UNISYS

A Series Communications Management System (COMS)

Programming Guide

Release 3.9.0

September 1991

Priced Item

U S America
8600 0650–000

UNISYS

A Series Communications Management System (COMS)

Programming Guide

Copyright © 1991 Unisys Corporation
All rights reserved.
Unisys is a registered trademark of Unisys Corporation.

Release 3.9.0

September 1991

U S America

Priced Item

8600 0650–000

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 THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or 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, indirect, 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.

Correspondence regarding this publication may be forwarded using the Product Information card at the back of the manual, or may be addressed directly to Unisys, Product Information, 25725 Jeronimo Road, Mission Viejo, CA 92691.

Page Status

Page	Issue
iii	–000
iv	Blank
v through xv	–000
xvi	Blank
xvii	–000
xviii	Blank
1–1 through 1–5	–000
1–6	Blank
2–1 through 2–5	–000
2–6	Blank
3–1 through 3–34	–000
4–1 through 4–16	–000
5–1 through 5–17	–000
5–18	Blank
6–1 through 6–20	–000
7–1 through 7–17	–000
7–18	Blank
8–1 through 8–5	–000
8–6	Blank
9–1 through 9–4	–000
A–1 through A–16	–000
B–1 through B–6	–000
C–1 through C–11	–000
C–12	Blank
D–1 through D–15	–000
D–16	Blank
E–1 through E–7	–000
E–8	Blank
F–1 through F–32	–000
Glossary–1 through 16	–000
Bibliography–1 through 2	–000
Index–1 through 10	–000

Unisys uses an 11-digit document numbering system. The suffix of the document number (1234 5678–*xyz*) indicates the document level. The first digit of the suffix (*x*) designates a revision level; the second digit (*y*) designates an update level. For example, the first release of a document has a suffix of –000. A suffix of –130 designates the third update to revision 1. The third digit (*z*) is used to indicate an errata for a particular level and is not reflected in the page status summary.

About This Guide

Purpose

This document is a guide for programming using the Communications Management System (COMS). The purpose of the guide is the following:

- To explain the various COMS features available to the programmer
- To outline how to program using the features available in COMS

The COMS product is a member of the InterPro (Interactive Productivity) Series of products designed for use with the Unisys A Series systems.

Scope

In this guide you can find information on programming for direct-window programs and remote-file programs. You can also find information on service functions, processing items, interactive and batch recovery, and security.

This guide does not include information on migrating from one release to another. For information on compatibility issues across releases, refer to the applicable release in the *A Series Mark 3.9 Software Release Capabilities Overview.*

Audience

This guide is intended for experienced applications programmers.

Prerequisites

Programmers should be proficient in the programming language they are using to write COMS application programs. They should also have a knowledge of data communications subsystems. If the application program is updating a Data Management System II (DMSII) database or a Semantic Information Manager (SIM) database, the programmer should have knowledge of DMSII or SIM database processing.

SIM is a member of the InfoExec™ family of products. The programmer should also be aware of the content of the COMS configuration file.

InfoExec is a trademark of Unisys Corporation.

How to Use This Guide

You should start by reading the overview to familiarize yourself with the features available through COMS. Separate sections of the guide are devoted to communicating with COMS through direct windows or remote-file windows. Each feature of COMS is presented in a separate chapter. The appendixes provide sample programs and quick reference to the meanings of messages your program receives. A glossary defines terminology related to COMS, and acronyms that appear in this guide are spelled out and defined in the glossary.

Manuals relevant to this product are given their full titles in the "Related Product Information" portion of this preface. In text, manuals are first referenced by their full titles and subsequent references use a shortened version of their titles. Unless otherwise specified, manuals referred to in the text are for A Series machines.

Organization

This guide is organized as follows:

Section 1. Introduction to COMS

This section introduces the COMS concepts and features, as they relate to programming.

Section 2. Creating Your COMS Application

This section outlines decisions that are important to a COMS programmer in developing applications through the use of the windows available in COMS.

Section 3. Communicating with COMS through Direct Windows

This section discusses how to use direct-window programs to receive and send messages involving COMS. Descriptions of the input and output header fields are provided, along with information on message-routing techniques.

Section 4. Accessing Service Functions

This section describes the service functions or entry points that COMS provides to allow you to obtain information on all configuration file entities.

Section 5. Processing Items

This section provides instructions for creating your own message control system (MCS) features in the form of separate modules called processing items that reside in processing-item libraries.

Section 6. Interactive Recovery

This section describes a method for writing programs that update databases and the behavior of those programs when reprocessing transactions after processing has been interrupted.

Section 7. Batch Recovery

This section describes a method for writing batch-oriented programs that update databases and the behavior of those programs when reprocessing transactions after processing has been interrupted.

Section 8. Security

This section provides information on the security-checking routines that you can write into application programs and processing items to augment COMS security or to function independently.

Section 9. Communicating with COMS through Remote-File Windows

This section describes the various types of remote-file windows and how they are used.

Appendix A. Tables of Values and Mnemonics

This appendix provides quick-reference access to the meanings of message values returned to your program and lists associated mnemonics.

Appendix B. COMS Header Layout

This appendix shows the layout of the COMS Header fields and attributes.

Appendix C. Sample COBOL74 Programs

This appendix provides sample programs that illustrate the use of various COMS features.

Appendix D. Sample Processing Items

This appendix contains two sample processing items and the set of global declarations that these processing items require.

Appendix E. Sample COBOL74-Processing Item Interface

This appendix contains a sample COBOL74 program that can be used to write processing items by interfacing with an ALGOL shell.

Appendix F. Service Functions for Previous Releases

This appendix presents service functions to be used with previous releases of COMS.

In addition, a glossary, a bibliography, and an index appear at the end of this guide.

Related Product Information

The information in this guide is supplemented by the following documents, which pertain to COMS:

***A Series Communications Management System (COMS) Capabilities Manual* (form 8600 0627)**

This manual introduces COMS, discusses the flexibility and efficiency of the COMS system, describes the COMS architecture, and discusses specific features available to the COMS user. This manual is written for upper management, the site manager, and the programming staff.

***A Series Communications Management System (COMS) Configuration Guide* (form 8600 0312)**

This guide provides an overview of the basic concepts and functions of COMS. It includes instructions for creating a working COMS configuration and information on how to monitor and fine-tune COMS system performance. This guide is written for installation analysts, systems analysts, programmers, administrators, and performance analysts.

***A Series Communications Management System (COMS) Migration Guide* (form 8600 1567)**

This guide