRENT option removes the current record from the subset specified by data-name-2. If you specify this option, the subset must have a valid current record. If it does not have a valid current record, the program returns an exception.

Data-name-1 is the name of the data set. Data-name-1 finds the record located by the current path and removes it from the subset. The program returns an exception if the record is not in the subset.

Data-name-2 specifies the subset from which you want to remove a record. Data-name-2 must specify a manual subset of the data set specified by data-name-1.

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 it is not locked, the program returns an exception.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. An exception is returned if one of the following occurs:

- You specify the CURRENT option, and the specified subset does not have a valid current record.
- You specify data-name-1, and the record is not in the subset.
- The subset you specified is embedded in a data set, and the data set does not have a current record defined and locked.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-REMOVE phrase delimits the scope of the REMOVE statement.

Example

The following DASDL description used by the COBOL85 code in Example 3–27 is compiled with the name DBASE:

```
D DATA SET (
  A ALPHA (3);
  B BOOLEAN;
  N NUMBER (3);
  R REAL;
  );
X SUBSET OF D BIT VECTOR;
```

Example 3–27 shows an example of coding for the REMOVE statement.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. EXAMPLE.
 DATA DIVISION.
 DATA-BASE SECTION.
 DB  DBASE ALL.
 PROCEDURE DIVISION.
 OPEN-DB.
     OPEN UPDATE DBASE.
     SET X TO BEGINNING.
 START-PRG.
     FIND NEXT X ON EXCEPTION
         CLOSE DBASE
         STOP RUN.
     IF N > 100
         REMOVE D FROM X.
     GO TO START-PRG.
```

Example 3–27. Using the REMOVE Statement

SAVE TRANSACTION POINT Statement

The SAVE TRANSACTION POINT statement provides an intermediate transaction point record for auditing. The transaction points apply only to the current transaction, and do not affect halt/load recovery. The system completes halt/load recovery at the end of the transaction, but not when it encounters a transaction point.

Format

```
SAVE TRANSACTION POINT data-name-1  [ arithmetic-expression-1 ]

    [                    { imperative-statement-1  }  ]
    [  ON EXCEPTION      { conditional-statement-1 }  ]
    [                    { NEXT SENTENCE           }  ]

    [                      { imperative-statement-2  }  ]
    [  NOT ON EXCEPTION    { conditional-statement-2 }  ]
    [                      { NEXT SENTENCE           }  ]

[ END-SAVE ]
```

Explanation

Data-name-1 is the name of a restart data set that identifies the database.

Arithmetic-expression-1 indicates a marker to be assigned to the present execution point in the transaction. Arithmetic expressions are discussed in Volume 1.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur. For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-SAVE phrase delimits the scope of the SAVE TRANSACTION POINT statement.

Example

The following line of code shows an example of the SAVE TRANSACTION POINT statement:

```
SAVE TRANSACTION POINT MY-RESTART 3.
```

SECURE Statement

The SECURE statement prevents other programs from updating a record by applying a shared lock. A shared lock allows other users to find or secure a record; however, they cannot include the record in a LOCK statement.

You can execute a LOCK statement to upgrade secured records to locked records. If two or more users try to upgrade the records at the same time, however, a deadlock can occur and cause an exception.

Format

```
SECURE { selection-expression-1 }
       { STRUCTURE data-name-1  }

[                  { imperative-statement-1   } ]
[  ON EXCEPTION    { conditional-statement-1  } ]
[                  { NEXT SENTENCE            } ]

[                     { imperative-statement-2   } ]
[  NOT ON EXCEPTION   { conditional-statement-2  } ]
[                     { NEXT SENTENCE            } ]

[ END-SECURE ]
```

Explanation

Selection-expression-1 specifies the record you want to secure. For more information, see "Using Selection Expressions" earlier in this section.

Data-name-1 specifies the global data record you want to secure. If the invoked database contains a remap of the global data, your program uses the name of the logical database, not the name of the global data remap, to lock the global data record.

If you use the STRUCTURE option, data-name-1 specifies the structure to be secured. The structure must be a data set. The STRUCTURE option secures all records in the structure simultaneously. If other users have locked records in the structure, you must wait until they free the records or end their transactions before you can secure the structure. A deadlock can occur if other users attempt to lock more records while you are securing the structure.

Ending the transaction does not free a secured structure; instead, you must use the FREE statement. More information on this statement is provided under "FREE Statement" earlier in this section.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur. For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

Example

The following line of code shows an example of the SECURE statement used with the STRUCTURE option:

```
SECURE STRUCTURE VENDOR-DATA.
```

SET Statement

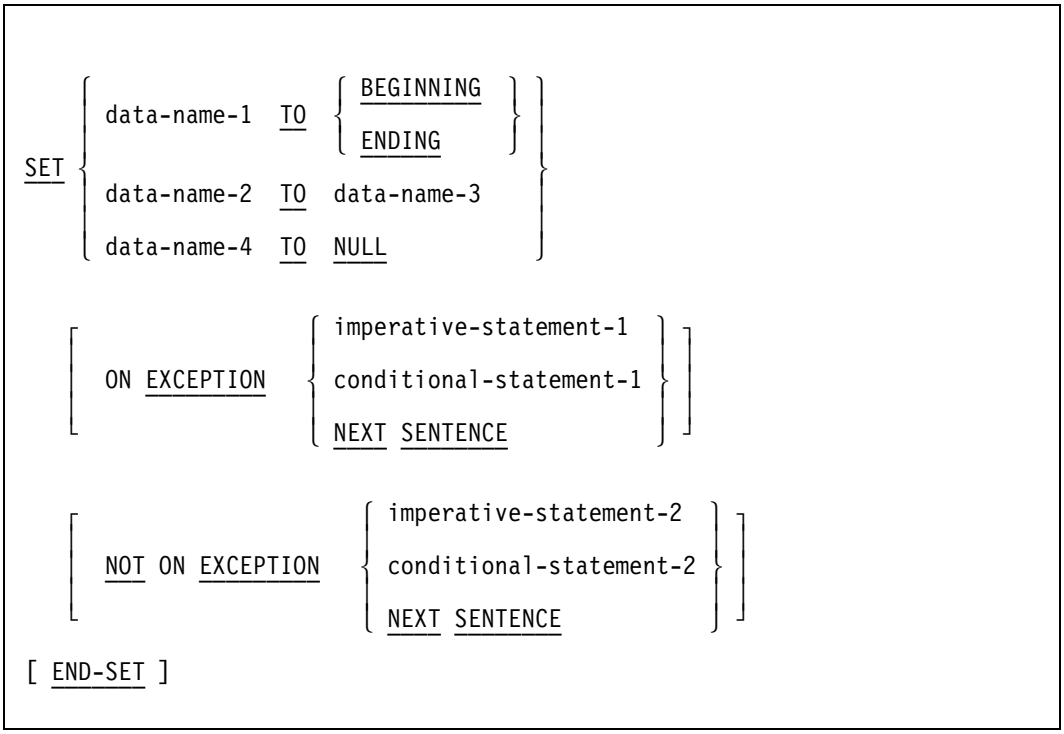
The SET statement alters the current path or changes the value of an item in the current record. The SET statement affects only the record area; it does not affect the data set until you execute a subsequent STORE statement.

The SET statement performs the following steps in order:

1. Frees the current path of the data set, set, or subset
2. Performs one of the following:
   - Alters the current path of a data set, set, or subset to point to the beginning or the ending of the respective structure
   - Alters a set or subset path to point to the current path of a data set
   - Assigns a null value to a particular item

A FIND NEXT statement appearing after a SET TO BEGINNING statement is equivalent to a FIND FIRST statement. A FIND PRIOR statement appearing after a SET TO ENDING statement is equivalent to a FIND LAST statement.

Format



Explanation

Data-name-1 specifies the data set, set, or subset whose current path you want to alter to point to the BEGINNING or ENDING of the data set.

Data-name-2 specifies the set or subset whose current path you want to alter to point to the current record of data-name-3.

Data-name-4 specifies an item of the current record that is assigned a null value. Data-name-4 cannot be a link item, and it cannot be used with the ON EXCEPTION clause or the NOT ON EXCEPTION clause, or with both clauses.

If you declare a a null value in the DASDL, it is used as the null value in this statement. Otherwise, the statement uses the system default null value.

The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur. For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-SET phrase delimits the scope of the SET statement.

Example

The following DASDL description used by the COBOL85 code in Example 3–28 is compiled with the name DBASE:

```
DS DATA SET
     (A ALPHA (20);
      N NUMBER (2)
     );
S SET OF DS
     KEY (A);
SS SUBSET OF DS
    WHERE (N=3);
```

Example 3–28 shows an example of coding for the SET statement.

```
FIND S AT A = "ABC".
SET SS TO DS
ON EXCEPTION
   NEXT SENTENCE.
FIND NEXT SS.
SET S TO BEGINNING
ON EXCEPTION
   DISPLAY "NONE"
END-SET.
SET SS TO ENDING
ON EXCEPTION
   DISPLAY "NONE"
END-SET.
```

Example 3–28. Using the SET Statement

STORE Statement

The STORE statement places a new or modified record into a data set. The statement inserts the data from the user work area for the data set or global record into the data set or global record area.

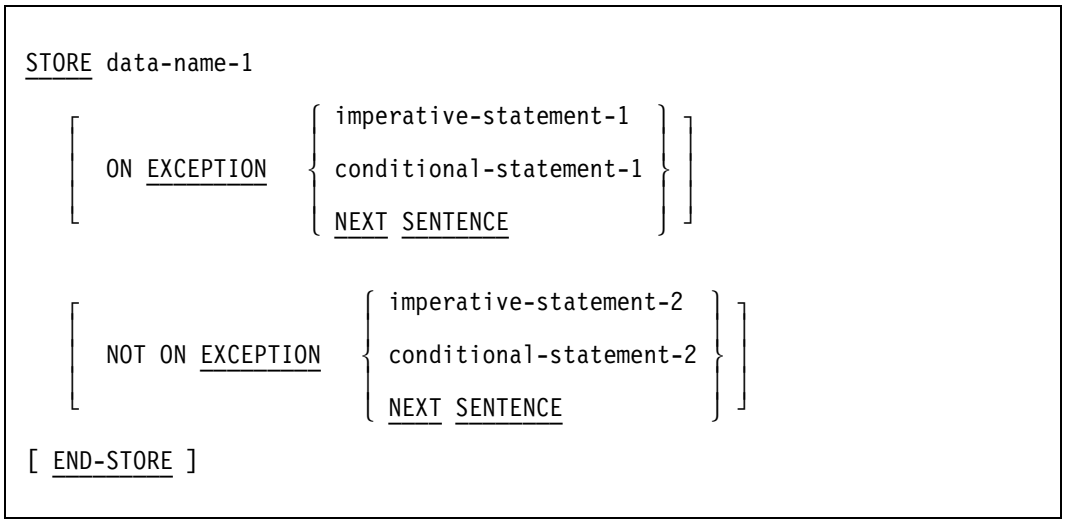
After a CREATE or RECREATE statement, the STORE statement performs the following steps:

- Checks the data in the user work area for validity if you have specified a VERIFY condition in the DASDL.
- Tests the record for validity before it inserts the record into each set in the data set. For example, the STORE statement can test the record to determine whether or not duplicate values for keys are allowed.
- Evaluates the WHERE condition for each automatic subset.
- Inserts the record into all sets and automatic subsets if all conditions are satisfied.
- Locks the new record.
- Alters the data set path to point to the new record.

After a LOCK or MODIFY statement, the STORE statement performs the following steps:

- Checks the data in the user work area for validity if you have specified a VERIFY condition in the DASDL.
- Depending on the VERIFY condition, performs the following steps:
  - If items involved in the insertion conditions have changed, reevaluates the conditions
  - If the condition yields FALSE, removes the record from each automatic subset that contains the record
  - If the condition yields TRUE, inserts the record into each automatic subset that does not contain the record
- Deletes the record and reinserts it into the proper position if you have modified a key used in ordering a set or automatic subset so that the record must be moved within that set or automatic subset.
- Stores the record in a manual subset, but does not reorder that subset. The user is responsible for maintaining manual subsets. A subsequent reference to the record using that subset produces undefined results.

Format



Explanation

Data-name-1 is the name of the data record or data set you want to store. Data-name-1 causes the STORE statement to do one of the following:

- Return the data in the specified data set work area to the data set.
- Return the data in the global data work area to the global data record area.

You must lock the global data record before you execute a STORE statement; otherwise, the program terminates the STORE statement with an exception. The ON EXCEPTION clause specifies an instruction to be performed if an exception condition occurs. The program returns an exception and does not store the record if the record does not meet any of the validation conditions. The program also returns an exception if

- The data set path is valid and the current record is not locked.
- The global data record is not locked.

The NOT ON EXCEPTION clause specifies an instruction to be performed if an exception condition does not occur.

For more information about the EXCEPTION clauses, refer to “DMSII Exceptions” later in this section.

The END-STORE phrase delimits the scope of the STORE statement.

Example

The following DASDL description used by the COBOL85 code in Example 3–29 is compiled with the name DBASE:

```
D DATA SET (
   A ALPHA (3);
   N NUMBER (3);
   );
S SET OF D KEY N;
```

Example 3–29 shows an example of coding for the STORE statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
DATA DIVISION.
DATA-BASE SECTION.
DB  DBASE ALL.
PROCEDURE DIVISION.
START-PRG.
    OPEN UPDATE DBASE.
    CREATE D.
    MOVE "ABC" TO A.
    MOVE 1 TO N.
    STORE D.
RECREATE-D.
    RECREATE D.
    ADD 1 TO N.
    STORE D.
    IF N < 500
        GO TO RECREATE-D
    ELSE
        CLOSE DBASE
        STOP RUN.
```

Example 3–29. Using the STORE Statement

An example of the STORE statement within the context of a complete program is provided at line 024700 in Example 2–25, “COMS Sample Program with a DMSII Database,” in Section 2.

Processing DMSII Exceptions

During the execution of data management statements, the program can encounter any one of several exception conditions. Exception conditions prevent an operation from being performed as specified. The conditions result if the program encounters a fault or does not perform the expected actions. For example, execution of the following statement results in an exception if no entry in S has a value of “JONES” for the key item:

```
FIND S AT NAME = "JONES"
```

If the operation terminates normally, the program returns no exception.

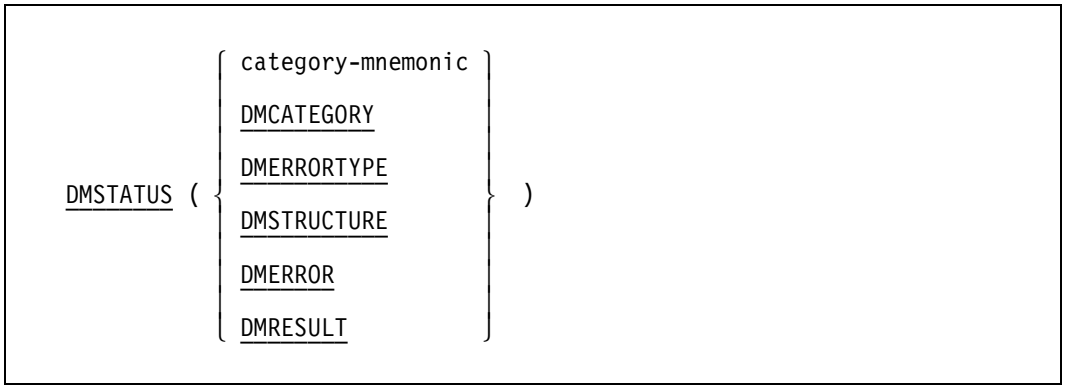
A database status word, DMSTATUS, is associated with each COBOL85 program that accesses a database. The value of DMSTATUS indicates whether an exception has occurred and specifies the nature of the exception. The data management structure number function, DMSTRUCTURE, can also be helpful in analyzing the results of exception conditions.

Information on the exception categories, subcategories, and mnemonics used in exception processing is provided in the *DMSII Application Programming Guide*.

DMSTATUS Database Status Word

The system sets the value of DMSTATUS at the completion of each data management statement. You can use the DMSTATUS entry to discover information about an exception. When interrogating DMSTATUS, you must include an attribute name in parentheses after the DMSTATUS entry.

Format



Explanation

Category-mnemonic yields a TRUE value if the major category specified by category-mnemonic has occurred.

The DMCATEGORY attribute yields a numeric value identifying the major category.

The DMERRORTYPE attribute yields a numeric value identifying the subcategory of the major category.

The DMSTRUCTURE function yields a numeric value identifying the structure number of the structure involved in the exception. See “DMSTRUCTURE Structure Number Function” in this section for more information.

The DMERROR attribute yields a TRUE value if any error has occurred.

The DMRESULT attribute yields the 48-bit contents of DMSTATUS as a PIC X(6) data item. If no exception has occurred, the program returns six EBCDIC nulls (that is, 48“000000000000).

DMSTRUCTURE Structure Number Function

The DMSTRUCTURE function allows a program to determine the structure number of a data set, set, or subset. The structure numbers of all invoked structures are shown in the invocation information in the program listing. Your program can use the structure number to analyze the results of exception conditions.

The DMSTRUCTURE function is most useful when the previous operation on the data set that is spanned by several data sets yields an exception. The program can determine from the structure number which structure caused the exception.

When you declare a partitioned structure in DASDL, it is assigned one or more structure numbers, depending on the following option:

```
OPEN PARTITIONS = integer
```

For example, three structure numbers are assigned to the structure when you specify the following:

```
OPEN PARTITIONS = 3
```

The DMSTRUCTURE function returns the lowest structure number assigned to the structure. However, the value in the result word (DMSTRUCTURE) can be any of the values assigned by DMSII at run time; it is not necessarily the same value every time.

Format

```
data-name-1  ( DMSTRUCTURE )
```

Explanation

Data-name-1 returns the structure number of the data set, set, or subset.

Example

The following provides an example of coding for the DMSTRUCTURE structure number function:

```
IF D(DMSTRUCTURE) = DMSTATUS(DMSTRUCTURE) DISPLAY "D FAULT".
```

DMSII Exceptions

You can use any of the following methods in your program code to handle exceptions:

- Calling the DMERROR Use procedure.
- Specifying the ON EXCEPTION clause with the data management statement.
- If you neither call the DMERROR Use procedure nor specify the ON EXCEPTION clause, the program returns an exception and terminates the program with an error. As a result, the values of the DMSTATUS category, subcategory, and structure number are displayed on the operators console, placed in the system log, and printed with the job summary output.

An explanation of the DMERROR Use procedure and ON EXCEPTION clause are included in this discussion of DMSII exceptions. See Volume 1 for information on the USE statement and Use procedures.

DMERROR Use Procedure

COBOL85 extends the Declaratives Section of the Procedure Division to enable you to specify a DMERROR Use procedure.

Format

```
USE [GLOBAL] ON DMERROR
```

Explanation

The program enters the DMERROR Use procedure each time DMSII returns an exception during the execution of a data management statement, unless the program contains an ON EXCEPTION clause for that statement. Upon exiting the DMERROR Use procedure, control is passed to the statement following the data management statement that encountered the exception.

The DMERROR Use procedure can appear by itself or in any order with other Use procedures in the Declaratives Section. You can declare only one DMERROR Use procedure in a COBOL85 program. The DMERROR Use procedure cannot contain GO TO statements that reference labels outside the procedure. If you use both a DMERROR Use procedure and an ON EXCEPTION clause, the ON EXCEPTION clause takes precedence, and the DMERROR Use procedure is not executed.

For NESTED programs, each program can have its own exception routine. As an alternative, a USE procedure in the main program can be declared as GLOBAL. With the GLOBAL declaration, any nested program which has no USE routine of its own uses the next more globally declared USE routine. The declaration of a USE routine in a nested program overrides the use of any other global USE routine.

Example

Example 3–30 shows the declaration for the DMERROR Use procedure.

```
DECLARATIVES.
DMERR-SECT SECTION.
      USE ON DMERROR.
DMERR-PARA.
      IF DMSTATUS(NOTFOUND)...
END DECLARATIVES.
```

Example 3–30. Declaring the DMERROR Use Procedure

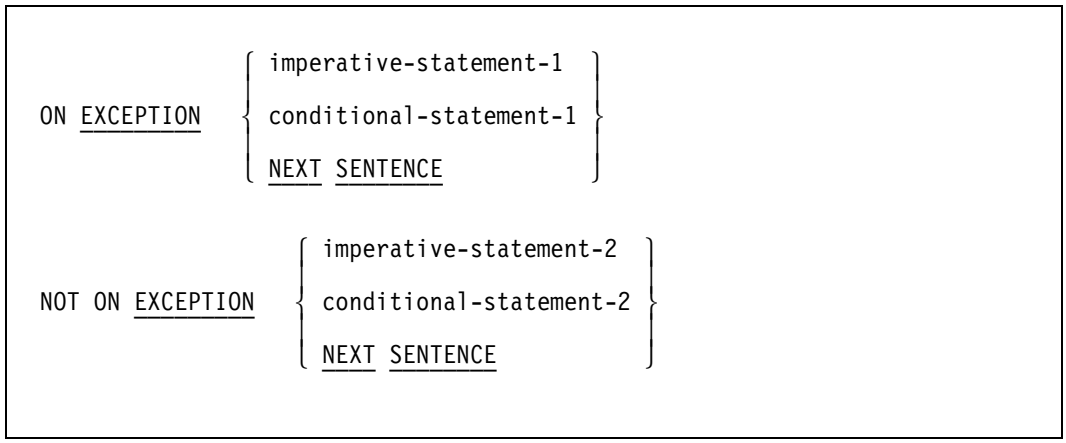
ON EXCEPTION/NOT ON EXCEPTION Clause

An exception condition is an error result that the data management software returns to a program to explain why a requested database operation was not performed. You can include the ON EXCEPTION clause with certain data management statements to specify an alternate statement to be performed when an exception condition occurs. These statements also provide a NOT ON EXCEPTION clause to enable you to specify an additional statement to be performed if an exception condition does not occur. The following data management statements use the ON EXCEPTION and NOT ON EXCEPTION clauses:

For more specific information about exception conditions for each statement, refer to the discussion of the statement earlier in this section.

```
ABORT-TRANSACTION
ASSIGN
BEGIN-TRANSACTION
CANCEL-TRANSACTION-POINT
CLOSE
CREATE
DELETE
END-TRANSACTION
FIND
FREE
GENERATE
INSERT
LOCK
MODIFY
OPEN
RECREATE
REMOVE
SAVE-TRANSACTION-POINT
SECURE
SET
STORE
```

Format



Explanation

For the ON EXCEPTION clause, imperative-statement-1, conditional-statement-1, or NEXT SENTENCE is executed if the program returns an exception.

For the NOT ON EXCEPTION clause, imperative-statement-2, conditional-statement-2, or NEXT SENTENCE is executed if the program does not return an exception.

If you use both a DMERROR Use procedure and an ON EXCEPTION clause, the ON EXCEPTION clause takes precedence, and the DMERROR Use procedure is not executed.

Examples

In the following line of code, a branch to LBL1 is executed if a STORE statement encounters an exception:

```
STORE D ON EXCEPTION GO TO LBL1.
```

Example 3–31 uses the ON EXCEPTION clause and interrogates DMSTATUS.

```
MODIFY S AT X = 3 ON EXCEPTION
      IF DMSTATUS (NOTFOUND) GO NOT-FOUND-L ELSE
      IF DMSTATUS (DEADLOCK) GO DEAD-LOCK-L ELSE
      .
      .
      .
 NOT-FOUND-L.
      IF DMSTATUS (DMERRORTYPE) = 1 statement ELSE
      IF DMSTATUS (DMERRORTYPE) = 2 statement ELSE
      .
      .
      .
 DEAD-LOCK-L.
      IF DMSTATUS (DMERRORTYPE) = 1 statement ELSE
```

Example 3–31. Handling Exceptions with the ON EXCEPTION Clause

Section 4
Using the ADDS Program Interface

The Advanced Data Dictionary System (ADDS) enables you to centrally create and maintain data descriptions. ADDS enables you to do the following:

- Manipulate data
- Define complex data structures
- Update and report on entities or structures in the data dictionary

The program interface for ADDS enables you to invoke entities such as files, records, and record collections. It also provides options for the following:

- Including in your program only entities with a particular status in the dictionary, using the DICTIONARY compiler control option
- Assigning alias identifiers to file and data names to be used in the program, using the INVOKE clause
- Tracking entities, data structures, and databases used by a program, using the PROGRAM clauses of the DICTIONARY statement

You can use ADDS to define Data Management System II (DMSII) databases. For information on DMSII, refer to Section 3, "Using the DMSII Program Interface." For information on using the ADDS product, refer to the *InfoExec Administration Guide*.

If you have created form record libraries using the Screen Design Facility Plus (SDF Plus) and stored them in an ADDS dictionary, you can access these form record libraries just as you would other entities. For more information, refer to Section 5 of this manual, "Using the SDF Plus Program Interface."

The information on the following pages explains how to write a program using the extensions developed for ADDS. Each extension is covered individually, with a description of its purpose or use, the syntax, an explanation, and an example. A sample program appears at the end of this section.

Accessing Entities with a Specific Status

The DICTIONARY compiler control option enables you to set up the status value of entities requested from the data dictionary. The use of this compiler control is optional.

Format

Explanation

PRODUCTION	This value ensures that only PRODUCTION status data dictionary entities are invoked.
TEST	This value ensures that only TEST status data dictionary entities are invoked.

Details

You can define a program entity in ADDS, and use the DICTIONARY compiler control syntax in your program to restrict the invocation of entities to those with a particular status.

Entities with an historical status cannot be invoked by the COBOL85 compiler. Refer to the *InfoExec Administration Guide* for more information on status and for the rules that ADDS uses to search for an entity.

The type is value. The default is none. For more information on compiler control options, refer to the *COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation*.

Example

This example sets the value to PRODUCTION and ensures that only PRODUCTION status data dictionary entities are invoked.

```
$SET DICTIONARY = PRODUCTION
```

Identifying Specific Entities

The version and directory are properties that, with the entity name, are used to uniquely identify an entity in the data dictionary. These attributes are assigned to an entity in an ADDS session. Refer to the *InfoExec Administration Guide* for information about the search rules that ADDS uses to search for entities.

In a program, the VERSION and DIRECTORY clauses are used as follows:

- In the PROGRAM clauses of the DICTIONARY statement to identify a program for tracking
- In the SELECT statement to identify a particular file in the dictionary
- In the data description FROM DICTIONARY clause to identify a particular lower-level entity (such as a data item or a record description) in the dictionary

Although the VERSION and DIRECTORY clauses are optional, it is good practice to provide as much information to identify a particular entity as possible, especially if there are many duplicate items under different directories in the dictionary.

VERSION Clause

When you create an entity in ADDS, the system assigns the entity a version number. The VERSION clause in a program identifies the 6-digit numeric literal version number of the record description.

Format

```
[ VERSION IS literal-1 ]
```

Explanation

literal-1 This must be a 6-digit numeric literal. It must be a valid VERSION number of the entity in the data dictionary.

Example

```
01 SAMPLELIB FROM DICTIONARY
           VERSION IS 1
           DIRECTORY IS  "USER1".
```

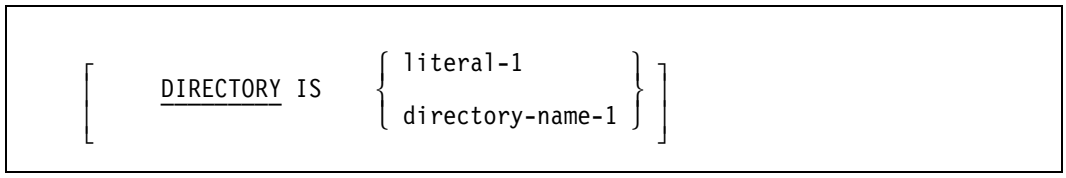
See Also

- "Identifying a Dictionary" in this section.
- "Invoking Data Descriptions in ADDS" in this section.
- "Selecting a File" in this section.

DIRECTORY Clause

The DIRECTORY clause specifies the directory under which the entity is stored in the data dictionary.

Format



Explanation

literal-1	This must be a nonnumeric literal up to 17 alphanumeric characters long.
directory-name-1	This must be a name of up to 17 alphanumeric characters long.

Details

Literal-1 and directory-name-1 must describe the directory under which the data or file description is stored in the dictionary and be specified in the SPECIAL-NAMES paragraph.

Example

```
SELECT ADDS-FILE FROM DICTIONARY
        VERSION IS 1
        DIRECTORY IS SMITH.
```

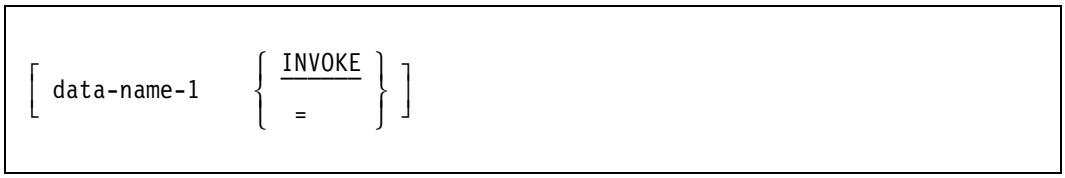
See Also

- "Invoking a Dictionary" in this section.
- "Invoking Data Descriptions in ADDS" in this section.
- "Selecting a File" in this section.

Assigning Alias Identifiers

You can assign an alias identifier to an entity name invoked from the dictionary by using the INVOKE clause. You can then refer to the entity by its alias identifier throughout the rest of your program. The use of the INVOKE clause is optional.

Format



Explanation

data-name-1	This user-defined identifier names a data item described in a data description entry or file select entry. Only 01-level data names or file names are allowed. Once you assign an alias, any reference to the entity in the program must specify data-name-1. Also, all Procedure Division statements must use this alias.
INVOKE clause	The INVOKE clause or the equal sign (=) is used before the FROM DICTIONARY clause to assign an alias in the SELECT statement to a file. Also, this clause is used in the 01-level data description entry to assign an alias to an entity such as a record or a data item.

Details

Assigning an alias is useful when, for example, you want to invoke the same record twice in your program. Assigning an alias enables you to use a unique qualifier.

In the program, the INVOKE clause is used in the Environment Division and the Data Division to assign an alias as follows:

- In the SELECT statement, to assign an alias to a file
- In the data description entry FROM DICTIONARY clause, to assign an alias to a lower-level entity such as a record or a data item

Example

```
01  MY-INTERNAL-NAME INVOKE ADDS-ENTITY-NAME
              FROM DICTIONARY.
```

See Also

- “Invoking File Descriptions” in this section.
- “Selecting a File” in this section.
- “Invoking Data Descriptions in ADDS” in this section.

Identifying a Dictionary

The dictionary that you use during compilation is identified in the DICTIONARY statement in the SPECIAL-NAMES paragraph of the program. Optional program clauses also enable program tracking.

Program tracking is a useful feature of ADDS. When defining a program entity, you can direct the dictionary to keep track of the data structures and entities that you invoke in your program. To do this, you identify the program by using the PROGRAM-NAME, PROGRAM-VERSION, and PROGRAM-DIRECTORY clauses. For more information on program tracking, refer to the *InfoExec Administration Guide*.

Format

```
┌                                    ┐
  DICTIONARY IS literal-1
[ PROGRAM-NAME  IS literal-2 ]
[ PROGRAM-VERSION  IS literal-3 ]
[ PROGRAM-DIRECTORY IS literal-4 ]
└                                    ┘
```

Explanation

literal-1	Literal-1 must be the function name of the dictionary library.
literal-2	Literal-2 must be a valid program name in the data dictionary.
literal-3	Literal-3 must be a 6-digit numeric literal.
literal-4	Literal-4 must be a valid data dictionary directory.
DICTIONARY clause	The DICTIONARY clause specifies the function name of the dictionary library. The function name is the name equated to a library code file when using the operator display terminal (ODT) SL command.
PROGRAM-NAME clause	The PROGRAM-NAME clause specifies the name of the entity of type program that is to be tracked. This clause is needed if the entity tracking is defined in ADDS. If the tracking is not defined, then this clause does not enforce entity tracking and only program information is sent.

PROGRAM-VERSION clause	The PROGRAM-VERSION clause specifies the version of the program to be tracked.
PROGRAM-DIRECTORY clause	The PROGRAM-DIRECTORY clause specifies the directory of the program to be tracked.

Details

You can identify the dictionary in the SPECIAL-NAMES paragraph. Optional program clauses enable entity tracking in ADDS. You can invoke only one data dictionary for a main program and all nested programs contained within it.

If multiple sequential programs exist in one source file, then you must specify the data dictionary for each sequential, separately compilable program. The dictionary identification clause can appear only in the main program of a separately compilable program and cannot appear in a nested program.

The DICTIONARY, PROGRAM-NAME, and PROGRAM-DIRECTORY literals can have an extra period at the end. The period on DICTIONARY is used for the FUNCTIONNAME library attribute and is appended if not already specified in the literal.

If you do not specify a dictionary by using the DICTIONARY statement, the compiler uses the dictionary named DATADICTIONARY by default.

Example

```
001500  SPECIAL-NAMES.
001600    DICTIONARY IS "DATADICTIONARY"
001700        PROGRAM-NAME IS "EXAMPLE-PROGRAM"
001800        PROGRAM-VERSION IS 1
001900        PROGRAM-DIRECTORY IS "JOHNDOE".
```

See Also

“Identifying Specific Entities” in this section.

Selecting a File

The following format for the SELECT statement is used to include files from the dictionary in your program.

Format

```
SELECT [ file-name-1 { INVOKE } ] file-name-2
                     {   =    }
       FROM DICTIONARY
       [ VERSION  IS literal-1 ]
       [  DIRECTORY  IS { literal-2        } ]
       [                { directory-name-1 } ] . [;]
```

Explanation

file-name-1 file-name-2 FROM DICTIONARY	The optional INVOKE clause specifies an alias for file-name-2. All subsequent references to this file must use the file-name-1 alias. File-name-2 is the entity invoked from the dictionary.
literal-1	Literal-1 must be a numeric literal up to 6 digits long specifying the version under which the file is stored in the data dictionary.
literal-2	Literal-2 must be a nonnumeric literal up to 17 alphanumeric characters long specifying the version under which the file is stored in the data dictionary.
directory-name-1	Directory-name-1 must be a name of up to 17 alphanumeric characters long specifying the directory under which the file is stored in the data dictionary.
VERSION clause DIRECTORY clause	Refer to “Identifying Specific Entities” in this section for information on the VERSION and DIRECTORY clauses.
INVOKE =	Refer to “Assigning Alias Identifiers” in this section for information on the INVOKE clause.

; (Semicolon)	The semicolon that follows the ending period can be used to control the behavior of compiler control records (CCRs) and the format of listings. This semicolon should always be separated from the ending period of the SELECT statement by at least one space. IF a CCR immediately follows a SELECT ... FROM DICTIONARY statement, the compiler option changes might occur before the compiler processes the information invoked from the dictionary. This situation can be avoided by using the semicolon after the ending period. The semicolon ensures that the compiler processes the invoked information before the option actually changes.

Details

You can assign an alias identifier to a selected file. You can also use the VERSION and DIRECTORY clauses to identify the particular file. The SELECT statement is placed in the File-Control paragraph of the Input-Output section of the program.

Example

```
002000 INPUT-OUTPUT SECTION.
002100 FILE-CONTROL.
002200    SELECT SORT-FILE INVOKE ADDS-FILE FROM DICTIONARY
002300        VERSION IS 1
002400        DIRECTORY  IS  "*".
002500   SELECT REMOTE-FILE FROM DICTIONARY.
002600 DATA DIVISION.
002700 FILE SECTION.
```

See Also

- “Assigning Alias Identifiers” in this section.
- “Identifying Specific Entities” in this section.

Invoking File Descriptions

The file description (FD) or sort-merge file description (SD) entry provides information about the following:

- The physical structure of a file
- The identification of a file
- The record names pertaining to a file

The FD and SD statements invoke all file attributes of the file named in the SELECT statement. By using the optional INVOKE ALL clause, you can invoke the record descriptions as well as the file attributes. For information on the File Section and the file description entry in a program, refer to the *COBOL ANSI-85 Programming Reference Manual*, *Volume 1: Basic Implementation*.

Format 1

```
FD file-name INVOKE [ ALL ] [ GLOBAL ] [ EXTERNAL ]. [;]
```

Explanation

FD	The level indicator FD identifies the beginning of a file description and must precede the file name. FD refers to file description.
INVOKE ALL clause	If you used the SELECT clause to select a file, the INVOKE ALL clause invokes all record descriptions as well as file attributes defined in the data dictionary for this file. If you do not specify the INVOKE ALL clause, then the FD statement must be followed by one or more record descriptions, which can be invoked from the dictionary only when the record is related to the file in the dictionary.
GLOBAL	You must specify the GLOBAL option in the main program if all nested subprograms need access to the file description.
EXTERNAL	You must specify the EXTERNAL option if the file structure is shared through interprogram communication (IPC) at run time.

Format 2

```
SD file-name INVOKE [ ALL ]. [;]
```

Explanation

SD	The level indicator SD identifies the beginning of a file description and must precede the file name. SD refers to a sort-merge file description.
INVOKE ALL clause	The INVOKE ALL clause specifies that all record descriptions defined in the data dictionary for this file are invoked. If you do not specify the INVOKE ALL clause, then the SD statement must be followed by one or more record descriptions, which can be invoked from the dictionary only when the record is related to the file in the dictionary.
; (Semicolon)	The semicolon that follows the ending period can be used to control the behavior of compiler control records (CCRs) and the format of listings. This semicolon should always be separated from the ending period of the FD or SD statement referencing a file from the dictionary by at least one space. IF a CCR immediately follows an FD or SD invoked from the dictionary, the compiler option changes might occur before the compiler processes the information invoked from the dictionary. This situation can be avoided by using the semicolon after the ending period. The semicolon ensures that the compiler processes the invoked information before the option actually changes.

Details

These file description statements are valid only for files previously invoked using the SELECT FROM DICTIONARY statement. If you assigned an alias by using the INVOKE option in the SELECT statement, then you must use that alias identifier for the file name.

If you define the file in the SELECT FROM DICTIONARY statement, then you obtain the file description from the data dictionary, and all attribute specifications in the source file (up to the period) are illegal.

Examples

The following are examples of the FD and SD statements:

```
002900 FD REMOTE-FILE.

003000 SD SORT-FILE  INVOKE ALL.
```

See Also

- "Assigning Alias Identifiers" in this section.
- "Selecting a File" in this section.

Invoking Data Descriptions in ADDS

A data description entry specifies the characteristics of a particular data item. You use the FROM DICTIONARY clause to obtain an entity from the dictionary.

Format

```
level-number

   {                                                  }
   {[data-name-1 { INVOKE } ] data-name-2             }
   {             {   =    }                           }
   {                                                  }
   { group-list-name-1                                }
   {                                                  }

   FROM DICTIONARY

   [ VERSION  IS literal-1 ]

   [DIRECTORY IS { literal-2        } ]
   [             { directory-name-1 } ]

   [ COMMON ]

   [ GLOBAL ]

   [ EXTERNAL ]. [;]
```

Explanation

level-number	You can invoke the 01-level data description entry within the File Section, the Working-Storage Section, the Linkage Section, or the Local-Storage Section. The data description entry is used within the File Section to invoke record descriptions for a file that has not been declared with the INVOKE ALL option.
data-name-1 data-name-2	The INVOKE data-name-2 clause can be used to invoke 01-record data descriptions so that data-name-1 is an alias referenced in the program.
group-list-name-1	The group-list-name-1 identifies a record collection of unrelated descriptions of 77-level or 01-level items and records to be included in the Working-Storage Section, the Linkage Section, or the Local-Storage Section.
VERSION clause	The VERSION clause imports the exact version of the data description being requested. See “VERSION Clause” under “Identifying Specific Entities” in this section for more information on the Version clause.
literal-1	Literal-1 must be a numeric literal up to 6 digits long.
DIRECTORY clause	The DIRECTORY clause specifies the directory under which the entity is stored in the data dictionary. See “DIRECTORY Clause” under “Identifying Specific Entities” in this section for more information.
literal-2	Literal-2 must be a nonnumeric literal up to 17 alphanumeric characters long.
directory-name-1	Directory-name-1 must be a name of up to 17 alphanumeric characters long.
COMMON	You must specify this attribute in a subprogram for referencing a data description declared in the host program when binding.
GLOBAL	You must specify this attribute in the main program if all nested subprograms need access to the data description.
EXTERNAL	You must specify this attribute if the data structure is shared through interprogram communication (IPC) at run time.

; (Semicolon)	The semicolon that follows the ending period can be used to control the behavior of compiler control records (CCRs) and the format of listings. This semicolon should always be separated from the ending period of the data invocation by at least one space. IF a CCR immediately follows a data item invoked from the dictionary, the compiler option changes might occur before the compiler processes the information invoked from the dictionary. This situation can be avoided by using the semicolon after the ending period. The semicolon ensures that the compiler processes the invoked information before the option actually changes.

Details

When a program or library invokes data from a data dictionary, the COBOL85 compiler includes a list of the invoked data in the listing file. The data appears immediately after the invocation. To prevent the invoked data from appearing in the listing, you can reset the LIST compiler option at the line that invokes the dictionary. For more information, refer to "How ADDS Data Appears in a COBOL85 Listing" later in this section.

The following rules apply for invoking data descriptions in ADDS:

- You can invoke only a 01-level record or record collection directly from the data dictionary. You cannot directly invoke an elementary item.
- You can invoke 01-level records within the File Section, the Working-Storage Section, the Linkage Section, or the Local-Storage Section. However, you cannot invoke record collections within the File Section. You can invoke record collections only within the Working-Storage Section, the Linkage Section, or the Local-Storage Section.
- Within the File Section, if you select a file by using the SELECT statement shown for ADDS, the record must be associated with that file in the data dictionary.
- A record collection cannot be given an alias identifier by means of the INVOKE clause.
- If a file is selected using the SELECT statement for ADDS, then record descriptions invoked from the dictionary are allowed and record descriptions coded as usual are allowed.

Example

```
003300  WORKING-STORAGE SECTION.
003400  01 MY-REC-LIST INVOKE
003500     ADDS-REC-LIST FROM DICTIONARY.
003600         VERSION IS 2
003700         DIRECTORY IS  "*".
```

See Also

- "Assigning Alias Identifiers" in this section.
- "Invoking File Descriptions" in this section.
- "Identifying Specific Entities" in this section.

Sample ADDS Program

The following sample program uses the ADDS interface syntax. First, a list presents the data definitions defined in the ADDS dictionary that are used in the program. Then the COBOL85 program is presented.

ADDS Descriptions

```
*****************************************************************
*                                                               *
*               DATA DEFINITIONS IN ADDS                        *
*                                                               *
*****************************************************************
 FILE-CONTROL.
   SELECT UNSORTED-SALES
       ASSIGN TO DISK.
   SELECT SORT-FILE
       ASSIGN TO SORT WITH DISK.
   SELECT SORTED-SALES
       ASSIGN TO DISK.
   SELECT PRINT-OUT
       ASSIGN TO PRINTER.
   SELECT EMPLOYEE-INFO
       ASSIGN TO DISK;
       ORGANIZATION IS INDEXED;
       ACCESS MODE IS RANDOM;
       RECORD KEY IS KEY-EMPNO.
 FILE SECTION.
 FD  SORTED-SALES
         LABEL RECORDS ARE STANDARD.
 01  SORTED-REC.
     05  DEPT-IN                 PIC 99.
     05  SLSNO-IN                PIC 9(5).
     05  AMT-OF-SALES-IN         PIC 9(4)V99.
     05  FILLER                  PIC X(67).
 FD  EMPLOYEE-INFO
         LABEL RECORDS ARE STANDARD.
 01  EMPLOYEE-REC.
     05  KEY-EMPNO               PIC 9(5).
     05  NAME.
         06  FIRSTNAME           PIC X(10).
         06  LASTNAME            PIC X(10).
     05  PHONE.
         06  AREACODE            PIC 999.
         06  PHONENUMBER         PIC 9(7).
     05  FILLER                  PIC X(45).
 SD  SORT-FILE.
 01  WORK-REC.
     05  W-DEPT-NO                PIC 99.
     05  FILLER                   PIC X(98).
 FD  UNSORTED-SALES.
```

```
 FD  PRINT-OUT
         LABEL RECORDS ARE OMITTED.
 01  PRINT-REC                    PIC X(133).
 WORKING-STORAGE SECTION.
*****************************************************************
*  HEADING-1 AND HEADING-2 WILL BE GROUPED UNDER HEADER-LIST     *
*                   IN THE ADDS DICTIONARY                       *
*****************************************************************
 01  HEADING-1.
     05  FILLER                  PIC X(50)      VALUE SPACES.
     05  FILLER                  PIC X(21)
           VALUE  "MONTHLY STATUS REPORT".
     05  FILLER                  PIC X(9)       VALUE SPACES.
     05  FILLER                PIC X(5)
           VALUE  "PAGE".
     05  HL-PAGE-NO-OUT          PIC 99.
     05  FILLER                  PIC X(46)      VALUE SPACES.
 01  HEADING-2.
     05  FILLER                  PIC X(11)      VALUE SPACES.
     05  FILLER                  PIC X(10)
           VALUE  "DEPT".
     05  FILLER                  PIC X(20)
           VALUE  "SALESPERSON NO".
     05  FILLER                  PIC X(20)
           VALUE  "NAME".
     05  FILLER                  PIC X(15)
           VALUE  "PHONE NUMBER".
     05  FILLER                  PIC X(12)
           VALUE  "AMT OF SALES".
     05  FILLER                  PIC X(80)      VALUE SPACES.
```

COBOL85 Program Using ADDS Interface Syntax

```
*****************************************************************
*                                                               *
*                     ADDS INTERFACE EXAMPLE                    *
*                                                               *
*    This is an example showing the proposed ADDS Interface     *
*    syntax in COBOL85.                                         *
*         This program creates a departmental sales report. It  *
*    first sorts a file containing sales information by         *
*    department number and then fetches the corresponding       *
*    salesman information from the salesman-info file.          *
*    This information is displayed and the total sales for the  *
*    department are displayed at the end of each department     *
*    section.                                                   *
*                                                               *
*****************************************************************
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
     DICTIONARY IS  "DATADICTIONARY"
          PROGRAM-NAME IS  "SAMPLE-PROGRAM"
          PROGRAM-VERSION IS 1
          PROGRAM-DIRECTORY IS  "SAMPLE".
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
        SELECT UNSORTED-SALES
        FROM DICTIONARY
            DIRECTORY IS  "SAMPLE".
        SELECT WORK-FILE INVOKE SORT-FILE
        FROM DICTIONARY
            VERSION IS 2
            DIRECTORY IS  "SAMPLE".
        SELECT SORTED-SALES
        FROM DICTIONARY
            VERSION IS 1
            DIRECTORY IS  "SAMPLE".
        SELECT SALESMAN-INFO INVOKE EMPLOYEE-INFO
        FROM DICTIONARY
            VERSION IS 1
            DIRECTORY IS  "SAMPLE".
         SELECT PRINT-OUT
         FROM DICTIONARY
            VERSION IS 1
            DIRECTORY IS  "SAMPLE".
 DATA DIVISION.
 FILE SECTION.
 FD     SORTED-SALES INVOKE ALL.
 FD     SALESMAN-INFO INVOKE ALL.
 SD     WORK-FILE INVOKE ALL.
```

```
 FD     UNSORTED-SALES.
 01     UNSORTED-REC            PIC X(100).
 FD     PRINT-OUT.
 01     PRINT-REC
          FROM DICTIONARY
              VERSION IS 1
              DIRECTORY IS  "SAMPLE".
 01  DETAIL-LINE.
     05  FILLER                  PIC X(12).
     05  DL-DEPT-OUT             PIC 99.
     05  FILLER                  PIC X(9).
     05  DL-SLSNO-OUT            PIC 9(5).
     05  FILLER                  PIC X(14).
     05  DL-NAME-OUT             PIC X(20).
     05  L-PAREN                 PIC X.
     05  DL-AREACODE-OUT         PIC 999.
     05  R-PAREN                 PIC X.
     05  DL-PHONENUMBER-OUT      PIC 9(7).
     05  FILLER                  PIC XX.
     05  DL-AMT-OF-SALES-OUT     PIC $$,$$$.99.
     05  FILLER                  PIC X(48).
 01  GROUP-REC.
     05  FILLER                  PIC X(61).
     05  TOTAL                   PIC X(18).
     05  DEPT-TOTAL-OUT          PIC $$$,$$$.99.
     05  FILLER                  PIC X(44).
/
*****************************************************************
*                                                               *
*                       WORKING STORAGE                         *
*                                                               *
*****************************************************************
 WORKING-STORAGE SECTION.
 01  WORK-AREAS.
     05  ARE-THERE-MORE-RECORDS  PIC X(3)       VALUE  YES .
         88  MORE-RECORDS                       VALUE  YES .
         88  NO-MORE-RECORDS                    VALUE  NO .
     05  WS-HOLD-DEPT            PIC 99         VALUE ZEROS.
     05  WS-DEPT-TOTAL           PIC 9(5)V99    VALUE ZEROS.
     05  WS-LINE-CT              PIC 99         VALUE ZEROS.
     05  WS-PAGE-CT              PIC 99         VALUE ZEROS.
 01  HEADER-LIST
         FROM DICTIONARY
         VERSION IS 1
         DIRECTORY IS  "SAMPLE".
/
```

```
*****************************************************************
*                                                               *
*                      MAIN BODY OF PROGRAM                     *
*                                                               *
*           Controls the direction of program logic.            *
*                                                               *
*****************************************************************
 PROCEDURE DIVISION.
 MAIN-MODULE.
     PERFORM INITIALIZATION-RTN.
     PERFORM HEADING-RTN.
     PERFORM DETAIL-RTN
         UNTIL NO-MORE-RECORDS.
     PERFORM END-OF-JOB-RTN.
     STOP RUN.
/
*****************************************************************
*                                                               *
*                           DETAIL-RTN                          *
*                                                               *
*    Is performed from the main-module-rtn. It controls depart- *
*    ment break, pagination, and reads the next record. It also *
*    gets the corresponding salesman information record for each*
*    report.                                                    *
*                                                               *
*****************************************************************
 DETAIL-RTN.
     IF DEPT-IN NOT = WS-HOLD-DEPT
         PERFORM CONTROL-BREAK.
     PERFORM INIT-DETAIL-LINE.
     MOVE DEPT-IN TO DL-DEPT-OUT.
     MOVE SLSNO-IN TO DL-SLSNO-OUT.
     MOVE SLSNO-IN TO KEY-EMPNO.
     READ SALESMAN-INFO
         KEY IS KEY-EMPNO.
     MOVE NAME TO DL-NAME-OUT.
     MOVE AREACODE TO DL-AREACODE-OUT.
     MOVE PHONENUMBER TO DL-PHONENUMBER-OUT.
     MOVE AMT-OF-SALES-IN TO DL-AMT-OF-SALES-OUT.
     IF WS-LINE-CT > 25
         PERFORM HEADING-RTN.
     WRITE DETAIL-LINE
         AFTER ADVANCING 2 LINES.
     ADD AMT-OF-SALES-IN TO WS-DEPT-TOTAL.
     ADD 1 TO WS-LINE-CT.
     READ SORTED-SALES
         AT END MOVE "NO " TO ARE-THERE-MORE-RECORDS.
/
```

```
*****************************************************************
*                                                               *
*                        CONTROL-BREAK                          *
*                                                               *
*    Is performed from detail-rtn and prints department         *
*    totals, resets control fields and totals.                  *
*                                                               *
*****************************************************************
 CONTROL-BREAK.
     PERFORM INIT-GROUP-REC.
     MOVE WS-DEPT-TOTAL TO DEPT-TOTAL-OUT.
     WRITE GROUP-REC
         AFTER ADVANCING 2 LINES.
     MOVE ZEROS TO WS-DEPT-TOTAL.
     MOVE DEPT-IN TO WS-HOLD-DEPT.
     ADD 1 TO WS-LINE-CT.
/
*****************************************************************
*                                                               *
*                         HEADING-RTN                           *
*                                                               *
*    Is performed from main-module, detail-rtn, and control-    *
*    break. It prints out headings and resets line counter.     *
*                                                               *
*****************************************************************
 HEADING-RTN.
     MOVE SPACES TO PRINT-REC.
     MOVE   " THIS IS THE ADDS INTERFACE EXAMPLE AS SEEN IN THE
           " ADDS/SDF PLUS INTERFACE IN COBOL85 DESIGN DOCUMENT"
         TO PRINT-REC.
     WRITE PRINT-REC AFTER ADVANCING PAGE.
     MOVE ALL "*" TO PRINT-REC.
     WRITE PRINT-REC AFTER ADVANCING 2 LINES.
     ADD 1 TO WS-PAGE-CT.
     MOVE WS-PAGE-CT TO HL-PAGE-NO-OUT.
     WRITE PRINT-REC FROM HEADING-1
         AFTER ADVANCING 3 LINES.
     WRITE PRINT-REC FROM HEADING-2
         AFTER ADVANCING 3 LINES.
     MOVE ALL "*" TO PRINT-REC.
     WRITE PRINT-REC AFTER ADVANCING 1 LINE.
     MOVE SPACES TO PRINT-REC.
     MOVE ZEROS TO WS-LINE-CT.
/
```

```
*****************************************************************
*                                                               *
*                      INIT-DETAIL-LINE                         *
*                                                               *
*    Is performed from detail-rtn. It initializes the detail    *
*    line before data is moved in to be printed.                *
*                                                               *
*****************************************************************
 INIT-DETAIL-LINE.
     MOVE SPACES TO DETAIL-LINE.
     MOVE "(" TO L-PAREN.
     MOVE ")" TO R-PAREN.
/
*****************************************************************
*                                                               *
*                      INIT-GROUP-REC                           *
*                                                               *
*    Is performed from control-break. It initializes the        *
*    total line before data is moved in to be printed.          *
*                                                               *
*****************************************************************
 INIT-GROUP-REC.
     MOVE SPACES TO GROUP-REC.
     MOVE "TOTAL FOR DEPT IS  " TO TOTAL.
/
```

```
*****************************************************************
*                                                               *
*                     INITIALIZATION-RTN                        *
*                                                               *
*    Is performed from main-module. It opens files, sorts the   *
*    sales information by department number, performs the       *
*    initial read and initializes dept-hold.                    *
*                                                               *
*****************************************************************
 INITIALIZATION-RNT.
     SORT WORK-FILE
         ON ASCENDING KEY W-DEPT-NO
         USING UNSORTED-SALES
         GIVING SORTED-SALES.
     OPEN INPUT  SORTED-SALES
          INPUT  SALESMAN-INFO
          OUTPUT PRINT-OUT.
     READ SORTED-SALES
         AT END MOVE "NO" TO ARE-THERE-MORE-RECORDS.
     MOVE DEPT-IN TO WS-HOLD-DEPT.
/
*****************************************************************
*                                                               *
*                       END-OF-JOB-RTN                          *
*                                                               *
*    Is performed from main-module. It performs end-of-job      *
*    functions, closes files and returns control to operation   *
*    system.                                                    *
*                                                               *
*****************************************************************
 END-OF-JOB-RTN.
     PERFORM INIT-GROUP-REC.
     MOVE WS-DEPT-TOTAL TO DEPT-TOTAL-OUT.
     WRITE GROUP-REC
         AFTER ADVANCING 2 LINES.
     CLOSE SORTED-SALES
           SALESMAN-INFO
           PRINT-OUT.
```

How ADDS Data Appears in a COBOL85 Listing

The preceding COBOL85 program invokes several data descriptions from an ADDS program. Each data invocation is documented in the COBOL listing.
The first line of the COBOL listing names the dictionary directory from which the data was invoked. The right-hand margin of each line contains the status and version for each entity invoked. The status is indicated by the first character of the word DEFINEIT, TEST, PRODUCTION, or HISTORY, or is blank if the status is unspecified. The version number is printed immediately after the status. Each line ends with the letter D to indicate that the line contains information invoked from the dictionary.

Example

The following SELECT statement appears in the File Control paragraph of the Input-Output section of the COBOL program presented earlier in this section. This statement invokes data from the ADDS data dictionary:

```
SELECT SALESMAN-INFO INVOKE EMPLOYEE-INFO
FROM DICTIONARY
    VERSION IS 1
    DIRECTORY IS "SAMPLE".
```

The resultant entry in the COBOL listing for this data invocation looks like the following:

```
000100*--DICTIONARY DIRECTORY: SAMPLE.
000110*   ASSIGN TO DISK;
000120*   ORGANIZATION IS INDEXED;
000130*   ACCESS MODE IS RANDOM;
000140*   RECORD KEY IS KEY-EMPNO.
      *
  000160    FD  EMPLOYEE-INFO
      *
000100*--DICTIONARY DIRECTORY: SAMPLE.
000110*   LABEL RECORDS ARE STANDARD.
000120 01 EMPLOYEE-REC.
000130    05  KEY-EMPNO              PIC 9(5).
000140    05  NAME.
000150       06  FIRSTNAME           PIC X(10).
000160       06  LASTNAME            PIC X(10).
000170    05  PHONE.
000180       06  AREACODE            PIC 999.
000190       06  PHONENUMBER         PIC 9(7).
000200    05  FILLER                 PIC X(45).
```

Note: *To prevent invoked data from appearing in the listing, you can reset the LIST compiler control option at the line that invokes the dictionary data.*

Section 5
Using the SDF Plus Program Interface

Screen Design Facility Plus (SDF Plus) is a user interface management system that gives you 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 that validates data entered on forms by application users.

The program interface for SDF Plus includes the following:

- Extensions that enable you to invoke form record library descriptions of SDF Plus forms into your program
- Extensions that enable you to send and receive form data
- Extensions that enable you to send transaction error messages and text messages

This section provides information about the extensions developed for SDF Plus. Each extension is presented with its syntax and examples. A sample program appears at the end of this section.

For information on defining the concepts and principles of SDF Plus, refer to the *Screen Design Facility Plus (SDF Plus) Capabilities Manual*.

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*.

You can use SDF Plus with the following:

- The Advanced Data Dictionary System (ADDS)
- The Communications Management System (COMS)

For information on the extensions used with ADDS and COMS, refer to Section 4, “Using the ADDS Program Interface,” and Section 2, “Using the COMS Program Interface.”

Understanding the SDF Plus Interface

COBOL85 application programs can interact with SDF Plus using either a CALL interface or a COMS interface. Programs using the CALL interface interact through a set of entry procedures, and programs using the COMS interface interact through the standard COMS SEND and RECEIVE verbs. Either interface can be used to interface with SDF Plus except that certain capabilities, such as COMS windows, require the COMS interface. Both the CALL interface and the COMS interface are described in this section.

The interface to SDF Plus is based on the concept of form record libraries.

Form Record Libraries

A form record library is a collection of descriptions of each of the message types and transaction types associated with an SDF Plus form library. The form record library resides in a data dictionary and is maintained by SDF Plus. It has the same name as the form library it describes.

A COBOL85 program must have access to the description within the form record library so that it can properly format data for transfer between the program and SDF Plus. Declaring the name of a form record library within a program enables the COBOL85 compiler to obtain a copy of the form record library from the data dictionary at compilation time. You can invoke multiple form record libraries within the same program.

The various elements of the form record library are described in the following pages.

Message Types

Message types represent records of data, either data received from a form or data sent to a form. Each message type consists of fields of defined length and data type. A form can have several message types associated with it; therefore, a one-to-one relationship between forms and messages does not exist. For example, a form might have a prefill request message type (which is also the prefill response message type), an update request message type, and an update response message type (also called a standard response), each having a different data format. In some manuals, the term *form record* is a synonym for *message type*.

With SDF Plus there is a complete separation between the form processing logic and the application logic. For example, modifying the layout or data validation logic of a form does not affect the amount or type of data transferred between the form and the application. Therefore, the message type definitions do not change. You need not recompile a program unless the message type definitions change.

Transaction Types

For each form, SDF Plus allows two types of transaction, prefill and update. For each transaction type, there is an associated request message type, response message type, and a list of transaction errors.

When an application receives data from the user interface, the application uses the transaction type to determine what data it has received (request message type) and what data to send in response (response message type), or what errors can be returned (list of transaction errors).

Example

A form library, MSGKEYS, contains two update-only forms, FORM1 and FORM2. Therefore, MSGKEYS contains two transaction types: FORM1 update transaction type, FORM1TT, and FORM2 update transaction type, FORM2TT. FORM1TT contains two associated message types, a request message type, FORM1, and a response message type, MSGKEYSSR. FORM2TT also contains two message types, FORM2 and MSGKEYSSR. The response message type is the same for all update transaction types in the form record library; it is the standard response (SR) for the form record library, MSGKEYSSR.

The following example shows the general structure, but not the actual record description, of the form record library obtained from SDF Plus:

```
FORM RECORD LIBRARY   MSGKEYS
TRANSACTION TYPE           FORM1TT
MESSAGE TYPE                    FORM1
MESSAGE TYPE                    MSGKEYSSR
TRANSACTION TYPE           FORM2TT
MESSAGE TYPE                    FORM2
MESSAGE TYPE                    MSGKEYSSR
```

The following example shows what you should do to invoke the form record library from the dictionary:

```
01 MSGKEYS FROM DICTIONARY
         DIRECTORY IS "SMITH".
```

The following example shows how COBOL85 interprets the form record library and constructs syntax for accessing the information described by the form record library:

```
01 FORM1.
      04 KEYFIELD   PIC X(5).
      04 DATAFIELD  PIC X(4).
      04 QUITFIELD  PIC X(1).
01 MSGKEYSSR.
      04 MSGKEYSSRF PIC X(1).
01 FORM2.
      04 KEY2FLD    PIC X(5).
      04 DATA2FLD   PIC 9(4).
      04 QUIT2FLD   PIC X(1).
```

Note that in the first example showing the general structure of the form record library, MSGKEYSSR appears twice. However, in the last example showing how COBOL85 interprets the form record library, MSGKEYSSR appears only once. SDF Plus maintains the logical structure, as shown in the first example, when processing the user interface. When coding and designing the program, you also must keep the logical structure of the form record library in mind.

Identifying the Dictionary

You identify the dictionary that contains the form library you want to use by including a DICTIONARY statement in the SPECIAL-NAMES paragraph of the Environment Division. You can use optional program clauses to enable entity tracking in ADDS.

Format

```
[ DICTIONARY IS literal-1. ]
```

Explanation

DICTIONARY	The DICTIONARY clause specifies the function name of the dictionary library.
literal-1	Literal-1 is the function name that you can equate to a library code file by using the SL (Support Library) system command.

Refer to the *Screen Design Facility Plus (SDF Plus) Installation and Operations Guide* for instructions on equating these names.

Example

```
IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

SPECIAL-NAMES.
   DICTIONARY IS "SDFPLUSDICTIONARY".
```

Invoking Data Descriptions in SDF Plus

You use a special data description entry to invoke a form record library from the dictionary. The syntax for the data description entry is as follows:

Format

```
level-number form-record-library-name-1
      FROM DICTIONARY
     [ { SAME RECORD AREA     } ]
     [ { SEPARATE RECORD AREA } ]
      [ VERSION IS literal-1 ]
     [ DIRECTORY IS { literal-2        } ]
                    { directory-name-1 }
      [ REDEFINES form-record-library-name-2 ]
      [ COMMON ]
      [ GLOBAL ]
      [ EXTERNAL ] . [;]
```

Explanation

level-number	You can invoke the 01-level data description entry within the File Section, the Working-Storage Section, the Linkage Section, or the Local-Storage Section. The level-number must be 01.
form-record-library-name-1	The form-record-library-name-1 identifies a collection of record descriptions for message types and transaction types. This must immediately follow the level-number. The REDEFINES clause, if present, must immediately follow form-record-library-name-1. All other clauses can be present in any order.
SAME RECORD AREA clause	The SAME RECORD AREA clause invokes all record descriptions in the form record library as redefinitions of the first record description in the form record library.

SEPARATE RECORD AREA clause	The SEPARATE RECORD AREA clause is used to invoke each record in the form record library as a separate data description, with its own record area.
VERSION clause	The VERSION clause invokes a specified version of the form record library. The most recent version is invoked by default if the version clause is omitted.
literal-1	This must be a numeric literal up to 6 digits long.
DIRECTORY clause	The DIRECTORY clause specifies the directory under which the form record library is stored in the data dictionary.
literal-2	This must be a nonnumeric literal up to 17 alphanumeric characters long.
directory-name-1	This must be a name of up to 17 alphanumeric characters long.
REDEFINES clause	The REDEFINES clause enables the same memory to be described by different data descriptions. You must have specified the SAME RECORD AREA clause in the data description of form-record-library-name-2 so that the records in form-record-library-name-1 redefine the first record in form-record-library-name-2. The records in form-record-library-name-1 redefine the first record in form-record-library-name-2.
COMMON	You must specify this in a subprogram for referencing a form record library declared in the host program when binding.
GLOBAL	You must specify this in the main program if all nested subprograms need access to the form record library.
EXTERNAL	You must specify this if the data structure is shared through interprogram communication (IPC) at run time.
; (Semicolon)	The semicolon that follows the ending period can be used to control the behavior of compiler control records (CCRs) and the format of listings. This semicolon should always be separated from the ending period of the data invocation by at least one space.

IF a CCR immediately follows a data item invoked from the dictionary, the compiler option changes might occur before the compiler processes the information invoked from the dictionary. This situation can be avoided by using the semicolon after the ending period. The semicolon ensures that the compiler processes the invoked information before the option actually changes.

Details

You can include a data description entry for a form record library in any of the following sections of the Data Division:

- File Section
- Working-Storage Section
- Linkage Section
- Local-Storage Section

Note, however, that you can use the SAME RECORD AREA, SEPARATE RECORD AREA, and REDEFINES clauses only with data description entries in the Working-Storage Section, the Linkage Section, and the Local-Storage Section.

If you do not specify the SAME RECORD AREA clause in your data description entry, separate record areas are assumed.

You cannot use the INVOKE clause to give an alias to a form record library.

You can only invoke a form record library directly from the dictionary. You cannot directly invoke either a transaction type or a message type.

Examples

In the following example, the form record library SAMPLELIB is imported from the dictionary:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SAMPLELIB FROM DICTIONARY.
```

In the following example, Version 2 of the form record library SAMPLELIB is imported from the data dictionary. The directory is SMITH.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SAMPLELIB FROM DICTIONARY
        VERSION IS 2
        DIRECTORY IS "SMITH".
```

In the following example, the form record library SAMPLELIB is redefined by SAMPLELIB3:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SAMPLELIB FROM DICTIONARY; SAME RECORD AREA.
01 SAMPLELIB3 FROM DICTIONARY;
         REDEFINES SAMPLELIB;
         SAME RECORD AREA.
```

See Also

- "Assigning Alias Identifiers" in Section 4.
- "Identifying Specific Entities" in Section 4.
- "Using the ADDS Program Interface" in Section 4.

Using SDF Plus Control Parameters

The application program uses a set of control parameters in conjunction with the CALL statements in order to communicate with SDF Plus at execution time. These control parameters are contained in a COPY library that must be included in the Working Storage Section of the application program.

The following paragraphs describe the COPY library and the control parameters.

SDF Plus COPY Library

After defining a form library, the user interface designer will request SDF Plus to generate a COBOL85 COPY library for the form library. The COPY library must be included in the application program. The COPY library contains the following types of information:

- Transaction number of each transaction
- Message number of each message
- Form record library description

Ensure that your application program does not modify the COPY library information.

The SDF Plus COPY library has the following layout:

```
   01 TRANSNUM-<form-record-library-name>.
      02 <transaction-type-name-1>.
           03 TRANSNUM  PIC 9(4) COMP VALUE   1.
      02 <transaction-type-name-2>.
           03 TRANSNUM  PIC 9(4) COMP VALUE   2.
      .
      .
      .
   01 MSGNUM-<form-record-library-name>.
      02 <message-type-name-1>M.
           03 MSGNUM  PIC 9(4) COMP VALUE   1.
      02 <message-type-name-2>M.
           03 MSGNUM  PIC 9(4) COMP VALUE   2.
      .
      .
      .
   01 FRLD-<form-record-library-name>.
 $ RESET LIST
      02 FILLER PIC X(12) VALUE @<24-hex-digits>@.
      .
      .
      .
      02 FILLER PIC X(12) VALUE @<24-hex-digits>@.
 $ POP LIST
```

The names of the data items in the COPY library cannot exceed 30 characters in length. Therefore, the names of the data items generated by SDF Plus might not include the full name of the form record library, as in the following example:

```
01 TRANSNUM-LONGFORMRECORDLIBRARY.
   .
   .
   .
 01 MSGNUM-LONGFORMRECORDLIBRARYNA.
   .
   .
   .
01 FRLD-LONGFORMRECORDLIBRARYNAME.
```

To include the COPY library in the program at compile time, you must include the following statement in the Working-Storage Section of the program:

```
COPY "SDFPLUS/COBOL/<directory-name>/<form-record-library-name>".
```

For example, to include the COPY library used by the example program at the end of this section, the following statement was used:

```
COPY "SDFPLUS/COBOL/SIMPLE/SIMPLEFL".
```

If the <form-record-library-name> is more than 17 characters long, the name is divided into two parts, with the first part containing 17 characters, as shown in the following example:

```
COPY "SDFPLUS/COBOL/UC/LONGFORMRECORDLIB/RARYNAME".
```

You can include more than one COPY library within a program.

Transaction Numbers

A unique transaction number is assigned to each transaction type within a form record library. The transaction numbers are given by the fields in the record called TRANSNUM-<form-record-library-name> within the COPY library. When an application program receives a message, it can use the transaction number to determine the form and transaction type to which the message applies.

The names of the transaction number fields are the same as the names of the transaction types defined in the form record library. Transaction type names are formed in SDF Plus by adding either TT or PTT to the end of the form name. TT is used for the update transaction type and PTT is used for the prefill transaction type. For example, to refer to the number of a particular transaction type such as SIMPLEENTRYTT, you would use the following code fragment in the Procedure Division of the program:

```
TRANSNUM OF SIMPLEENTRYTT
```

The transaction number is typically used to determine which transaction type was received, as in the following example:

```
IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEENTRYTT
     PERFORM HANDLE-SIMPLEENTRYTT THRU
             HANDLE-SIMPLEENTRYTT-EXIT
ELSE . . .
```

Message Numbers

A unique message number is assigned to each message type in a form record library. In some manuals, the term *form record number* is a synonym for *message number*.

The names of the message number fields are the same as the names of the message types defined in the form record library, except that the letter *M* is appended. Message number field names are formed in SDF Plus by adding M to the form name for update transaction request message types, PREM to the form name for prefill transaction message types, or by adding SRM to the form record library name for standard response message types. An *M* is appended to the names of the message numbers in order to distinguish them from the names of the message types themselves. The names of the message types are incorporated into the application program when the form record library is invoked.

To refer to the number of a particular message type, such as the standard response message type SIMPLEFLSR, the following code fragment appears in the Procedure Division of the program:

```
MSGNUM OF SIMPLEFLSRM
```

The message number is typically used to indicate the message type to be sent from the program to the user interface system. Therefore, this is how the code fragment would most likely appear in the program:

```
MOVE MSGNUM OF SIMPLEFLSRM TO SDFPLUS-MSGNUM.
```

Form Library Description

This description consists of such things as timestamps for the message types and transaction types, layouts of the message types, and form names for command messages. Since this information is encoded in binary, the A Series COBOL extension of designating a hex string by @<hex-characters>@ in the data description VALUE clause is used for this information.

The form record library description information is only used by SDF Plus; it is not used by the application program. Since this information can be extensive, RESET and POP of the LIST compiler control option surround this information.

Generating the COPY Library

The COPY library is generated in a way similar to generating a COMS processing item. This is accomplished by selecting a form library on the Form Library List form in SDF Plus and choosing the COBOL operation.

Additional SDF Plus Control Parameters

If a program uses COMS direct windows, then fields within the COMS header are used to exchange control information with SDF Plus. For a program that uses the CALL interface rather than the COMS interface, additional data items must be declared to serve a similar purpose. The following data items are used as parameters in the calls to SDF Plus:

```
77    SDFPLUS-RESULT        PIC S9(11) BINARY.
77    SDFPLUS-TRANSNUM      PIC  9(11) BINARY.
77    SDFPLUS-MSGNUM        PIC  9(11) BINARY.
77    SDFPLUS-TRANERROR     PIC  9(11) BINARY.
77    SDFPLUS-DEFAULTMSG    PIC  9(11) BINARY.
77    SDFPLUS-TEXTLENGTH    PIC  9(11) BINARY.
```

Since these parameters are required in every program that calls SDF Plus, they will appear as part of the COPY library generated for the form record library. Also, since you can include more than one COPY library in a program, the reserved compiler control option SDFPLUSPARAMETERS is used to ensure that these control parameters are included only once in the program.

Each parameter is described in detail in the following pages. Note that not every parameter is used in every call.

SDFPLUS-RESULT

This parameter is the result parameter and is used in every call. SDF Plus uses this parameter to indicate the success or failure of the call made.

Table 5–1 provides an interpretation of the values found in the SDFPLUS-RESULT field. The values and interpretation of this field are the same as the values and interpretation of the SDFINFO field of a COMS input header.

Table 5–1. Values and Meaning of SDFPLUS-RESULT Field

Result Number	Meaning
0	No error.
-100	Timestamp mismatch. SDF Plus keeps track of the timestamps of the message types and transaction types within the form library. These timestamps reflect the time of the last update to the message type or transaction type. When the COPY library is generated, timestamps for the transaction types and message types are placed into the form record library description record. Whenever a message or transaction is passed between SDF Plus and the program, the timestamp in the form library is compared with the timestamp in the form record library description record. If they do not match, which might happen if the form library is changed after the COPY library is generated, this result is returned. To obtain the correct timestamps, you should generate the COPY library and compile the program again.
-200	Invalid message type number. The message types are numbered from 1 to n, where n is the number of message types in the form record library. The program either set the message number to 0 or used a message number greater than n. This is most likely the result of a programming error. Correct the error and recompile the program.
-300	Invalid transaction type number. The transactions are numbered from 1 to n, where n is the number of transactions in the form record library. The program either set the transaction number to 0 or used a transaction number greater than n. This is most likely the result of a programming error. Correct the error and recompile the program.
-400	Invalid message key. This result is returned if the form library uses message keys and the last input from the end user did not contain a valid message key. The raw data received from the end user is placed into the data buffer. Both the message number and transaction number are set to 0.

SDFPLUS-TRANSNUM

After an application program receives a message from SDF Plus through the CALL interface, this parameter contains the transaction number of the data received. The programs checks the transaction number to determine which transaction type was received, as shown in the following example:

```
IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEENTRYTT
     PERFORM HANDLE-SIMPLEENTRYTT THRU
             HANDLE-SIMPLEENTRYTT-EXIT
ELSE IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEDISPLAYPTT
     PERFORM HANDLE-SIMPLEDISPLAYPTT THRU
             HANDLE-SIMPLEDISPLAYPTT-EXIT
ELSE . . .
```

The transaction number is also used when sending a transaction error. See "SDFPLUS-TRANERROR" in this section for more details.

SDFPLUS-MSGNUM

When an application program sends data to SDF Plus through the CALL interface, this parameter is used to indicate which message type the application program is sending. The program moves the message number of the appropriate message type to this parameter, as follows:

```
MOVE MSGNUM OF SIMPLEFLSRM TO SDFPLUS-MSGNUM.
```

A command message is a request from the application program to SDF Plus to display a form, allowing the program to control the sequence of forms. An application program can send a command message to SDF Plus by moving the message number of a command message to the SDFPLUS-MSGNUM parameter. A command message defined through SDF Plus has the same name as the form with an M appended. The following statement sets the message number properly for sending the command message:

```
MOVE MSGNUM OF SIMPLEDISPLAYM TO SDFPLUS-MSGNUM.
```

When an application program receives a message from SDF Plus through the CALL interface, this parameter contains the message number of the message type received. Although the program could check the message number rather than the transaction number to determine what was received, this method should be avoided because future changes to SDF Plus might cause unexpected results.

SDFPLUS-TRANERROR

The application program uses this parameter to indicate to SDF Plus that a transaction error has occurred. The application program moves a transaction error number to this field and sends a transaction error message. The SDFPLUS-TRANSNUM parameter is also required when sending a transaction error; however, it is always set correctly since it contains the transaction number of the last transaction received.

```
MOVE SE-ALPHAERROR TO SDFPLUS-TRANERROR.
```

SDFPLUS-DEFAULTMSG

When an application program sends a message to SDF Plus through the CALL interface, this parameter indicates whether or not the message contains data. The program sets this parameter to 0 if data is sent with the message and to 1 if there is no data.

Normally, the program sets this to 0. If the program sets this parameter to 1, the form library uses the default values for the fields in the message. This flag is set in conjunction with setting the message number field, as follows:

```
MOVE 1 TO SDFPLUS-DEFAULTMSG.
MOVE MSGNUM OF SIMPLEFLSRM TO SDFPLUS-MSGNUM.
```

SDFPLUS-TEXTLENGTH

When the program is sending text to the user interface system, the program sets this parameter to indicate the number of characters of text to be sent. The program can also set this to 0 to indicate that all of the text in the text record is to be sent. The text-length parameter is shown in the following example:

```
MOVE "Here is some text to display." TO TEXT-RECORD.
MOVE 29 TO SDFPLUS-TEXTLENGTH.
```

Run Time Support and Initialization

If your program uses COMS direct windows, then your program communicates to SDF Plus through the use of the SEND and RECEIVE verbs. If your program does not use COMS direct windows, your program uses the following four entry points to communicate with SDF Plus:

Your program uses . . .	**To . . .**
WAIT_FOR_TRANSACTION	Receive a transaction.
SEND_MESSAGE	Send a message.
SEND_TRANSACTION_ERROR	Send an error.
SEND_TEXT	Send text.

Each entry point is discussed in the following pages, as well as how to link to the SDF Plus Forms Support library.

COBOL85 programs that do **not** use COM direct windows access SDF Plus through the use of the CALL verb. Before calling SDF Plus, the program performs the following statements during initialization:

```
CHANGE ATTRIBUTE LIBACCESS    OF "SDFPLUS" TO BYFUNCTION.
CHANGE ATTRIBUTE FUNCTIONNAME OF "SDFPLUS" TO "FORMSSUPPORT".
```

These statements are necessary to properly link the program to the SDF Plus Forms Support library that is identified through the ODT *SL* command.

WAIT_FOR_TRANSACTION

This routine is called by the program in order to receive the next transaction from the user interface system. The syntax is as follows:

```
CALL "WAIT_FOR_TRANSACTION IN SDFPLUS"
     USING <description-record>,
           <data-structure>,
           SDFPLUS-TRANSNUM,
           SDFPLUS-MSGNUM,
           SDFPLUS-RESULT.
```

The parameters for the WAIT_FOR_TRANSACTION routine are as follows:

<description-record>	This is the form record library description record given in the COPY library.
<data-structure>	This is the working storage area into which the data sent from the user interface system to the program is to be placed. Typically this is a 01-level data structure, large enough to accept all of the data for the largest message type.
SDFPLUS-TRANSNUM SDFPLUS-MSGNUM SDFPLUS-RESULT	These are the SDF Plus control parameters discussed previously. The program does not set up any of the parameters before making the call. After the call, the program can check the SDFPLUS-TRANSNUM, SDFPLUS-MSGNUM, and SDFPLUS-RESULT parameters.

The program is waiting for a transaction in the following example:

```
CALL  "WAIT_FOR_TRANSACTION IN SDFPLUS"
     USING FRLD-SIMPLEFL,
           SIMPLEFL-RECORD,
           SDFPLUS-TRANSNUM,
```

```
            SDFPLUS-MSGNUM,
            SDFPLUS-RESULT.
```

SEND_MESSAGE

This routine is called by the program in order to send a message to the user interface system. The syntax is as follows:

```
  CALL "SEND_MESSAGE IN SDFPLUS"
       USING <description-record>,
             <data-structure>,
             SDFPLUS-MSGNUM,
             SDFPLUS-DEFAULTMSG,
             SDFPLUS-RESULT.
```

The parameters for the SEND_MESSAGE routine are as follows:

<description-record>	This is the form record library description record.
<data-structure>	This is the working storage area from which the data is sent to the user interface system.
SDFPLUS-MSGNUM SDFPLUS-DEFAULTMSG SDFPLUS-RESULT	These are the SDF Plus control parameters. The program must set up parameters for SDFPLUS-MSGNUM and SDFPLUS-DEFAULTMSG before making the call. After the call, the program can check the SDFPLUS-RESULT parameter.

The program is sending a message in the following example:

```
  CALL "SEND_MESSAGE IN SDFPLUS"
       USING FRLD-SIMPLEFL,
             SIMPLEFL-RECORD,
             SDFPLUS-MSGNUM,
             SDFPLUS-DEFAULTMSG,
             SDFPLUS-RESULT.
```

SEND_TRANSACTION_ERROR

This routine is called by the program in order to send a transaction error number to the user interface system. The syntax is as follows:

```
  CALL "SEND_TRANSACTION_ERROR IN SDFPLUS"
       USING <description-record>,
             SDFPLUS-TRANSNUM,
             SDFPLUS-TRANERROR,
             SDFPLUS-RESULT.
```

<description-record>	This is the form record library description record.
SDFPLUS-TRANSNUM SDFPLUS-TRANERROR SDFPLUS-RESULT	These are the SDF Plus control parameters. The program must set up parameters for SDFPLUS-TRANSNUM and SDFPLUS-TRANERROR before making the call. After the call, the program can check the SDFPLUS-RESULT parameter.

Since no data is sent with a transaction error, <data-structure> does not appear.

Note: *The call for SEND_TRANSACTION_ERROR only sends a transaction error number to SDF Plus; it does not cause SDF Plus to initiate transaction error processing. SDF Plus stores the transaction error number until the program calls either SEND_MESSAGE or WAIT_FOR_TRANSACTION. In most cases the program calls SEND_TRANSACTION_ERROR and then calls SEND_MESSAGE, to send the standard response, in order to initiate transaction error processing. The program might call SEND_TRANSACTION_ERROR more than once to indicate several errors before calling SEND_MESSAGE.*

The program is sending a transaction error in the following example:

```
CALL "SEND_TRANSACTION_ERROR IN SDFPLUS"
     USING FRLD-SIMPLEFL,
           SDFPLUS-TRANSNUM,
           SDFPLUS-TRANERROR,
           SDFPLUS-RESULT.
```

SEND_TEXT

This routine is called by the program in order to send some text to the user interface system. The syntax is as follows:

```
CALL "SEND_TEXT IN SDFPLUS"
     USING <text-data-structure>,
           SDFPLUS-TEXTLENGTH,
           SDFPLUS-RESULT.
```

The parameters for the SEND_TEXT routine are as follows:

<text-data-structure>	This must be a 01-level data structure that contains the text to be sent. If SDPLUS-TEXTLENGTH is set to zero, the entire data-structure is sent.

SDFPLUS-TEXTLENGTH SDFPLUS-RESULT	These are the SDF Plus control parameters. The program must set up the SDFPLUS-TEXTLENGTH parameter before making the call. After the call, the program can check the SDFPLUS-RESULT parameter.

Note: *The call for SEND_TEXT only sends a text message to SDF Plus; it does not cause SDF Plus to display the text. SDF Plus stores the text until the program calls either SEND_MESSAGE or WAIT_FOR_TRANSACTION. At that point SDF Plus appends the text message to the form that is displayed as a result of calling SEND_MESSAGE or WAIT_FOR_TRANSACTION. The program might call SEND_TEXT more than once before calling SEND_MESSAGE or SEND_TRANSACTION.*

Continuing the example, the program is sending some text contained in a data-structure called SOME-TEXT, as shown in the following example:

```
CALL "SEND_TEXT IN SDFPLUS"
     USING SOME-TEXT,
           SDFPLUS-TEXTLENGTH,
           SDFPLUS-RESULT.
```

Remote File

COBOL85 programs that use the CALL interface, rather than the COMS direct window interface, use remote files for communication. However, the remote file is not declared in the application program but in the SDF Plus Forms Support library.

Remote File READ and WRITE

SDF Plus owns the remote file, and that remote file is open for input and output to the end user. The program can declare a remote file, open it for output only, and send messages directly to the end user since several remote files can be opened for output only to a single station. However, sending messages directly to the end user, rather than calling "SEND_TEXT," is not recommended.

Only one input-capable remote file can be attached to a station. Since SDF Plus opens its remote file as input-capable, the program cannot open another input-capable remote file for that station.

Multi-User Remote File

If the program uses the CALL interface, and can handle multiple users, then the program should be declared in a COMS remote file window. Multiple users can then attach to the program by opening the corresponding window. The program cannot attach new stations to the remote file, since SDF Plus *owns* the remote file.

Debugging with TADS

COBOL85 application programs that use SDF Plus to manage the user interface can be debugged by using Test And Debug System (TADS). Although there are different ways that this might be accomplished, the following steps are suggested:

1. Place the TADS compiler option in your program:

   ```
   $$SET TADS
   ```

   (Do not declare a remote file in your program; TADS declares one for you.)

2. Compile the program.
3. Run the program with or without TADS.

   a. You can run the program without TADS by entering

      ```
      RUN MYPROG
      ```

   b. You can run the program with TADS, by providing the logical station number (LSN) of a physical station within your network. You can determine the LSN by entering the ?WRU command in COMS. Assuming that your physical LSN is 179, you can start up the program with a TADS session through CANDE as follows:

      ```
      RUN MYPROG; TADS; STATION=179
      ```

   The TADS option starts the TADS session. The STATION option redirects all remote files to the station with LSN 179. Two remote files are opened, one for the TADS session and the other for your SDF Plus application.

The requests to open the remote files are rerouted to COMS, which owns the physical station. You are notified that your MARC window contains a new message for you. Perform the following steps to access the remote files:

1. Switch to the MARC window.

   This window shows that, for example, window REM0001 is open.

2. Switch to the indicated window, REM0001.

   This gets you into the TADS session.

3. Set up your breakpoints and continue.

   SDF Plus opens its remote file, resulting in another message in the MARC window to inform you that a second window, REM002, is open.

4. Switch to the second window.

   This window displays the first form for your application.

From this point, you can interact with your application through window REM0002 and with TADS through window REM0001.

If you terminate the application, both windows will close.

Refer to the *COBOL ANSI-85 Test and Debug System (TADS) Programming Reference Manual* for additional information about TADS procedures and capabilities.

Using SDF Plus with COMS

You can use SDF Plus with COMS to take advantage of COMS direct windows. This feature gives you enhanced routing capabilities for message types and also enables preprocessing and postprocessing of message types.

Refer to the *Communications Management System (COMS) Programming Guide* for detailed information on the use of the COMS direct window interface.

The procedures for using SDF Plus with COMS are explained in the following pages.

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
- SDFTRANSNUM

A description of each field appears in the following paragraphs.

SDFINFO Field

When a program sends a message to SDF Plus, the SDFINFO field identifies the type of message processing being requested. After a program either sends or receives a message, this field contains status information indicating the success or failure of the preceding SEND or RECEIVE.

When the program sends a message, it specifies the type of message processing being requested by using the following values:

Value	Explanation
0	Normal form message processing
100	Transaction error processing (last error)
101	Transaction error processing (more to come)
200	Text message processing

After the program either sends or receives a message, SDF Plus indicates the success or failure of the message processing request by placing one of the following values in the SDFINFO field:

Value	Explanation
0	No error
-100	Form message timestamp mismatch
-200	Incorrect form record number specified in the send procedure
-300	Incorrect transaction number specified in the send procedure
-400	Invalid message key

The COMS SEND and RECEIVE verbs are used to simulate the four basic entry points of the SDF Plus CALL interface as shown in the following table:

The SDF Plus CALL interface entry point . . .	Is simulated by the COMS . . . verb.
WAIT_FOR_TRANSACTION	RECEIVE
SEND_MESSAGE	SEND
SEND_TRANSACTION_ERROR	SEND
SEND_TEXT	SEND

The following table shows how the COMS header fields are used in place of the control parameters used by the SDF Plus CALL interface:

SDF Plus CALL interface	SDFINFO COMS Header Field	SDFFORMRECNUM COMS Header Field	SDFTRANSNUM COMS Header Field
WAIT_FOR_TRANSACTION	Before RECEIVE: Not used After RECEIVE: SDFPLUS-RESULT	SDFPLUS-MSGNUM	SEND_MESSAGE
SEND_MESSAGE	Before SEND: 0 After SEND: SDFPLUS_RESULT	SDFPLUS-MSGNUM	Not used
SEND_TRANSACTION_ERROR	Before SEND 101 After SEND: SDFPLUS-RESULT	SDFPLUS-TRANERROR	SDFPLUS-TRANSNUM

SDF Plus CALL interface	SDFINFO COMS Header Field	SDFFORMRECNUM COMS Header Field	SDFTRANSNUM COMS Header Field
SEND_TEXT	Before SEND: 200 After SEND: SDFPLUS-RESULT	Not used	Not used

Notes:

The TEXTLENGTH field in the COMS output header can be set to zero (0), which is equivalent to SDFPLUS_DEFAULTMSG set to 1.

You can use the value 100, however, this value corresponds to SEND_TRANSACTION_ERROR followed by a SEND_MESSAGE of the response message for the transaction. This prevents the program from indicated several transaction errors. It is recommended that you do not use the value 100.

The TEXTLENGTH field in the COMS output header is used in the same way as SDFPLUS_TEXTLENGTH.

See "SDFPLUS_RESULT" earlier in this section for detailed explanations of the values found in the SDFINFO field of the COMS header.

SDFFORMRECNUM Field

The SDFFORMRECNUM field is used to specify the message type to be sent (on output) or the message type that was received (on input).

SDFTRANSNUM Field

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, you should follow the usual statements for each product, with the following guidelines:

- For sending messages, the application program must first move the value 0 (zero) into the SDFINFO field of the output header. The application program must also move the message number of the form record library into the SDFFORMRECNUM field. The buffer of the form record library must be passed as the message area in the SEND statement.

- To receive a message, the application program must do the following:
  - If the SDFINFO field contains a value less than 0 (zero), then this field contains an error code that indicates a problem with message processing. In addition, the FUNCTION-INDEX field of the input header will contain the value 100.
  - If the SDFINFO field contains the value 0 (zero), then the application program can query the message 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 library.

To send transaction errors, the user application must do the following:

- Move the value 101 into the SDFINFO field of the output header.
- Move the value of the transaction error into the SDFFORMRECNUM field of the output header.
- Move the SDFTRANSNUM field from the input header to the output header.
- Send the output header to display the message.

The user program can send any arbitrary message area along with the output header. SDF Plus will process only the information within the output header.

Example

In this example, INX is assigned the number of the transaction error. SDF-BUFFER is the user-defined buffer area.

```
MOVE 101 TO SDFINFO OF COMS-OUT .
MOVE INX TO SDFFORMRECNUM OF COMS-OUT.
MOVE SDFTRANSNUM OF COMS-IN TO
   SDFTRANSNUM OF COMS-OUT .
SEND COMS-OUT FROM SDF-BUFFER.
```

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, your program must do the following:

- Move the value 200 into the SDFINFO field of the output header.
- Move the text message into the message area to be sent through COMS.
- Use the SEND statement to store the text message.
- Move 0 (zero) to the SDFINFO field of the output header.

- Send the response message type to display the text message.

For information about the extensions used with COMS, refer to Section 2, "Using the COMS Program Interface."

Example

In this example, literal text is moved into the message area.

```
MOVE 200 TO SDFINFO OF COMS-OUT .
MOVE "This is an example of application text" TO SDF-BUFFER .
SEND COMS-OUT FROM SDF-BUFFER .
MOVE 0 TO SDFINFO OF COMS-OUT .
MOVE MSGNUM OF EXAMPLEFLSRM TO
   SDFFORMRECNUM OF COMS-OUT .
SEND COMS-OUT FROM SDF-BUFFER.
```

Specific Differences between COBOL74 and COBOL85

The following tables compare the COBOL74 and COBOL85 programmatic interfaces to SDF Plus. The first table shows the differences in the syntax when converting COBOL74 programs that use the remote file interface or the COMS interface into COBOL85 programs that use the CALL interface or the COMS interface. The second table compares only the differences between COBOL74 programs that used the remote file interface and COBOL85 programs that now use the CALL interface.

Syntax Applicable to All SDF Plus Programs

Tables 5–2 through 5–4 show the syntax required to convert a COBOL74 program that uses either the remote file interface or the COMS interface into a COBOL85 program that uses the COMS interface or the CALL interface.

Table 5–2. Syntax for Invoking a Form Record Library

COBOL74 Syntax	COBOL85 Syntax
DICTIONARY IS DATADICTIONARY	DICTIONARY IS DATADICTIONARY
01 SIMPLEFL FROM DICTIONARY. . . For the COMS interface, this syntax appears in the Working-Storage Section. For the remote file interface, this syntax appears in the File Section.	01 SIMPLEFL FROM DICTIONARY. . . For the COMS and the CALL interface, this syntax appears in the Working-Storage Section.

Table 5–2. Syntax for Invoking a Form Record Library

COBOL74 Syntax	COBOL85 Syntax
Transaction numbers, message numbers, and the form record library description are compiled into the program as part of the invoking of the form record library.	COPY SDFPLUS/COBOL/SIMPLE/SIMPLEFL. The COBOL85 COPY library contains the information on transaction numbers, message numbers, and the form record library description.

Table 5–3. Accessing Message Numbers

COBOL74 Syntax	COBOL85 Syntax
ATTRIBUTE FORMRECNUM OF SIMPLEFLSR	MSGNUM OF SIMPLEFLSRM Note that all message number fields end with M.
CHANGE ATTRIBUTE FORMRECNUM OF SIMPLEFL TO ATTRIBUTE FORMRECNUM OF SIMPLEFLSR.	MOVE MSGNUM OF SIMPLEFLSRM TO SDFPLUS-MSGNUM.

Table 5–4. Accessing Transaction Numbers

COBOL74 Syntax	COBOL85 Syntax
ATTRIBUTE TRANSNUM OF SIMPLEENTRYTT	TRANSNUM OF SIMPLEENTRYTT
IF ATTRIBUTE TRANSNUM OF SIMPLEFL = ATTRIBUTE TRANSNUM OF SIMPLEENTRYTT PERFORM HANDLE-SIMPLEENTRYTT ELSE . . .	IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEENTRYTT PERFORM HANDLE- SIMPLEENTRYTT ELSE...

For SDF Plus application programs that use the COMS interface, other than the items mentioned in the preceding table, there is no difference when using SDF Plus between a COBOL74 program and a COBOL85 program. However, if the COBOL74 program originally used SDF for its user interface, and currently uses the FORM-KEY function, you must change the program to use SDFFORMRECNUM and SDFINFO, as shown in the following example:

```
MOVE FORM-KEY(FORM1) TO COMS-OUT-CONVERSATION.
```

The preceding example becomes

```
MOVE MSGNUM OF FORM1M TO SDFFORMRECNUM OF COMS-OUT.
MOVE 0 TO SDFINFO OF COMS-OUT.
```

Differences between a COBOL74 Remote File Interface Program and a COBOL85 CALL Interface Program

Table 5–5 shows the syntax that is applicable when converting a COBOL74 program that uses the remote file interface into a COBOL85 program that uses the CALL interface.

Table 5–5. Converting a COBOL74 Remote File Program into a COBOL85 CALL Interface Program

COBOL74 Calls to SDF Plus	COBOL85 Calls to SDF Plus
OPEN I-O REMOTEFILENAME.	CHANGE ATTRIBUTE LIBACCESS OF "SDFPLUS" TO BYFUNCTION. CHANGE ATTRIBUTE FUNCTIONNAME OF "SDFPLUS" TO "FORMSSUPPORT".
READ/WRITE FORM . . . ON ERROR . . . You have the option of handling errors within the program.	CALL . . . USING . . . SDFPLUS-RESULT. The program must always check the SDFPLUS-RESULT parameter after making a CALL. This is similar to declaring an ON ERROR parameter.
READ FORM RMT USING SIMPLEFL.	CALL "WAIT FOR TRANSACTION IN SDFPLUS" USING FRLD-SIMPLEFL SIMPLEFL-RECORD SDFPLUS-TRANSNUM SDFPLUS-MSGNUM SDFPLUS-RESULT.
WRITE FORM SIMPLEFL	CALL "SEND MESSAGE IN SDFPLUS" USING FRLD-SIMPLEFL SIMPLEFL-RECORD SDFPLUS-MSGNUM SDFPLUS-DEFAULTMSG SDFPLUS-RESULT.
WRITE FORM SIMPLEENTRY FOR ERROR MESSAGE 1.	MOVE 1 TO SDFPLUS-TRANERROR. CALL "SEND TRANSACTION ERROR IN SDFPLUS" USING FRLD-SIMPLEFL SDFPLUS-TRANSNUM SDFPLUS-TRANERROR SDFPLUS-RESULT.
WRITE FORM SIMPLEENTRY USING TEXT BIG-NUMBER FOR 40 CHARACTERS. The FOR <integer> CHARACTERS parameter is optional. If not given, the length of record-name-1 is used.	MOVE 40 TO SDFPLUS-TEXTLENGTH. CALL "SEND TEXT IN SDFPLUS" USING BIG-NUMBER SDFPLUS-TEXTLENGTH SDFPLUS-RESULT. To use the length of the <text-data-structure> parameter, use MOVE 0 to SDFPLUS-TEXTLENGTH.

Sample SDF Plus Programs

The following sample programs use the SDF Plus interface syntax. Listings of the form record library that was invoked from the dictionary and the SDF Plus COPY library are shown, followed by a program using the CALL interface and a program using the COMS interface.

Form Record Library

```
*--DICTIONARY DIRECTORY : SIMPLE.
*--DICTIONARY FORMLIST< SIMPLEFL >.
*--SDF TRANSACTION( SIMPLEDISPLAYPTT ).
 01 SIMPLEDISPLAYPRE.
       04 SIMPLEALPHA1 PIC X(40)
       04 SIMPLENUMERIC1 REAL.
       04 SIMPLEBOOLEAN1 PIC 9(1) COMP.
       04 SIMPLEDATE1 PIC 9(8) COMP.
       04 SIMPLETIME1 PIC 9(6).
*01 SIMPLEDISPLAYPRE REDEFINES SIMPLEDISPLAYPRE.
*--SDF TRANSACTION( SIMPLEDISPLAYTT ).
 01 SIMPLEDISPLAY REDEFINES SIMPLEDISPLAYPRE.
       04 SIMPLEALPHA1 PIC X(40).
       04 SIMPLENUMERIC1 REAL.
       04 SIMPLEBOOLEAN1 PIC 9(1) COMP.
       04 SIMPLEDATE1 PIC 9(8) COMP.
       04 SIMPLETIME1 PIC 9(6).
 01 SIMPLEFLSR REDEFINES SIMPLEDISPLAYPRE.
       04 SIMPLEFLSRF PIC X(1).
*--SDF TRANSACTION( SIMPLEENTRYTT ).
 01 SIMPLEENTRY REDEFINES SIMPLEDISPLAYPRE.
       04 SIMPLEALPHA PIC X(40).
       04 SIMPLENUMERIC REAL.
       04 SIMPLEBOOLEAN PIC 9(1) COMP.
       04 SIMPLEDATE PIC 9(8) COMP.
       04 SIMPLETIME PIC 9(6).
*01 SIMPLEFLSR REDEFINES SIMPLEDISPLAYPRE.
*--SDF TRANSACTION( SIMPLEQUITTT ).
 01 SIMPLEQUIT REDEFINES SIMPLEDISPLAYPRE.
       04 SIMPLESTUPIDFIELDFORCOBOL PIC X(1).
*01 SIMPLEFLSR REDEFINES SIMPLEDISPLAYPRE.
*01 SIMPLEDISPLAY REDEFINES SIMPLEDISPLAYPRE.
*01 SIMPLEENTRY REDEFINES SIMPLEDISPLAYPRE.
*01 SIMPLEQUIT REDEFINES SIMPLEDISPLAYPRE.*
```

COPY Library

```
*==> The SDF Plus control parameters:
$ SET OMIT = SDFPLUSPARAMETERS
 77  SDFPLUS-RESULT                PIC S9(11) BINARY.
     88  SDFPLUS-RESULT-OK                    VALUE    0.
     88  SDFPLUS-RESULT-TSMISMATCH            VALUE -100.
     88  SDFPLUS-RESULT-INVALIDMSGNUM         VALUE -200.
     88  SDFPLUS-RESULT-INVALIDTRANSNUM       VALUE -300.
     88  SDFPLUS-RESULT-INVALIDMSGKEY         VALUE -400.
 77  SDFPLUS-TRANSNUM              PIC S9(11) BINARY.
 77  SDFPLUS-MSGNUM                PIC S9(11) BINARY.
 77  SDFPLUS-TRANERROR             PIC S9(11) BINARY.
 77  SDFPLUS-DEFAULTMSG            PIC S9(11) BINARY.
 77  SDFPLUS-TEXTLENGTH            PIC S9(11) BINARY.
$ POP OMIT
$ SET SDFPLUSPARAMETERS
 *==> The transaction number enumeration:
 01  TRANSNUM-SIMPLEFL.
     02  SIMPLEDISPLAYPTT.
         03  TRANSNUM                  PIC 9(4) COMP VALUE    1.
     02  SIMPLEDISPLAYTT.
         03  TRANSNUM                  PIC 9(4) COMP VALUE    2.
     02  SIMPLEENTRYTT.
         03  TRANSNUM                  PIC 9(4) COMP VALUE    3.
     02  SIMPLEQUITTT.
         03  TRANSNUM                  PIC 9(4) COMP VALUE    4.
*==> The message number enumeration:
 01  MSGNUM-SIMPLEFL.
     02  SIMPLEDISPLAYPREM.
         03  MSGNUM                    PIC 9(4) COMP VALUE    1.
     02  SIMPLEDISPLAYM.
         03  MSGNUM                    PIC 9(4) COMP VALUE    2.
     02  SIMPLEFLSRM.
         03  MSGNUM                    PIC 9(4) COMP VALUE    3.
     02  SIMPLEENTRYM.
         03  MSGNUM                    PIC 9(4) COMP VALUE    4.
     02  SIMPLEQUITM.
         03  MSGNUM                    PIC 9(4) COMP VALUE    5.
*==> The form record library description record:
 01  FRLD-SIMPLEFL.
$ RESET LIST
     02  FILLER PIC X(12) VALUE @00000000400054E5540EB3922@.
     02  FILLER PIC X(12) VALUE @4E5540F0974B4E5540F34FB6@.
*    ... <many more FILLER entries>
     02  FILLER PIC X(12) VALUE @100A3008100C204E100C3006@.
$ POP LIST
```

COBOL85 CALL Interface Program

```
$$ SET LIST WARNSUPR
*****************************************************************
*                                                               *
*                       COBOL85 EXAMPLE                         *
*                                                               *
*    This is an example showing the SDF Plus Interface in       *
*    COBOL85.                                                   *
*                                                               *
*****************************************************************
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
     DICTIONARY IS "SDFPLUSDICTIONARY".
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  BOOLEANS.
     02  BFALSE                  PIC 9 COMP     VALUE 0.
     02  BTRUE                   PIC 9 COMP     VALUE 1.
     02  ALL-DONE-FLAG           PIC 9 COMP.
         88  ALL-DONE                           VALUE 1.
 01  BIG-NUMBER.
     02  FILLER                  PIC X(40)
         VALUE "A really big number was entered!".
 77  DISP1                       PIC -ZZZ9.
 77  DISP2                       PIC -ZZZ9.
 01  SAVE-LAST-DATA.
     02  SAVE-ALPHA.
         03  SAVE-ALPHA-CHAR OCCURS 40 TIMES PIC X.
     02  SAVE-NUMERIC            REAL.
     02  SAVE-BOOLEAN            PIC 9 COMP.
     02  SAVE-DATE               PIC 9(8) COMP.
     02  SAVE-TIME               PIC 9(6).
/
*****************************************************************
*                                                               *
*                     FORM LIBRARY INFO                         *
*                                                               *
*****************************************************************
*==> This is information imported from the dictionary.  This
*    consists of the layouts of the messages:
 01  SIMPLEFL FROM DICTIONARY
       SAME RECORD AREA
       DIRECTORY IS "SIMPLE".
 01  SIMPLEFL-RECORD                  PIC X(100).
*==> This is information supplied by SDF Plus regarding the
*    same form library:
     COPY "SDFPLUS/COBOL/SIMPLE/SIMPLEFL".
*==> These are the transaction errors:
 01  TRANERR-SIMPLEENTRYTT.
```

```
     02  SE-ALPHAERROR                  PIC 9(4) COMP VALUE    1.
/
*****************************************************************
*                                                               *
*                      MAIN BODY OF PROGRAM                     *
*                                                               *
*****************************************************************
 PROCEDURE DIVISION.
 MAIN-BODY-OF-PROGRAM.
*==> Set up the linkage to the SDF Plus runtime support library:
     CHANGE ATTRIBUTE LIBACCESS    OF "SDFPLUS" TO BYFUNCTION.
     CHANGE ATTRIBUTE FUNCTIONNAME OF "SDFPLUS" TO "FORMSSUPPORT".
*==> Now we are ready to accept transactions from the form
*    library.  We will do this until told to stop:
     MOVE BFALSE TO ALL-DONE-FLAG.
     PERFORM WAIT-FOR-TRANSACTION THRU
             WAIT-FOR-TRANSACTION-EXIT
         UNTIL ALL-DONE.
*==> All done:
 END-OF-TASK.
     STOP RUN.
/
*****************************************************************
*                                                               *
*                     WAIT FOR TRANSACTION                      *
*                                                               *
*****************************************************************
 WAIT-FOR-TRANSACTION.
*==> Get the transaction:
     CALL "WAIT_FOR_TRANSACTION IN SDFPLUS"
         USING FRLD-SIMPLEFL,
               SIMPLEFL-RECORD,
               SDFPLUS-TRANSNUM,
               SDFPLUS-MSGNUM,
               SDFPLUS-RESULT.
*==> After any CALL to SDF Plus, you should check the result:
     IF NOT SDFPLUS-RESULT-OK
         PERFORM CHECK-SDFPLUS-RESULT THRU
                 CHECK-SDFPLUS-RESULT-EXIT.
*==> Determine which transaction this is and call a routine
*    to handle that specific transaction:
     IF      SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEENTRYTT
         PERFORM SIMPLEENTRY-TRANSACTION THRU
                 SIMPLEENTRY-TRANSACTION-EXIT
     ELSE IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEDISPLAYPTT
         PERFORM SIMPLEDISPLAY-TRANSACTION THRU
                 SIMPLEDISPLAY-TRANSACTION-EXIT
*==> If this is the Quit transaction, we do not send back any
*    response.  Rather, we just set the flag to indicate that
*    we are done:
     ELSE IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEQUITTT
         MOVE BTRUE TO ALL-DONE-FLAG
```

```
*==> If it is not a recognized transaction, just respond with the
*    standard response message.  DefaultMsg is set to 1 to
*    indicate that no data is being sent for the message:
     ELSE
         MOVE MSGNUM OF SIMPLEFLSRM TO SDFPLUS-MSGNUM
         MOVE BTRUE                 TO SDFPLUS-DEFAULTMSG
         CALL "SEND_MESSAGE IN SDFPLUS"
             USING FRLD-SIMPLEFL,
                   SIMPLEFL-RECORD,
                   SDFPLUS-MSGNUM,
                   SDFPLUS-DEFAULTMSG,
                   SDFPLUS-RESULT
         IF NOT SDFPLUS-RESULT-OK
             PERFORM CHECK-SDFPLUS-RESULT THRU
                     CHECK-SDFPLUS-RESULT-EXIT.
 WAIT-FOR-TRANSACTION-EXIT.
     EXIT.
/
*********************************************************************
*                                                                   *
*                    SIMPLE ENTRY TRANSACTION                       *
*                                                                   *
*    The update transaction for the Simple Entry form has been      *
*    received.  The data in the message will be written to          *
*    disk and also saved.                                           *
*                                                                   *
*********************************************************************
 SIMPLEENTRY-TRANSACTION.
*==> Save the data, but only if the alpha field is not blank.
*    A blank alpha field signifies to the form that it is to
*    terminate.  In such a case, the data is not valid:
     MOVE SIMPLEFL-RECORD TO SIMPLEENTRY.
     IF SIMPLEALPHA NOT = SPACES
         MOVE SIMPLEALPHA   TO SAVE-ALPHA
         MOVE SIMPLENUMERIC TO SAVE-NUMERIC
         MOVE SIMPLEBOOLEAN TO SAVE-BOOLEAN
         MOVE SIMPLEDATE    TO SAVE-DATE
         MOVE SIMPLETIME    TO SAVE-TIME.
*==> If the first and last characters of the alpha field do
*    not match, send a transaction error.  For a transaction
*    error, both transnum and tranerror must be set, but
*    we know that transnum is already set so we will not
*    set it again:
     IF SAVE-ALPHA-CHAR (1) NOT = SAVE-ALPHA-CHAR (40)
         MOVE SE-ALPHAERROR TO SDFPLUS-TRANERROR
         CALL "SEND_TRANSACTION_ERROR IN SDFPLUS"
             USING FRLD-SIMPLEFL,
                   SDFPLUS-TRANSNUM,
                   SDFPLUS-TRANERROR,
                   SDFPLUS-RESULT
         IF NOT SDFPLUS-RESULT-OK
             PERFORM CHECK-SDFPLUS-RESULT THRU
```

```
                    CHECK-SDFPLUS-RESULT-EXIT.
*==> If the numeric field is too big, send a text message:
     IF SAVE-NUMERIC > 500000.00
         MOVE 0 TO SDFPLUS-TEXTLENGTH
         CALL "SEND_TEXT IN SDFPLUS"
             USING BIG-NUMBER,
                   SDFPLUS-TEXTLENGTH
                   SDFPLUS-RESULT
         IF NOT SDFPLUS-RESULT-OK
             PERFORM CHECK-SDFPLUS-RESULT THRU
                     CHECK-SDFPLUS-RESULT-EXIT.
*==> All update transactions expect the standard response
*    message as a response.  The form waits until this
*    response message, or a command message, is sent by the
*    program.  If the alpha field begins with the backslash (\)
*    character, we will do a command write of the display form.
*    If not, we just send the standard response.  Either way,
*    only the message header is sent (no data):
     IF SAVE-ALPHA-CHAR (1) = "\"
         MOVE MSGNUM OF SIMPLEDISPLAYM TO SDFPLUS-MSGNUM
     ELSE
         MOVE MSGNUM OF SIMPLEFLSRM    TO SDFPLUS-MSGNUM.
     MOVE BTRUE TO SDFPLUS-DEFAULTMSG.
     CALL "SEND_MESSAGE IN SDFPLUS"
         USING FRLD-SIMPLEFL,
               SIMPLEFL-RECORD,
               SDFPLUS-MSGNUM,
               SDFPLUS-DEFAULTMSG,
               SDFPLUS-RESULT.
     IF NOT SDFPLUS-RESULT-OK
         PERFORM CHECK-SDFPLUS-RESULT THRU
                 CHECK-SDFPLUS-RESULT-EXIT.
 SIMPLEENTRY-TRANSACTION-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                  SIMPLE DISPLAY TRANSACTION                   *
*                                                               *
*    The prefill transaction for the Simple Display form has    *
*    been received.  The saved data from the last data entry    *
*    transaction will be returned to the form.                  *
*                                                               *
*****************************************************************
 SIMPLEDISPLAY-TRANSACTION.
*==> Move the saved data to the prefill response message:
     MOVE SAVE-ALPHA   TO SIMPLEALPHA1   OF SIMPLEDISPLAYPRE.
     MOVE SAVE-NUMERIC TO SIMPLENUMERIC1 OF SIMPLEDISPLAYPRE.
     MOVE SAVE-BOOLEAN TO SIMPLEBOOLEAN1 OF SIMPLEDISPLAYPRE.
     MOVE SAVE-DATE    TO SIMPLEDATE1    OF SIMPLEDISPLAYPRE.
     MOVE SAVE-TIME    TO SIMPLETIME1    OF SIMPLEDISPLAYPRE.
     MOVE SIMPLEDISPLAYPRE TO SIMPLEFL-RECORD.
```

```
*==> For a prefill transaction, the request and response
*    messages are the same.  Therefore, we do not need
*    the MOVE since SDFPLUS-MSGNUM already contains the
*    message number for SIMPLEDISPLAYPRE:
*    MOVE MSGNUM OF SIMPLEDISPLAYPREM TO SDFPLUS-MSGNUM.
*==> Send the prefill message with data:
*    MOVE BFALSE TO SDFPLUS-DEFAULTMSG.
     CALL "SEND_MESSAGE IN SDFPLUS"
         USING FRLD-SIMPLEFL,
               SIMPLEFL-RECORD,
               SDFPLUS-MSGNUM,
               SDFPLUS-DEFAULTMSG,
               SDFPLUS-RESULT
     IF NOT SDFPLUS-RESULT-OK
         PERFORM CHECK-SDFPLUS-RESULT THRU
                 CHECK-SDFPLUS-RESULT-EXIT.
 SIMPLEDISPLAY-TRANSACTION-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                     CHECK SDF PLUS RESULT                     *
*                                                               *
*    This routine is called if there is an error indicated      *
*    by the last call to SDF Plus.  This routine simply         *
*    displays a message regarding the error indicated.          *
*                                                               *
*****************************************************************
 CHECK-SDFPLUS-RESULT.
*==> See what the SDF Plus result is and display appropriate message:
     IF      SDFPLUS-RESULT-TSMISMATCH
         MOVE SDFPLUS-TRANSNUM TO DISP1
         MOVE SDFPLUS-MSGNUM   TO DISP2
         DISPLAY "Time stamp mismatch, "
                 "TransNum = " DISP1,
                 ", MsgNum = " DISP2 "."
         MOVE BTRUE TO ALL-DONE-FLAG
     ELSE IF SDFPLUS-RESULT-INVALIDMSGNUM
         MOVE SDFPLUS-MSGNUM TO DISP1
         DISPLAY "Invalid message type number "
                 DISP1 "."
     ELSE IF SDFPLUS-RESULT-INVALIDTRANSNUM
         MOVE SDFPLUS-TRANSNUM TO DISP1
         DISPLAY "Invalid transaction type number "
                 DISP1 "."
     ELSE IF SDFPLUS-RESULT-INVALIDMSGKEY
         DISPLAY "Invalid message key."
     ELSE
         MOVE SDFPLUS-RESULT TO DISP1
         DISPLAY "Unknown SDF Plus result "
                 DISP1 ".".
 CHECK-SDFPLUS-RESULT-EXIT.
```

```
    EXIT.
```

COBOL85 COMS Interface Program

```
$$   SET LIST WARNSUPR
*****************************************************************
*                                                               *
*                        COBOL85 EXAMPLE                        *
*                                                               *
*    This is an example COBOL85 program which uses the COMS     *
*    interface to SDF Plus.                                     *
*                                                               *
*****************************************************************
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
     DICTIONARY IS "SDFPLUSDICTIONARY".
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  BOOLEANS.
     02  BFALSE                  PIC 9 COMP     VALUE 0.
     02  BTRUE                   PIC 9 COMP     VALUE 1.
     02  ALL-DONE-FLAG           PIC 9 COMP.
         88  ALL-DONE                           VALUE 1.
 01  BIG-NUMBER.
     02  FILLER                  PIC X(40)
         VALUE "A really big number was entered!".
 77  DISP1                       PIC -ZZZ9.
 77  DISP2                       PIC -ZZZ9.
 01  SAVE-LAST-DATA.
     02  SAVE-ALPHA.
         03  SAVE-ALPHA-CHAR OCCURS 40 TIMES PIC X.
     02  SAVE-NUMERIC            REAL.
     02  SAVE-BOOLEAN            PIC 9 COMP.
     02  SAVE-DATE               PIC 9(8) COMP.
     02  SAVE-TIME               PIC 9(6).
/
*****************************************************************
*                                                               *
*                 SDF PLUS RELATED INFORMATION                  *
*                                                               *
*****************************************************************
 01  SIMPLEFL FROM DICTIONARY
     ; SAME RECORD AREA
     ; DIRECTORY "SIMPLE".
 01  FRL-RECORD REDEFINES SIMPLEDISPLAY
                                 PIC X(100).
 77  FRL-RECORD-SIZE             PIC 9(4) COMP VALUE 100.
*==> This is the COPY library containing SDF Plus specific
*    information extracted from the data dictionary:
```

```
     COPY "SDFPLUS/COBOL/SIMPLE/SIMPLEFL".
*==> Transaction errors:
 77  SE-ALPHAERROR               PIC S9(11) BINARY VALUE 1.
*==> Used to determine if we received a transaction that
*    needs to be processed:
 77  TRANSACTION-RECEIVED-FLAG        PIC  9    COMP.
     88  TRANSACTION-RECEIVED                        VALUE 1.
*==> Used to determine if we may continue to process a message
*    even if the an error was encountered:
 77  CONTINUE-PROCESSING-FLAG         PIC  9    COMP.
     88  CONTINUE-PROCESSING                         VALUE 1.
*==> These values are used by the program to indicate
*    what is being sent:
 01  SDFPLUS-INFO.
     02  SDFPLUS-INFO-MSG             PIC S9(3) COMP VALUE    0.
     02  SDFPLUS-INFO-LAST-TRAN-ERR   PIC S9(3) COMP VALUE  100.
     02  SDFPLUS-INFO-TRAN-ERR        PIC S9(3) COMP VALUE  101.
     02  SDFPLUS-INFO-TEXT-MSG        PIC S9(3) COMP VALUE  200.
/
*****************************************************************
*                                                               *
*                   COMS RELATED INFORMATION                    *
*                                                               *
*****************************************************************
 77  COMS-NAME                   PIC X(72)  VALUE "COMSSUPPORT".
 77  COMS-CALL-ERROR             PIC S9(11) USAGE IS BINARY.
*==> Our agenda name, and its designator (which we will get
*    from COMS during initialization):
 01  COMS-AGENDA                 PIC X(17)  VALUE "SIMPLE".
 77  COMS-AGENDA-DESIGNATOR                 USAGE IS REAL.
*==> Used to display error messages:
 01  COMS-RECORD.
     02  COMS-TYPE               PIC X(20).
     02  COMS-NBR                PIC S9(5).
     02  COMS-DASH               PIC X(2).
     02  COMS-TEXT               PIC X(40).
*==> The various COMS input status key values:
 77  COMS-INPUT-STATUS           PIC S9(3) COMP.
     88  COMS-IS-OK                        VALUE  0.
     88  COMS-IS-CONTINUE                  VALUE 89, 92, 93, 99.
     88  COMS-IS-UNKNOWN-STATION           VALUE 20.
     88  COMS-IS-MSG-TRUNC                 VALUE 89.
     88  COMS-IS-RECOVERY-MSG              VALUE 92.
     88  COMS-IS-LAST-MSG-ABORT            VALUE 93.
     88  COMS-IS-INVALID-PROG-STATION      VALUE 94.
     88  COMS-IS-INVALID-AGENDA            VALUE 95.
     88  COMS-IS-TERMINATE                 VALUE 99.
     88  COMS-IS-ATTACHED                  VALUE 100.
*==> The various COMS input function values:
 77  COMS-INPUT-FUNCTION         PIC S9(3) COMP.
     88  COMS-IF-OK                        VALUE    0.
     88  COMS-IF-BADTCODE                  VALUE   -4.
```

```
     88  COMS-IF-NOTCODE                  VALUE   -5.
     88  COMS-IF-NOITEM                   VALUE  -10.
     88  COMS-IF-OPEN                     VALUE  -16.
     88  COMS-IF-ON                       VALUE  -17.
     88  COMS-IF-CLOSE                    VALUE  -50.
     88  COMS-IF-EOJ                      VALUE  -60.
     88  COMS-IF-DISABLE                  VALUE  -61.
     88  COMS-IF-REDUCED                  VALUE  -62.
     88  COMS-IF-BADMKEY                  VALUE -100.
*==> The various COMS output status key values:
 77  COMS-OUTPUT-STATUS          PIC S9(3) COMP.
     88  COMS-OS-OK                       VALUE  0.
     88  COMS-OS-MSG-TRUNC                VALUE 89.
     88  COMS-OS-RECOVERY-MSG             VALUE 92.
     88  COMS-OS-INVALID-PROG-STATION     VALUE 94.
     88  COMS-OS-INVALID-AGENDA           VALUE 95.
     88  COMS-OS-PROC-ITEM                VALUE 96.
/
*****************************************************************
*                                                               *
*                         COMS HEADERS                          *
*                                                               *
*****************************************************************
 COMMUNICATION SECTION.
 INPUT HEADER COMS-IN;
     PROGRAMDESG                IS COMS-IN-PROGRAM;
     FUNCTIONSTATUS             IS COMS-IN-FUNCTION-STATUS;
     FUNCTIONINDEX              IS COMS-IN-FUNCTION-INDEX;
     USERCODE                   IS COMS-IN-USERCODE;
     SECURITYDESG               IS COMS-IN-SECURITY-DESG;
     TRANSPARENT                IS COMS-IN-TRANSPARENT;
     VTFLAG                     IS COMS-IN-VT-FLAG;
     TIMESTAMP                  IS COMS-IN-TIMESTAMP;
     STATION                    IS COMS-IN-STATION;
     TEXTLENGTH                 IS COMS-IN-TEXT-LENGTH;
     STATUSVALUE                IS COMS-IN-STATUS-KEY;
     MESSAGECOUNT               IS COMS-IN-MESSAGE-COUNT;
     RESTART                    IS COMS-IN-RESTART;
     AGENDA                     IS COMS-IN-AGENDA;
     SDFINFO                    IS COMS-IN-SDFPLUS-INFO;
     SDFTRANSNUM                IS COMS-IN-SDFPLUS-TRANSNUM;
     SDFFORMRECNUM              IS COMS-IN-SDFPLUS-MSGNUM.
 OUTPUT HEADER COMS-OUT;
     DESTCOUNT                  IS COMS-OUT-COUNT;
     TEXTLENGTH                 IS COMS-OUT-TEXT-LENGTH;
     STATUSVALUE                IS COMS-OUT-STATUS-KEY;
     TRANSPARENT                IS COMS-OUT-TRANSPARENT;
     VTFLAG                     IS COMS-OUT-VT-FLAG;
     CONFIRMFLAG                IS COMS-OUT-CONFIRM-FLAG;
     CONFIRMKEY                 IS COMS-OUT-CONFIRM-KEY;
     DESTINATIONDESG            IS COMS-OUT-STATION;
     NEXTINPUTAGENDA            IS COMS-OUT-NEXT-IN-AGENDA;
```

```
     SETNEXTINPUTAGENDA          IS COMS-OUT-SET-NEXT-IN-AGENDA;
     RETAINTRANSACTIONMODE       IS COMS-OUT-RETAIN-TRANS-MODE;
     AGENDA                      IS COMS-OUT-AGENDA;
     SDFINFO                     IS COMS-OUT-SDFPLUS-INFO;
     SDFTRANSNUM                 IS COMS-OUT-SDFPLUS-TRANSNUM;
     SDFFORMRECNUM               IS COMS-OUT-SDFPLUS-MSGNUM.
/
*****************************************************************
*                                                               *
*                      MAIN BODY OF PROGRAM                     *
*                                                               *
*****************************************************************
 PROCEDURE DIVISION.
 MAIN-BODY-OF-PROGRAM.
*==> Sign on to COMS:
     PERFORM INITIALIZATION THRU
             INITIALIZATION-EXIT.
*==> Now we are ready to accept transactions from the form
*    library.  We will do this until told to stop:
     MOVE BFALSE TO ALL-DONE-FLAG.
     PERFORM HANDLE-TRANSACTIONS THRU
             HANDLE-TRANSACTIONS-EXIT
         UNTIL ALL-DONE.
*==> All done:
 END-OF-TASK.
     STOP RUN.
/
*****************************************************************
*                                                               *
*                        HANDLE TRANSACTIONS                    *
*                                                               *
*****************************************************************
 HANDLE-TRANSACTIONS.
*==> Get the transaction:
     PERFORM WAIT-FOR-TRANSACTION THRU
             WAIT-FOR-TRANSACTION-EXIT.
*==> Determine which transaction this is, and call a routine
*    to handle that specific transaction:
     IF      SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEENTRYTT
         PERFORM SIMPLEENTRY-TRANSACTION THRU
                 SIMPLEENTRY-TRANSACTION-EXIT
     ELSE IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEDISPLAYPTT
         PERFORM SIMPLEDISPLAY-TRANSACTION THRU
                 SIMPLEDISPLAY-TRANSACTION-EXIT
*==> If the Quit transaction is received, we will send back
*    the standard response:
     ELSE IF SDFPLUS-TRANSNUM = TRANSNUM OF SIMPLEQUITTT
         MOVE MSGNUM OF SIMPLEFLSRM TO SDFPLUS-MSGNUM
         MOVE BTRUE                 TO SDFPLUS-DEFAULTMSG
         PERFORM SEND-MESSAGE THRU
                 SEND-MESSAGE-EXIT
*==> If it is not a recognized transaction, just respond with the
```

```
*    standard response message.  DefaultMsg is set to 1 to
*    indicate that no data is being sent for the message
     ELSE
         MOVE MSGNUM OF SIMPLEFLSRM TO SDFPLUS-MSGNUM
         MOVE BTRUE                 TO SDFPLUS-DEFAULTMSG
         PERFORM SEND-MESSAGE THRU
                 SEND-MESSAGE-EXIT.
 HANDLE-TRANSACTIONS-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                    SIMPLE ENTRY TRANSACTION                   *
*                                                               *
*    The update transaction for the Simple Entry form has been  *
*    received.  The data in the message will be saved.          *
*                                                               *
*****************************************************************
 SIMPLEENTRY-TRANSACTION.
*==> Save the data, but only if the alpha field is not blank.
*    A blank alpha field signifies to the form that it is to
*    terminate.  In such a case, the data is not valid:
     IF SIMPLEALPHA NOT = SPACES
         MOVE SIMPLEALPHA   TO SAVE-ALPHA
         MOVE SIMPLENUMERIC TO SAVE-NUMERIC
         MOVE SIMPLEBOOLEAN TO SAVE-BOOLEAN
         MOVE SIMPLEDATE    TO SAVE-DATE
         MOVE SIMPLETIME    TO SAVE-TIME.
*==> If the first and last characters of the alpha field do
*    not match, send a transaction error.  For a transaction
*    error, both TransNum and TranError must be set, but we
*    know that TransNum is already set so we will not set it
*    again:
     IF SAVE-ALPHA-CHAR (1) NOT = SAVE-ALPHA-CHAR (40)
         MOVE SE-ALPHAERROR TO SDFPLUS-TRANERROR
         PERFORM SEND-TRANSACTION-ERROR THRU
                 SEND-TRANSACTION-ERROR-EXIT.
*==> If the numeric field is too big, send a text message:
     IF SAVE-NUMERIC > 500000.00
         MOVE 40 TO SDFPLUS-TEXTLENGTH
         MOVE BIG-NUMBER TO FRL-RECORD
         PERFORM SEND-TEXT THRU
                 SEND-TEXT-EXIT.
*==> All update transactions expect the standard response
*    message as a response.  The form waits until this
*    response message, or a command message, is sent by the
*    program.  If the alpha field begins with the backslash (\)
*    character, we will do a command write of the display form.
*    If not, we just send the standard response.  Either way,
*    only the message header is sent (no data):
     IF SAVE-ALPHA-CHAR (1) = "(\)"
         MOVE MSGNUM OF SIMPLEDISPLAYM TO SDFPLUS-MSGNUM
```

```
     ELSE
         MOVE MSGNUM OF SIMPLEFLSRM    TO SDFPLUS-MSGNUM.
     MOVE BTRUE TO SDFPLUS-DEFAULTMSG.
     PERFORM SEND-MESSAGE THRU
             SEND-MESSAGE-EXIT.
 SIMPLEENTRY-TRANSACTION-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                   SIMPLE DISPLAY TRANSACTION                  *
*                                                               *
*    The prefill transaction for the Simple Display form has    *
*    been received.  The saved data from the last data entry    *
*    transaction will be returned to the form.                  *
*                                                               *
*****************************************************************
 SIMPLEDISPLAY-TRANSACTION.
*==> Move the saved data to the prefill response message:
     MOVE SAVE-ALPHA   TO SIMPLEALPHA1   OF SIMPLEDISPLAYPRE.
     MOVE SAVE-NUMERIC TO SIMPLENUMERIC1 OF SIMPLEDISPLAYPRE.
     MOVE SAVE-BOOLEAN TO SIMPLEBOOLEAN1 OF SIMPLEDISPLAYPRE.
     MOVE SAVE-DATE    TO SIMPLEDATE1    OF SIMPLEDISPLAYPRE.
     MOVE SAVE-TIME    TO SIMPLETIME1    OF SIMPLEDISPLAYPRE.
*==> For a prefill transaction, the request and response
*    messages are the same.  Therefore, we do not need the
*    following MOVE since MsgNum is already set to the
*    message number for SimpleEntryPre:
*    MOVE MSGNUM OF SIMPLEENTRYPREM TO SDFPLUS-MSGNUM.
*==> Send the prefill message with data:
     PERFORM SEND-MESSAGE THRU
             SEND-MESSAGE-EXIT.
 SIMPLEDISPLAY-TRANSACTION-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                         INITIALIZATION                        *
*                                                               *
*****************************************************************
 INITIALIZATION.
*==> Let's "sign on" to COMS:
     CHANGE ATTRIBUTE LIBACCESS    OF "DCILIBRARY" TO BYFUNCTION.
     CHANGE ATTRIBUTE FUNCTIONNAME OF "DCILIBRARY" TO COMS-NAME.
     ENABLE INPUT COMS-IN KEY "ONLINE".
*==> We need to know the output agenda designator:
     CALL "GET_DESIGNATOR_USING_NAME IN DCILIBRARY"
         USING  COMS-AGENDA
                VALUE AGENDA
                COMS-AGENDA-DESIGNATOR
         GIVING COMS-CALL-ERROR.
     IF COMS-CALL-ERROR NOT = 0
```

```
*======> There is a problem with getting the agenda designator.
*        Give up:
         DISPLAY "Invalid agenda name: '"
                 COMS-AGENDA
                 "'"
         STOP RUN.
 INITIALIZATION-EXIT.
    EXIT.
/
*****************************************************************
*                                                               *
*                      WAIT FOR TRANSACTION                     *
*                                                               *
*    This routine is called on to get the next transaction      *
*    from the user interface.                                   *
*                                                               *
*****************************************************************
 WAIT-FOR-TRANSACTION.
*==> Have mixed feeling regarding what will happen:
     MOVE BFALSE TO TRANSACTION-RECEIVED-FLAG.
     MOVE BTRUE  TO CONTINUE-PROCESSING-FLAG.
*==> Wait for some input:
     RECEIVE COMS-IN MESSAGE INTO FRL-RECORD.
*==> Check the status key to see if the receive was good:
     MOVE COMS-IN-STATUS-KEY TO COMS-INPUT-STATUS.
     IF NOT COMS-IS-OK
         PERFORM HANDLE-COMS-IN-STATUS THRU
                 HANDLE-COMS-IN-STATUS-EXIT.
*==> Check the function status/index:
     MOVE COMS-IN-FUNCTION-STATUS TO COMS-INPUT-FUNCTION.
     IF NOT COMS-IF-OK
         PERFORM HANDLE-COMS-IN-FUNCTION THRU
                 HANDLE-COMS-IN-FUNCTION-EXIT.
*==> Get the SDF Plus information out of the header:
     MOVE COMS-IN-SDFPLUS-INFO     TO SDFPLUS-RESULT.
     MOVE COMS-IN-SDFPLUS-TRANSNUM TO SDFPLUS-TRANSNUM.
     MOVE COMS-IN-SDFPLUS-MSGNUM   TO SDFPLUS-MSGNUM.
*==> Check the SDF Plus result to see if there was an error:
     IF NOT SDFPLUS-RESULT-OK
         PERFORM CHECK-SDFPLUS-RESULT THRU
                 CHECK-SDFPLUS-RESULT-EXIT.
*==> Should we process this transaction:
     IF CONTINUE-PROCESSING
         MOVE BTRUE TO TRANSACTION-RECEIVED-FLAG.
 WAIT-FOR-TRANSACTION-EXIT.
     EXIT.
```

```
/
*****************************************************************
*                                                               *
*                          SEND MESSAGE                         *
*                                                               *
*    This routine is called on to send the response message,    *
*    which has been set up by the calling routine, to the       *
*    user interface.                                            *
*                                                               *
*****************************************************************
 SEND-MESSAGE.
*==> Send the response message:
     IF SDFPLUS-DEFAULTMSG = 1
         MOVE 0               TO COMS-OUT-TEXT-LENGTH
     ELSE
         MOVE FRL-RECORD-SIZE TO COMS-OUT-TEXT-LENGTH.
     MOVE SDFPLUS-INFO-MSG    TO COMS-OUT-SDFPLUS-INFO.
     MOVE SDFPLUS-MSGNUM      TO COMS-OUT-SDFPLUS-MSGNUM.
     PERFORM SEND-COMS-MESSAGE THRU
             SEND-COMS-MESSAGE-EXIT.
*==> See if SDF Plus gave us an error:
     IF NOT SDFPLUS-RESULT-OK
         PERFORM CHECK-SDFPLUS-RESULT THRU
                 CHECK-SDFPLUS-RESULT-EXIT.
*==> Assume that the next message will be sent with text:
     MOVE BFALSE TO SDFPLUS-DEFAULTMSG.
 SEND-MESSAGE-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                    SEND TRANSACTION ERROR                     *
*                                                               *
*    This routine is called on to return a transaction error    *
*    to the user interface.                                     *
*                                                               *
*****************************************************************
 SEND-TRANSACTION-ERROR.
*==> Send the transaction error:
     MOVE 0                     TO COMS-OUT-TEXT-LENGTH.
     MOVE SDFPLUS-INFO-TRAN-ERR TO COMS-OUT-SDFPLUS-INFO.
     MOVE SDFPLUS-TRANERROR     TO COMS-OUT-SDFPLUS-MSGNUM.
     MOVE SDFPLUS-TRANSNUM      TO COMS-OUT-SDFPLUS-TRANSNUM.
     PERFORM SEND-COMS-MESSAGE THRU
             SEND-COMS-MESSAGE-EXIT.
*==> See if SDF Plus gave us an error:
     IF NOT SDFPLUS-RESULT-OK
         PERFORM CHECK-SDFPLUS-RESULT THRU
                 CHECK-SDFPLUS-RESULT-EXIT.
 SEND-TRANSACTION-ERROR-EXIT.
     EXIT.
```

```
/
*****************************************************************
*                                                               *
*                          SEND TEXT                            *
*                                                               *
*    This routine is called on to return a text message to the  *
*    user interface.                                            *
*                                                               *
*****************************************************************
 SEND-TEXT.
*==> Send the text message:
     MOVE SDFPLUS-TEXTLENGTH    TO COMS-OUT-TEXT-LENGTH.
     MOVE SDFPLUS-INFO-TEXT-MSG TO COMS-OUT-SDFPLUS-INFO.
     PERFORM SEND-COMS-MESSAGE THRU
             SEND-COMS-MESSAGE-EXIT.
*==> See if SDF Plus gave us an error:
     IF NOT SDFPLUS-RESULT-OK
         PERFORM CHECK-SDFPLUS-RESULT THRU
                 CHECK-SDFPLUS-RESULT-EXIT.
 SEND-TEXT-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                      SEND COMS MESSAGE                        *
*                                                               *
*    The routine sends the message to COMS.                     *
*                                                               *
*****************************************************************
 SEND-COMS-MESSAGE.
*==> Fill in the generic COMS header info and send the message:
     MOVE 1                      TO COMS-OUT-COUNT.
     MOVE COMS-IN-STATION        TO COMS-OUT-STATION.
     MOVE COMS-AGENDA-DESIGNATOR TO COMS-OUT-AGENDA.
     SEND COMS-OUT FROM FRL-RECORD.
*==> See if COMS gave us an error:
     MOVE COMS-OUT-STATUS-KEY TO COMS-OUTPUT-STATUS.
     IF NOT COMS-OS-OK
         PERFORM HANDLE-COMS-OUT-STATUS THRU
                 HANDLE-COMS-OUT-STATUS-EXIT.
*==> Get the SDF Plus result out of the header:
     MOVE COMS-OUT-SDFPLUS-INFO TO SDFPLUS-RESULT.
 SEND-COMS-MESSAGE-EXIT.
     EXIT.
```

```
/
*****************************************************************
*                                                               *
*                      CHECK SDF PLUS RESULT                    *
*                                                               *
*    This routine is called if there is an error indicated      *
*    by the last call to SDF Plus.  This routine simply         *
*    displays a message regarding the error indicated.          *
*                                                               *
*****************************************************************
 CHECK-SDFPLUS-RESULT.
*==> See what the SDF Plus result is, print appropriate message:
     IF      SDFPLUS-RESULT-TSMISMATCH
         MOVE SDFPLUS-TRANSNUM TO DISP1
         MOVE SDFPLUS-MSGNUM   TO DISP2
         DISPLAY "Time stamp mismatch, TransNum = "
                 DISP1
                 ", MsgNum = "
                 DISP2
                 "."
         MOVE BTRUE TO ALL-DONE-FLAG
     ELSE IF SDFPLUS-RESULT-INVALIDMSGNUM
         MOVE SDFPLUS-MSGNUM TO DISP1
         DISPLAY "Invalid message type number "
                 DISP1
                 "."
     ELSE IF SDFPLUS-RESULT-INVALIDTRANSNUM
         MOVE SDFPLUS-TRANSNUM TO DISP1
         DISPLAY "Invalid transaction type number "
                 DISP1
                 "."
     ELSE IF SDFPLUS-RESULT-INVALIDMSGKEY
         DISPLAY "Invalid message key."
     ELSE
         MOVE SDFPLUS-RESULT TO DISP1
         DISPLAY "Unknown SDF Plus result "
                 DISP1
                 ".".
 CHECK-SDFPLUS-RESULT-EXIT.
     EXIT.
```

```
/
*****************************************************************
*                                                               *
*                     HANDLE COMS IN STATUS                     *
*                                                               *
*    The message just received had a status of other than 0.    *
*    This routine determines what the status was and displays   *
*    an appropriate message.                                    *
*                                                               *
*****************************************************************
 HANDLE-COMS-IN-STATUS.
*==> Can we continue to process the transaction:
     IF COMS-IS-CONTINUE
         MOVE BTRUE TO CONTINUE-PROCESSING-FLAG
     ELSE
         MOVE BFALSE TO CONTINUE-PROCESSING-FLAG.
*==> Get the status number ready:
     MOVE SPACES TO COMS-RECORD.
     MOVE "COMS In Status" TO COMS-TYPE.
     MOVE COMS-INPUT-STATUS TO COMS-NBR.
     MOVE ": " TO COMS-DASH.
*==> See what the status is, give appropriate message:
     IF      COMS-IS-UNKNOWN-STATION
         MOVE "Unknown station" TO COMS-TEXT
     ELSE IF COMS-IS-MSG-TRUNC
         MOVE "Message truncated" TO COMS-TEXT
     ELSE IF COMS-IS-RECOVERY-MSG
         MOVE "Recovery message" TO COMS-TEXT
     ELSE IF COMS-IS-LAST-MSG-ABORT
         MOVE "Last message caused abort" TO COMS-TEXT
     ELSE IF COMS-IS-INVALID-PROG-STATION
         MOVE "Invalid program or station designator"
           TO COMS-TEXT
     ELSE IF COMS-IS-INVALID-AGENDA
         MOVE "Invalid agenda designator" TO COMS-TEXT
     ELSE IF COMS-IS-TERMINATE
         MOVE BTRUE TO ALL-DONE-FLAG
         MOVE "COMS requests us to go down" TO COMS-TEXT
     ELSE IF COMS-IS-ATTACHED
         MOVE "Station already attached to another program"
           TO COMS-TEXT
     ELSE
         MOVE "Unknown COMS error" TO COMS-TEXT.
     DISPLAY COMS-RECORD.
 HANDLE-COMS-IN-STATUS-EXIT.
     EXIT.
```

```
/
*****************************************************************
*                                                               *
*                    HANDLE COMS IN FUNCTION                    *
*                                                               *
*    The message just received had a function status of other   *
*    than 0.  This means that either COMS sent an error status, *
*    or this message has a function index.                      *
*                                                               *
*****************************************************************
 HANDLE-COMS-IN-FUNCTION.
*==> Check the function status:
     IF COMS-INPUT-FUNCTION < -5
         MOVE BFALSE TO CONTINUE-PROCESSING-FLAG
     ELSE
         MOVE BTRUE TO CONTINUE-PROCESSING-FLAG
         IF COMS-INPUT-FUNCTION NOT < 0
             GO TO HANDLE-COMS-IN-FUNCTION-EXIT.
*==> Get the function number ready:
     MOVE SPACES TO COMS-RECORD.
     MOVE "COMS In Function" TO COMS-TYPE.
     MOVE COMS-INPUT-FUNCTION TO COMS-NBR.
     MOVE ": " TO COMS-DASH.
*==> See what the status is, give appropriate message:
     IF      COMS-IF-BADTCODE
         MOVE "Undefined trancode" TO COMS-TEXT
     ELSE IF COMS-IF-NOTCODE
         MOVE "Invalid trancode" TO COMS-TEXT
     ELSE IF COMS-IF-NOITEM
         MOVE "No processing item" TO COMS-TEXT
     ELSE IF COMS-IF-OPEN
         MOVE "Open notification" TO COMS-TEXT
         MOVE BTRUE TO CONTINUE-PROCESSING-FLAG
     ELSE IF COMS-IF-ON
         MOVE "On notification" TO COMS-TEXT
     ELSE IF COMS-IF-CLOSE
         MOVE "Close notification" TO COMS-TEXT
     ELSE IF COMS-IF-EOJ
         MOVE "COMS is shutting down" TO COMS-TEXT
         MOVE BTRUE TO ALL-DONE-FLAG
     ELSE IF COMS-IF-DISABLE
         MOVE "Program has been disabled" TO COMS-TEXT
         MOVE BTRUE TO ALL-DONE-FLAG
     ELSE IF COMS-IF-REDUCED
         MOVE "Activity reduced, mincopies exceeded"
           TO COMS-TEXT
         MOVE BTRUE TO ALL-DONE-FLAG
     ELSE IF COMS-IF-BADMKEY
         MOVE "COMS detected invalid message key"
           TO COMS-TEXT
         MOVE BTRUE TO CONTINUE-PROCESSING-FLAG
     ELSE
```

```
         MOVE "Unknown COMS function status" TO COMS-TEXT.
     DISPLAY COMS-RECORD.
 HANDLE-COMS-IN-FUNCTION-EXIT.
     EXIT.
/
*****************************************************************
*                                                               *
*                     HANDLE COMS OUT STATUS                    *
*                                                               *
*    The message just sent had a status key of other than 0.    *
*    This routine determines what the status was, and handles   *
*    it accordingly.                                            *
*                                                               *
*****************************************************************
 HANDLE-COMS-OUT-STATUS.
*==> Get the status number ready:
     MOVE SPACES TO COMS-RECORD.
     MOVE "COMS Out Status" TO COMS-TYPE.
     MOVE COMS-OUTPUT-STATUS TO COMS-NBR.
     MOVE ": " TO COMS-DASH.
*==> See what the status is, give appropriate message:
     IF      COMS-OS-MSG-TRUNC
         MOVE "Message truncated" TO COMS-TEXT
     ELSE IF COMS-OS-RECOVERY-MSG
         MOVE "Message discarded due to recovery"
           TO COMS-TEXT
     ELSE IF COMS-OS-INVALID-PROG-STATION
         MOVE "Invalid program or station designator"
           TO COMS-TEXT
     ELSE IF COMS-OS-INVALID-AGENDA
         MOVE "Invalid agenda designator" TO COMS-TEXT
     ELSE IF COMS-OS-PROC-ITEM
         MOVE "Message prematurely stopped by proc item"
           TO COMS-TEXT
     ELSE
         MOVE "Unknown COMS error" TO COMS-TEXT.
     DISPLAY COMS-RECORD.
 HANDLE-COMS-OUT-STATUS-EXIT.
     EXIT.
```

Section 6
Using the SDF Program Interface

The Screen Design Facility (SDF) is a tool to help programmers design and process forms for applications. SDF provides form processing that eliminates the need for complicated format language or code, and it enables you to provide validation for data entered on forms by application users.

The COBOL85 program interface developed for SDF includes the following:

- Extensions that enable you to easily read and write forms
- Ability to invoke form data into your program as COBOL85 declarations
- Message keys for form processing
- Programmatic control over data manipulation and display on a form image

This section provides information about the extensions developed for SDF and explains the syntax for using message keys and programmatic controls in an application. Each extension is presented with its syntax and an example. Sample programs are included at the end of the section.

You can use SDF with the Advanced Data Dictionary System (ADDS) and with the Communications Management System (COMS). When you use SDF with ADDS, you can take advantage of the following ADDS capabilities:

- Defining prefixes for entities such as DMSII database elements, COBOL85 data description items, and fields on SDF and SDF Plus forms
- Defining a synonym, which means referring to an entity by another name

Related Information

The following table indicates the section in this manual or the title of the document in which you can find additional information about using SDF with COBOL85:

For Information About . . .	**Refer to . . .**
COBOL85 extensions for SDF	Section 1.
Defining prefixes and synonyms for entities	The *InfoExec Advanced Data Dictionary System (ADDS) Operations Guide*.
SDF concepts and programming considerations	The *Screen Design Facility (SDF) Operations and Programming Guide*.

Identifying the Dictionary

You identify the dictionary that contains the SDF form library you want to use by including a DICTIONARY clause in the SPECIAL-NAMES paragraph of the Environment Division of your COBOL85 program.

Note that a program can invoke only one dictionary. Thus, if a program accesses both a SIM database (from a dictionary) and SDF forms, both the database and the forms must be in the same dictionary.

Format

```
{ DICTIONARY IS literal-1 }
```

Explanation

Literal-1 is the value of the library attribute FUNCTIONNAME. This is the same function name that you specified on the SDF DICTIONARY SELECTION screen during the SDF session. Note that this function name is determined when you install SDF. You can equate the function name to a library code file by using the SL (Support Library) system command. For instructions on equating these file names, see the *SDF Operations and Programming Guide*.

Example

The following example specifies a dictionary whose function name is SCREENDESIGN:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
   DICTIONARY IS "SCREENDESIGN".
```

Declaring the Form Record Library Invocation

You must use a special data description entry to identify which form record library you want to invoke and to specify certain characteristics of that form record library. You can place the data description entry in any section of the Data Division **except** the following:

- Program-Library Section
- DATA-BASE Section
- COMMUNICATIONS Section

Format

```
level-number-1  form-record-library-name-1
     FROM DICTIONARY
     [ ; VERSION IS literal-1 ]
     [ ; DIRECTORY IS literal-2 ]
     [ ; SAME RECORD AREA ]
     [ ; REDEFINES form-record-library-name-2 ] [;]
```

Explanation

level-number-1	This must be the level number 01.
form-record-library-name-1	This is the name of the form record library that contains the descriptions of each of the message types and transaction types associated with an SDF formlibrary. You cannot use the INVOKE clause to assign an alias to form-record-library-name-1.
VERSION IS literal-1	This clause specifies a numeric literal that identifies a version of the file. This clause is valid only if you are using ADDS.
DIRECTORY IS literal-2	This clause specifies a nonnumeric literal that identifies the directory of the data dictionary in which the file is stored.

SAME-RECORD-AREA	This clause invokes all form record descriptions in the form library as redefinitions of the first form record in the library. You can use this clause only with data description entries declared in the Working-Storage, Linkage, and Local Storage Sections of the Data Division. Using this clause in any other section results in a syntax error.
REDEFINES form-record-library-name-2	This clause redefines a form record library whose data description entry uses the SAME RECORD AREA clause. You can use the REDEFINES clause only with formlibrary invocations declared in the Working-Storage, Linkage, and Local Storage Sections of the Data Division. Using the REDEFINES clause in any other section results in a syntax error.
; (Semicolon)	The semicolon that follows the ending period can be used to control the behavior of compiler control records (CCRs) and the format of listings. This semicolon should always be separated from the ending period of the data invocation by at least one space. If a CCR immediately follows a data item invoked from the dictionary, the compiler option changes might occur before the compiler processes the information invoked from the dictionary. This situation can be avoided by using the semicolon after the ending period. The semicolon ensures that the compiler processes the invoked information before the option actually changes.

Details

The data item described by an SDF data description entry must be declared as COMMON, GLOBAL, or EXTERNAL in the following situations:

The SDF data item must be declared . . .	**When it is declared in a . . .**
COMMON	A subprogram that references a formlibrary declared in the host program during binding.
GLOBAL	A host program that has nested subprograms that must access the form library.
EXTERNAL	A data structure that is shared through interprogram communication at run time.

READ FORM Statement

The READ FORM statement is used in the Procedure Division to read the input on a form from a user terminal to a program.

Format

```
READ FORM file-name-1

      {form-name-1 [ FROM DEFAULT FORM ] }
USING {                                  }
      {formlibrary-name-1                }

[ INTO identifier-1 ]

[ ON ERROR imperative-statement-1 ]
```

Explanation

filename-1	This is the name of a remote file from which the data is to be read. The file must be opened as INPUT or I/O at the time this statement is executed. The storage area associated with file-name-1 and identifier-1 cannot be the same area.
form-name-1	This is the name of a specific form that you want to read.
FROM DEFAULT FORM	This phrase causes the fields on the form to be completed with default values before read operations and record validations are performed.
formlibrary-name-1	This name indicates the name of a form library that contains self-identifying forms, which are forms for which you have defined a Message Key field. For details about defining message keys on forms, refer to the *SDF Operations and Programming Guide*.

INTO identifier-1	You can use the INTO phrase only if you invoked the associated formlibrary in the file description entry associated with file-name-1 in the File Section. The storage area associated with identifier-1 and the record area associated with file-name-1 cannot be the same area. The INTO phrase moves the record being read from the record area into the area specified by identifier-1. Identifier-1 is a data item declared in the Working-Storage Section that is used to store the information received as a result of the READ FORM statement. The move occurs according to the rules for the MOVE statement. (Refer to "MOVE Statement" in Volume 1 of the *COBOL85 Programming Reference Manual* for these rules.) The sending area is considered to be a group item equal in size to the maximum record size for this file. Any subscripting or indexing associated with the identifier is evaluated after the record is read and immediately before the record is moved to the data item. The record is available in both the input record area and the data area associated with the identifier. The move does not occur if the READ FORM operation is unsuccessful.
imperative-statement-1	This statement is executed if an error condition occurs. For details about error conditions, refer to "Error Conditions" under the heading of "Details."

Details

When the READ FORM statement is executed, the system

1. Reads the specified form
2. Validates the record
3. Performs error screen handling
4. Passes the valid record (in the record storage area associated with file-name-1) or the detected error condition back to the program
5. Updates the value of the file status data item associated with filename-1

   The message code 82 is returned in the file status data item when either of the following errors occurs:

   - A read operation is executed for a form that is not present in the form library.
   - The compile-time version of the form does not equal the run-time version of the form.

Defining a File Status Data Item

You define a file status data item as a two-character, alphanumeric data item in the Data Division of your COBOL85 program. (For details about declaring a data item, refer to Volume 1 of the *COBOL85 Programming Reference Manual*.) You link this data item with a file by specifying its data name in the FILE STATUS clause in the Input-Output Section of the Environment Division. (For details about the FILE STATUS clause, refer to Section 3 in Volume 1 of the *COBOL85 Programming Reference Manual*.)

Avoiding Truncation of Trailing Characters

When a form is invoked in a section other than the File Section, the data is read into the file-name-1 record area and is then transferred to the forms record area. To avoid truncation of trailing characters in the message, make the record description entry for file-name-1 as large as the largest form to be used with the file.

Multiple Record Descriptions for a File

When the logical records of a file are described with more than one record description entry, the records automatically share the same storage area. This sharing is equivalent to an implicit redefinition of the area. The contents of any data item that is beyond the range of the current data record is undefined after the READ FORM statement is executed.

Error Conditions

When an error condition is recognized, the READ FORM statement is unsuccessful. A value is placed in the FILE STATUS data item (if you specified that data item for the file) and the following action occurs:

If the ON ERROR clause . . .	And . . .	Then . . .
Is specified		Control is transferred to imperative-statement-1. Any USE procedure specified for this file is not executed
Is **not** specified	No USE procedure is specified	The program is terminated.
Is **not** specified	A USE procedure is specified	The USE procedure is executed

For general information on error handling with remote files, refer to the *SDF Operations and Programming Guide*.

WRITE FORM Statement

The WRITE FORM statement is used to write a form and its data from the program to a user terminal.

Format

```
WRITE FORM { form-name-1                          }
           { record-name-1 USING form-name-2      }

[ FROM { identifier-1 } ]
       { DEFAULT FORM }

[ ON ERROR imperative-statement-1 ]
```

Explanation

form-name-1	This is the name of a form in the formlibrary.
record-name-1	This is the name of a logical record for a remote file. This file name can have a qualifier. (Qualification is discussed in Section 4 of Volume 1 of this manual.) Record-name-1 and identifier-1 cannot reference the same storage area.
USING form-name-2	The USING clause enables the writing of forms from a formlibrary declared in the Working Storage Section. The normal record area of the file is ignored, and the record area for form-name-2 is written.
FROM identifier-1 DEFAULT FORM	A WRITE FORM statement with the FROM phrase is equivalent to the statement "MOVE identifier-1 TO record-name" followed by the WRITE FORM statement without the FROM phrase. When the DEFAULT FORM clause is used, the form library inserts default values in the fields on the form.

Details

The execution of the WRITE FORM statement releases a logical record to a REMOTE file. The remote file must be opened as OUTPUT, IO, or EXTEND at the time this statement is executed.

The execution of a WRITE FORM statement does not affect the contents or accessibility of the record area of the remote file. In addition, the current record pointer is unaffected by the WRITE FORM statement.

Effect of SAME RECORD AREA Clause

If the file associated with record-name-1 has the SAME RECORD AREA clause in its data description entry, the logical record is also available as a record of other files referenced in that SAME RECORD AREA clause.

Error Conditions

When an error condition is recognized, the WRITE FORM statement is unsuccessful. A value is placed in the FILE STATUS data item (if you specified that data item for the file) and the following action occurs:

If the ON ERROR clause . . .	And . . .	Then . . .
Is specified		Control is transferred to imperative-statement-1. Any USE procedure specified for this file is not executed.
Is **not** specified	No USE procedure is specified	The program is terminated.
Is **not** specified	A USE procedure is specified	The USE procedure is executed.

You can use the FILE STATUS clause in the Input-Output Section of the Environment Division to enable a form error message code to be returned. The message code 82 is returned when either of the following errors occurs:

- A write operation is executed for a form that is not present in the formlibrary.
- The compile-time version of the form does not equal the run-time version of the form.

For details about the FILE STATUS clause, refer to the discussion of the Environment Division in Volume 1 of the COBOL85 manual.

FORM-KEY Function

The FORM-KEY function enables the compiler to access the unique internal binary form number of the specified form. The FORM-KEY function is required for using SDF with COMS. This function is used with the MOVE statement.

Format

```
FORM-KEY (form-name-1)
```

Example

```
MOVE FORM-KEY(SDFFORM) TO COMS-OUT-CONVERSATION.
```

This example shows how to use the FORM-KEY function syntax within a MOVE statement.

Details

When using SDF with COMS (which is required for users migrating from V Series COBOL74 to A Series COBOL85), you must use the FORM-KEY function to move the form key into the first word of the output conversation area before executing a SEND statement.

Programmatic Control Flags

Programmatic control flags are provided at both the form level and the field level to cause extra data items to be generated into the COBOL85 program record description.

The symbolic name for a programmatic control flag is either

- Form-name-flag suffix
- Field-name-flag suffix

Table 6–1 lists the default SDF suffixes. If you choose not to use the default SDF suffixes, you must specify unique suffixes for each form. The entire name for the programmatic control flag cannot contain more than 30 characters.

Flags are set by the formlibrary when a read operation is performed or by the program when a write operation is performed, depending upon the type of flag (see Table 6–1). Flags set by the program are set before the first write operation and retain those values throughout the program, unless you reset them.

Table 6–2 lists the COBOL85 picture representation of the programmatic control flags along with their valid values.

Table 6–1. Default SDF Suffixes for Programmatic Control Flags

Flag Name	Suffix	Form or Field Level	When Set
Cursor	-CURSOR	Field	Before a write operation
Data Only	-DATA	Both	Before a write operation
Flag groups	-FLAGS	Form	Before a write operation
Highlight	-HIGHLIGHT	Field	Before a write operation
Input/Output	-IOTYPE	Field	Before a write operation
No input	-NOINPUT	Field	By the formlibrary (checked by the program after a read operation)
Page	-PAGE	Form	Before a write operation
Specify	-SPECIFY	Both	By the formlibrary when a read operation occurs
Field suppress	-SUPPRESS	Field	Before a write operation

Table 6–2. COBOL85 Picture Representations and Values of Programmatic Control Flags

Flag Name	COBOL85 Picture	Valid Values
Cursor	PIC 9 (1) COMP	0 - No cursor positioning 1 - Cursor positioning
Data only	PIC 9(1) COMP	0 - No data only 1 - Data only
Field suppress	PIC 9(1) COMP	0 - Not suppressed 1 - Suppressed
Flag groups	Not applicable	Not applicable
Highlight	PIC 9(1) COMP	Fixed highlighting: 0 - Not specified >0 - Specified Variable highlighting 0 - None 1 - Bright 2 - Reverse 3 - Secure 4 - Underline 5 - Blink
Input/Output	PIC 9(1) COMP	1 - Input 2 - Input only 3 - Output 4 - Output transmittable
No input	PIC 9(1) COMP	0 - Data input 1 - No data input
Page	PIC 9(1) COMP	Terminal page 1 through 9
Specify	PIC 9(4) COMP	0 - Not specified >0 - Specified

Generating Flag Groups

You can create a flag group through SDF, which enables you to reset all generated flags. You provide a group name for all the flags in the form, and individual group names for each type of flag. The names of the flag groups must follow COBOL85 naming conventions and must be unique for each field or form.

The group name for the form has the following syntax:

```
<form name>-<group flag suffix>
```

The group name for each type of flag has the following syntax:

```
<form name>-<flag suffix>-<group flag suffix>
```

At the group level, you can use hexadecimal values for zero. You can set individual flags to hex 0 by using the figurative constant LOW-VALUES. LOW-VALUES used at the group level causes spaces to be moved rather than hex zeroes, because the destination is considered to be alphanumeric.

Resetting Control Flags to Zero

The following sample COBOL85 code assumes that FORM-1 has two fields, FIELD-1 and FIELD-2. If the form and each field had all the possible programmatic control flags set, the COBOL85 01 record would appear as shown.

```
01  FORM-1.
    02  FIELD-1                            PIC X(10).
    02  FIELD-2                            PIC 9(6) V9(2).
    02  FORM-1-FLAGS.
      03  FORM-1-PAGE-FLAGS.
        04  FORM-1-PAGE                    PIC 9 COMP.
      03  FORM-1-SPECIFY-FLAGS.
        04  FORM-1-SPECIFY                 PIC 9(4) COMP.
        04  FIELD-1-SPECIFY                PIC 9(4) COMP.
        04  FIELD-2-SPECIFY                PIC 9(4) COMP.
      03  FORM-1-IOTYPE-FLAGS.
        04  FIELD-1-IOTYPE                 PIC 9 COMP.
        04  FIELD-2-IOTYPE                 PIC 9 COMP.
      03  FORM-1-CURSOR-FLAGS.
        04  FIELD-1-CURSOR                 PIC 9 COMP.
        04  FIELD-2-CURSOR                 PIC 9 COMP.
      03  FORM-1-SUPPRESS-FLAGS.
        04  FIELD-1-SUPPRESS               PIC 9 COMP.
        04  FIELD-2-SUPPRESS               PIC 9 COMP.
      03  FORM-1-HIGHLIGHT-FLAGS.
        04  FIELD-1-HIGHLIGHT              PIC 9 COMP.
        04  FIELD-2-HIGHLIGHT              PIC 9 COMP.
      03  FORM-1-DATA-FLAGS.
        04  FORM-1-DATA                    PIC 9 COMP.
        04  FIELD-1-DATA                   PIC 9 COMP.
        04  FIELD-2-DATA                   PIC 9 COMP.
      03  FORM-1-NOINPUT-FLAGS.
        04  FIELD-1-NOINPUT                PIC 9 COMP.
        04  FIELD-2-NOINPUT                PIC 9 COMP.
```

To reset all the flags declared in the preceding data description entry to zero, you would use the following statement:

```
MOVE ALL @00@ TO FORM-1-FLAGS.
```

Using SDF with COMS

You can use SDF with COMS to take advantage of COMS direct windows, which give you enhanced routing capabilities for forms and also allow preprocessing and postprocessing of forms. When using SDF and COMS together, follow the instructions for using each product as documented in the appropriate user manuals, except as noted in the following paragraphs.

REDEFINES and SAME RECORD AREA Clauses

When using the COMS direct-window interface, you can use the REDEFINES and the SAME RECORD AREA clauses in the data description entry for a formlibrary in the Working-Storage Section. The following example illustrates the use of the SAME RECORD AREA clause in a COBOL85 program that uses both SDF and COMS:

```
001800 WORKING-STORAGE SECTION.
002000 01  COMS-NAME                PIC X(072).
002500 01  COMS-MESSAGE-AREA.
002600     02  COMS-MESSAGE          PIC X(1920).
002620 01  SDFFORMLIBRARY FROM DICTIONARY;
002625          SMAE RECORD AREA.
```

RECEIVE Statement

In the main processing loop, the RECEIVE statement uses the SDF form as the message area, as shown in the following example:

```
006300 RECEIVE COMS-IN MESSAGE INTO SDFFORM.
```

In this example, COMS-IN is the name of the COMS header. SDFFORM is the name of the form in the formlibrary named SDFFORMLIBRARY.

FORM-KEY Function

The FORM-KEY function moves the form key into the first word of the output conversation area. The FORM-KEY function syntax must precede a SEND statement. Sample code that uses this function is as follows:

```
006355
006360  MOVE 1                     TO COMS-OUT-COUNT.
006400  MOVE COMS-IN-STATION       TO COMS-OUT-DESTINATION.
006700  MOVE 60                    TO COMS-OUT-TEXT-LENGTH.
006720  MOVE SDF-AGENDA-DESIGNATOR TO COMS-OUT-AGENDA.
006740  MOVE FORM-KEY(SDFFORM)     TO COMS-OUT-CONVERSATION.
006800  SEND COMS-OUT FROM SDFFORM.
007100 END-OF-JOB.
```

Transmitting a Default Form

To transmit a form with default values, you can perform one of the following actions:

Move . . .	To . . .
Spaces	Display items.
Zeroes	Numeric items.

Sample COBOL85 Programs That Use SDF

The following are sample programs that illustrate the different uses of the SDF program interface within COBOL85. Comment lines explain the various sections of the program. For information about how to handle remote file errors in an application program, refer to the *SDF Operations and Programming Guide*.

Code for Remote File Interface and READ Statement

The following sample program uses a remote file and specific forms. The program contains a READ FORM statement for a default form. The form record library was created with SDF.

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*************************************************************
* The following lines specify the dictionary that stores  *
* the form record library.                                  *
*************************************************************
 SPECIAL-NAMES.
   DICTIONARY IS "SCREENDESIGN".
*************************************************************
* The following lines specify the remote file.            *
*************************************************************
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT REMFILE
       ASSIGN TO REMOTE.
*************************************************************
* The following lines invoke the form library in the File *
* Section and associate it with the file. Invoking the    *
* form library also causes the proper maximum record size *
* (MAXRECSIZE) to be designated for the file.             *
*************************************************************
```

```
 DATA DIVISION.
 FILE SECTION.
 FD REMFILE.
 01  SAMPLELIB   FROM DICTIONARY.
************************************************************
* The following lines include program record descriptions *
* for all forms in the form library that are              *
* automatically invoked and copied into the program       *
* during compilation (see the following dictionary lines  *
* identified with the D flag).                            *
************************************************************
*-DICTIONARY                                            D
*-DICTIONARY FORMLIST <SAMPLELIB>.                      D
 01  SAMPLEFORM1.                                       D
    02  ACTION                     PIC X(10).          D
    02  ACCOUNT-NO                 PIC (9).            D
    02  NAME                       PIC X(15).          D
    02  STREET                     PIC X(25).          D
    02  CITY                       PIC X(15).          D
    02  STATE                      PIC X(2).           D
    02  ZIPCODE                    PIC 9(9).           D
 PROCEDURE DIVISION.
 MAIN-PARA.
************************************************************
* The following line opens the remote file in I/O mode.   *
************************************************************
   OPEN I-O REMFILE.
************************************************************
* The following READ statement writes the form named      *
* SAMPLEFORM1 with its default values and then reads the  *
* form. Note that the WRITE FORM statement is not         *
* required to send a form with default values to a        *
* station.                                                *
************************************************************
   READ FORM REMFILE USING SAMPLEFORM1 FROM DEFAULT FORM.
 STOP RUN.
```

Remote File Interface and READ and WRITE Statements

The following sample code shows how the WRITE and READ statements could be used in the Procedure Division of the program shown in the preceding subsection.

```
 PROCEDURE DIVISION.
 MAIN-PARA.
************************************************************
* The following line opens the remote file in I/O mode.   *
************************************************************
   OPEN I-O REMFILE.
```

```
*********************************************************
* The following WRITE FORM statement writes the form     *
* named SAMPLEFORM1 to the station.                      *
*********************************************************
 WRITE FORM SAMPLEFORM1.
*********************************************************
* The following READ FORM statement relays the data that  *
* the user entered on SAMPLEFORM1 to the program when the *
* user transmits the form.                               *
*********************************************************
   READ FORM REMFILE USING SAMPLEFORM1.
 STOP RUN.
```

Remote File Interface and Programmatic Controls

The following sample code is a COBOL85 program that uses SDF, a remote file, specific forms, and programmatic controls. As you read this example, remember the following information about programming flags:

- If your forms use either SPECIFY or NO INPUT flags, design your program to verify that these flags are set after a READ operation before processing data from the fields on the forms. If you use both SPECIFY and NO INPUT flags, direct your program to check the SPECIFY flag first.
- If the value in any SPECIFY flag field is greater than zero, the values in the data fields of the form are unchanged from the previous operation.

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*********************************************************
* The following lines specify the dictionary that stores  *
* the form record library.                               *
*********************************************************
 SPECIAL-NAMES.
   DICTIONARY IS "SCREENDESIGN";
   ALPHABET XXX IS EBCDIC.
*********************************************************
* The following lines declare the remote and disk files.  *
*********************************************************
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT MITERM
          ASSIGN TO REMOTE.
     SELECT DISK-FILE
          ASSIGN TO DISK.
 DATA DIVISION.
 FILE SECTION.
```

```
************************************************************
* The following line specifies the file that will be      *
* associated with the form library specified in this      *
* section.                                                *
************************************************************
 FD  MITERM
************************************************************
* The following lines specify file attributes to ensure   *
* the correct record size for write operations in which   *
* the form record can be longer than 80 characters.       *
************************************************************
 BLOCK CONTAINS 2200 CHARACTERS
 RECORD CONTAINS 2200 CHARACTERS
 VALUE OF MAXRECSIZE IS 2200
 VALUE OF FILETYPE IS 3
 VALUE OF MYUSE IS IO
 CODE-SET IS XXX.
************************************************************
* The following line invokes the form library.            *
************************************************************
 01  VOTERLIB FROM DICTIONARY.
************************************************************
* The following lines include record descriptions for     *
* all of the forms in the form library that are           *
* automatically invoked and copied into the program       *
* during compilation (see the following dictionary data   *
* lines identified by the D flag).                        *
************************************************************
*-DICTIONARY                                              D
*-DICTIONARY FORMLIST <VOTERLIB>.                         D
 01  VRFORM.                                              D
     02  PRECINCT                    PIC 9(4).            D
     02  LOCATION                    PIC X(28).           D
     02  VRNAME                      PIC X(54).           D
     02  ADDRESS                     PIC X(54).           D
     02  CITY                        PIC X(24).           D
     02  COUNTY                      PIC X(24).           D
     02  CONGRESSDIS                 PIC 9(4).            D
     02  REPRESDIS                   PIC 9(4).            D
     02  SENATEDIS                   PIC 9(4).            D
     02  COMMISSDIS                  PIC 9(4).            D
     02  VRDATE                      PIC 9(6).            D
     02  CLERK                       PIC X(29).           D
     02  VRNAME-CURSOR               PIC 9(1)  COMP.      D
     02  VRNAME-HIGHLIGHT            PIC 9(1)  COMP.      D
 FD  DISK-FILE.
 01  DATA-RECORD                     PIC X(300).
 WORKING-STORAGE SECTION.
 PROCEDURE DIVISION.
 MAIN-PARA.
```

```
************************************************************
* The following lines open the remote file and the disk  *
* file.                                                  *
************************************************************
 OPEN I-O MITERM.
 OPEN OUTPUT DISK-FILE.
************************************************************
* The following lines move values to the fields of the    *
* form so that the form can be written with those values. *
************************************************************
 MOVE SPACES TO VRFORM.
 MOVE ZEROS TO VRDATE.
************************************************************
* The following lines prevent highlighting from being     *
* incorrectly set.                                       *
************************************************************
 MOVE 0 TO VRNAME-CURSOR.
 MOVE 0 TO VRNAME-HIGHLIGHT.
************************************************************
* The following lines create a loop that enters and       *
* stores data in a disk file.                            *
************************************************************
 PERFORM DATA-ENTRY UNTIL CLERK = "DONE".
 END-MAIN-PARA.
 STOP RUN.
************************************************************
* The following lines signify the beginning of the data   *
* entry loop. These lines move values to the indicated    *
* fields.                                                *
************************************************************
 DATA-ENTRY.
 MOVE SPACES TO VRNAME.
 MOVE SPACES TO ADDRESS.
 MOVE SPACES TO CITY.
 MOVE SPACES TO COUNTY.
************************************************************
* The following WRITE FORM statement writes the form      *
* name. The READ statement reads the form from the        *
* terminal. The MOVE and WRITE statements store the form  *
* in a record file.                                      *
************************************************************
 WRITE FORM VRFORM
     ON ERROR STOP RUN.
 READ FORM MITERM USING VRFORM
     ON ERROR STOP RUN.
 MOVE VRFORM TO DATA-RECORD.
 WRITE DATA-RECORD.
```

```
*************************************************************
* The following lines use programmatic control to         *
* position the cursor (VRNAME-CURSOR) and place the       *
* cursor in the VRNAME field when the form is displayed.  *
* Note that station users can tab back to the first       *
* field to enter data if they desire.                     *
*************************************************************
 MOVE 1 TO VRNAME-CURSOR.
 END-DATA-ENTRY.
```

Message Keys and Independent Record Area

The following code shows the use of message keys and an independent record area in a COBOL85 program that uses SDF. In this example, the SDF formlibrary is named SAMPLELIB, and the forms are named SAMPLEFORM1 and SAMPLEFORM2. The Action field is defined as the Message Key field.

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*************************************************************
* The following lines specify the dictionary that stores  *
* the form record library.                                *
*************************************************************
 SPECIAL-NAMES.
   DICTIONARY IS "SCREENDESIGN";
   ALPHABET XXX IS EBCDIC.
*************************************************************
* The following lines declare the remote file.            *
*************************************************************
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT REMFILE
          ASSIGN TO REMOTE.
 DATA DIVISION.
 FILE SECTION.
*************************************************************
* The following line specifies the file that will be      *
* associated with the form library.                       *
*************************************************************
 FD  REMFILE
*************************************************************
* The following lines specify file attributes to ensure   *
* the correct record size for write operations in which   *
* the form record can be longer than 80 characters.       *
*************************************************************
 BLOCK CONTAINS 2500 CHARACTERS
 RECORD CONTAINS 2500 CHARACTERS
 VALUE OF FILETYPE IS 3
 VALUE OF MYUSE IS IO
```

```
 CODE-SET IS XXX.
************************************************************
* The following line invokes the form library.             *
************************************************************
 01  SAMPLELIB FROM DICTIONARY.
************************************************************
* The following lines include record descriptions for      *
* all of the forms in the form library that are            *
* automatically invoked and copied into the program        *
* during compilation (see the following dictionary data    *
* lines identified by the D flag).                         *
************************************************************
*-DICTIONARY                                               D
*-DICTIONARY FORMLIST <SAMPLELIB>.                         D
 01  SAMPLEFORM1.                                          D
     02  ACTION                      PIC X(11).           D
     02  ACCOUNT-NO                  PIC 9(9).            D
     02  NAME                        PIC X(15).           D
     02  STREET                      PIC X(25).           D
     02  CITY                        PIC X(15).           D
     02  STATE                       PIC X(2).            D
     02  ZIPCODE                     PIC 9(9).            D
 01  SAMPLEFORM2.                                          D
     02  ACTION                      PIC X(11).           D
     02  ACCOUNT-BALANCE             PIC 9(9).            D
     02  PAYMENT-DUE-DATE            PIC X(6).            D
     02  DATE-LAST-PAYMENT           PIC X(6).            D
     02  FINANCE-CHARGE              PIC 9(5).            D
************************************************************
* The following lines specify the SDF message key. The     *
* size of the SDF message key must be the same size as     *
* the input message key. The total message area size       *
* must be large enough to hold any SDF input message.      *
************************************************************
 WORKING-STORAGE SECTION.
 01  SDF-MESSAGE-AREA.
     02  SDF-MESSAGE-KEY             PIC X(011).
     02  SDF-MESSAGE                 PIC X(2500).
************************************************************
* Program the main processing loop so that the RECEIVE     *
* uses a working storage area, SDF-MESSAGE-AREA, for the   *
* the input message. Messages or errors might arrive for   *
* program from your formlibrary.                           *
************************************************************
 PROCEDURE DIVISION.
 MAIN-PARA.
************************************************************
* Open remote file I/O.                                    *
************************************************************
 OPEN I-O REMFILE.
```

```
*************************************************************
* Move values to the fields of the form so that a write   *
* can be done to display the form with those values.      *
*************************************************************
 MOVE SPACES TO NAME.
 MOVE SPACES TO STREET.
 MOVE SPACES TO CITY.
 MOVE SPACES TO STATE.
 MOVE ZEROS TO ACCOUNT-NO.
 MOVE ZEROS TO ZIPCODE.
*************************************************************
* These are WRITE FORM and READ FORM statements that      *
* explicitly state the form name. Use the WRITE FORM      *
* statement to write the form from the formlibrary to the *
* terminal with the changed values. The READ FORM         *
* statement uses the form name. The form name can be used *
 * even if message keys have been defined for the forms in*
 * the formlibrary.                                       *
*************************************************************
 WRITE FORM SAMPLEFORM1
 ON ERROR STOP RUN.
 READ FORM REMFILE USING SAMPLEFORM1
 ON ERROR STOP RUN.
*************************************************************
* The forms in the formlibrary contain message keys;      *
* therefore, they are self-identifying forms. Note the    *
* syntax of the READ FORM statement.                      *
*                                                         *
* Use the WRITE FORM statement to write the specific form *
* when using message keys to identify input forms. The    *
* READ FORM statement uses the formlibrary name and a     *
* separate working storage area instead of the form name. *
* The program examines the field SDF-MESSAGE-KEY that     *
* contains the message key to identify the form.          *
*************************************************************
 READ FORM REMFILE USING SAMPLELIB
 INTO SDF-MESSAGE-AREA
 ON ERROR STOP RUN.
```

```
************************************************************
* The value of the Message Key field SDF-MESSAGE-KEY       *
* determines the conditional function to be performed.     *
*                                                          *
* Once the program determines that a valid input has been  *
* received from a form in your formlibrary, you can        *
* process the data received. The example uses the STOP     *
* RUN statement to handle any error.                       *
*                                                          *
* After the SDF errors have been processed, you can        *
* determine which form was used for input and move the     *
* data in the SDF-MESSAGE-AREA to the SDF form record for  *
* further processing. In the following example,            *
* SDF-MESSAGE-KEY is checked for the form message key and  *
* then the SDF form SAMPLEFORM1 or SAMPLEFORM2 is          *
* processed.                                               *
*                                                          *
*Move the data in SDF-MESSAGE-AREA to the SDF form         *
*record before further processing.                        *
************************************************************
 IF SDF-MESSAGE-KEY = "ADD-ITEM"
 MOVE SDF-MESSAGE-AREA TO SAMPLEFORM1
 PERFORM ADD-ITEM
 ELSE
 IF SDF-MESSAGE-KEY = "MODIFY-ITEM"
 MOVE SDF-MESSAGE-AREA TO SAMPLEFORM2
 PERFORM MODIFY-ITEM.
************************************************************
* First process all Specify programmatic control data for  *
* form before checking for No input programmatic control   *
* data and processing other input data.                    *
************************************************************
 ADD-ITEM.
************************************************************
* Insert code to add an item.                              *
************************************************************
 MODIFY-ITEM.
************************************************************
* Insert code to modify an existing item.                  *
************************************************************
 STOP RUN.
```

Section 7
TransIT Open/OLTP

What is Open/OLTP?

Traditional online transaction processing (OLTP) enables an application program to update databases, but leaves the responsibility of coordinating the updates and recovering from errors to the application program. With the advent of Open/OLTP, the application program is relieved of these duties. Open/OLTP ensures that transactions are committed after services have been successfully completed or rolled back to the previous version if the services have not been successfully completed. 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.

Open/OLTP also implements a *client/server* model. Clients invoke database services, but do not directly update the databases. Servers provide multiple services including a service to update a DMSII database. The client/server model applies only to COMS online programs, because the implementation of services is provided by COMS.

Accessing Open/OLTP

You can access Open/OLTP from COBOL85 through the TX and XATMI interfaces. These interfaces have entry points that you access by using the COBOL85 CALL statement. To assist you in accessing these interfaces, you are provided with a COBOL85 include file on the release media. For instructions on using the include file, refer to the *Open/OLTP Programming Guide*.

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.
```

For More Information

For complete information about creating applications that use Open/OLTP, refer to the *TransIT Open/OLTP Programming Guide*.

Section 3 of this manual describes the following COMS service functions, which are related to the XATMI interface:

- GET_BUFFER_DESIGNATOR
- GET_DESIGNATOR_USING_DESIGNATOR
- GET_ERRORTEXT_USING_NUMBER
- GET_INTEGER_USING_DESIGNATOR

Appendix A
Reserved Words

The following is a list of reserved words. It includes all reserved words from the complete American National Standard (ANSI-85), and additions used with Unisys extensions. Reserved words that are new to the ANSI-85 standard are marked with a double asterisk (**). Reserved words that are in the ANSI-85 standard but were also in the ANSI-74 standard are not marked.

Unisys does not use all of these words at the present time, but they are all in the reserved word list for the A Series COBOL ANSI-85 compiler. The use of any of these words as a user-defined word causes an error.

A

ABORT-TRANSACTION
ACCEPT
ACCESS
ACTUAL
ADD
ADVANCING
AFTER
ALL
ALLOW
ALPHABET**
ALPHABETIC
ALPHABETIC-LOWER
ALPHABETIC-UPPER**
ALPHANUMERIC**
ALPHANUMERIC-EDITED**
ALSO
ALTER
ALTERNATE
AND
ANY**
ARE
AREA
AREAS
AS
ASCENDING
ASCII
ASSIGN
AT
ATTACH
ATTRIBUTE
AUDIT
AUTHOR
AVAILABLE

B

BACKUP
BEFORE
BEGIN-TRANSACTION
BEGINNING
BINARY**
BLANK
BLOCK
BOTTOM
BY

C

CALL
CANCEL
CARDS
CASSETTE
CAUSE
CD
CF
CH
CHANGE
CODE-SET
COLLATING
COLUMN
COLUMNS
COMMA
COMMON**
COMMUNICATION
COMP
COMP-5
CONTROL
CONTROL-POINT
CONTROLS
CONVERSATION
CONVERTING**
COPY
CORR
CORRESPONDING
COUNT

C (cont.)

CHANNEL
CHARACTER
CHARACTERS
CLASS**
CLOCK-UNITS
CLOSE
COBOL
CODE
COMPUTATIONAL
COMPUTATIONAL-5
COMPUTE
CONFIGURATION
CONTAINS
CONTENT**
CONTINUE**
CRCR-INPUT
CRCR-OUTPUT
CREATE
CRUNCH
CURRENCY
CURRENT
CYLINDER

D

DATA
DATA-BASE
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DAY-OF-WEEK**
DB
DE
DEADLOCK
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGNG
DECIMAL-POINT
DECLARATIVES
DEFAULT
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETACH
DETAIL
DICTIONARY
DISABLE
DISALLOW
DISK
DISPACK
DISMISS
DISPLAY
DIVIDE
DIVISION
DMCANCEL
DMCLOSE
DMDELETE
DMERROR
DMOPEN
DMREMOVE
DMSAVE
DMSET
DMSTATUS
DMSTRUCTURE
DMTERMINATE
DOUBLE
DOWN
DUMP
DUPLICATES
DYNAMIC

E

EBCDIC
EGI
ELSE
EMI
ENABLE
END
END-ABORT-TRANSACTION
END-ADD**
END-ASSIGN
END-BEGIN-TRANSACTION
END-CALL**
END-CANCEL
END-CLOSE
END-COMPUTE**
END-CREATE
END-DELETE**
END-DIVIDE**
END-END-TRANSACTION
END-EVALUATE**
END-GENEREATE
END-IF**
END-INSERT
END-LOCK
END-MODIFY
END-MULTIPY**
END-OF-PAGE
END-OPEN
END-PERFORM**
END-READ**
END-RECEIVE**
END-RECREATE
END-REMOVE
END-RETURN**
END-REWRITE**
END-SAVE
END-SEARCH**
END-SECURE
END-SET
END-STRING**
END-SUBTRACT**
END-TRANSACTION
END-UNSTRING**
END-WRITE**
ENDING
ENTER
ENTRY
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVALUATE**
EVENT
EVERY
EXCEPTION
EXIT
EXTEND

E (cont.)

END-FIND
END-FREE
END-START**
END-STORE
EXTERNAL**
EXTERNAL-FORMAT

F

FALSE**
FD
FIELD
FILE
FILE-CONTROL
FILLER
FINAL
FIND
FIRST
FOOTING
FOR
FORM
FORM-KEY
FORMS
FREE
FROM
FUNCTION

G

GCR
GENERATE
GIVING
GLOBAL**
GO
GREATER
GROUP

H

HEADING
HIGH-VALUE
HIGH-VALUES

I

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE**
INITIATE
INPUT
INPUT-OUTPUT
INQUIRY
INSERT
INSPECT
INSTALLATION
INTEGER
INTERROGATE
INTERRUPT
INTO
INVALID
INVOKE
IS

J

JUST
JUSTIFIED

K

KANJI
KEY

L

LABEL
LAST
LB
LD
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCAL
LOCAL-STORAGE
LOCK
LOCKED
LOW-VALUE
LOW-VALUES
LOWER-BOUND
LOWER-BOUNDS

M

MEMORY	MODE	MOVE
MERGE	MODIFY	MULTIPLE
MESSAGE	MODULE	MULTIPLY
MID-TRANSACTION	MODULES	

N

NATIONAL	NO	NULL
NATIONAL-EDITED	NO-AUDIT	NUMBER
NATIVE	NONE	NUMERIC
NEGATIVE	NOT	NUMERIC-EDITED**
NEXT		

O

OBJECT-COMPUTER	OFFSET	ORDER**
OCCURS	OMITTED	ORGANIZATION
ODT	ON	OTHER**
ODT-INPUT-PRESENT	OPEN	OUTPUT
OF	OPTIONAL	OVERFLOW
OFF	OR	OWN
OFFER		

P

PACKED-DECIMAL**	PICTURE	PROCEDURE
PADDING**	PLUS	PROCEDURES
PAGE	POINT	PROCEED
PAGE-COUNTER	POINTER	PROCESS
PAPERTAPE	PORT	PROGRAM
PERFORM	POSITION	PROGRAM-ID
PF	POSITIVE	PROGRAM-LIBRARY
PH	PRINTER	PUNCH
PHASE-ENCODED	PRINTING	PURGE**
PIC	PRIOR	

Q

QUEUE	QUOTE	QUOTES

R

RANDOM	REFERENCE**	REPORTS
RD	REFERENCES	RERUN
READ	RELATIVE	RESERVE
READ-OK	RELEASE	RESET
READER	REMAINDER	RE-START
RECEIVE	REMOTE	RETURN
RECEIVED	REMOVAL	REVERSED
RECORD	REMOVE	REWIND
RECORDS	RENAMES	REWRITE
RECREATE	REPLACE**	RF

R (cont.)

RH
REDEFINES
REAL
REEL
REF
REPLACING
REPORT
REPORTING
RIGHT
ROUNDED
RUN

S

SAME
SAVE
SD
SEARCH
SECTION
SECURE
SECURITY
SEEK
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SINGLE
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STACK
STANDARD
STANDARD-1
STANDARD-2**
START
STATUS
STOP
STOQ-INPUT
STOQ-OUTPUT
STORE
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8
SYMBOLIC
SYNC
SYNCHRONIZED
SYSTEM
SYSTEMERROR

T

TABLE
TAG-KEY
TAG-SEARCH
TALLYING
TAPE
TAPES
TASK
TERMINAL
TERMINATE
TEST**
TEXT
THAN
THEN**
THROUGH
THRU
TIME
TIMER
TIMES
TO
TODAYS-DATE
TODAYS-NAME
TOP
TRACE-OFF
TRACE-ON
TRAILING
TRANSACTION
TRANSCEIVE
TRUE**
TYPE

U

UNIT
UNLOCK
UNSTRING
UNTIL
UP
UPDATE
UPON
USAGE
USE
USING

V

VALUE
VALUES
VARYING
VIA

W

WAIT	WITH	WRITE
WHEN	WORDS	WRITE-OK
WHERE	WORKING-STORAGE	

Z

ZERO	ZEROS	ZEROS
ZEROES		

Special Characters

+	-	*
/	**	>
<	=	>=
<=		

Appendix B
User-Defined Words

A user-defined word is a COBOL85 word that you must supply to satisfy the format of a clause or statement. Each character of a user-defined word is selected from the set of characters A through Z, a through z, 0 (zero) through 9, and the hyphen (-). The hyphen cannot appear as the first or last character of a word. The words that you can define are shown in the following list, for reference. Detailed information about user-defined words is provided in Volume 1.

alphabet-name
class-name
COMS-header-name
condition-name
data-name
family-name
file-name
formlibrary-name
form-name
group-list-name
index-name
level-number
library-name
mnemonic-name
paragraph-name
program-name
record-name
report-name
routine-name
section-name
segment-number
symbolic-character
system-name
text-name

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X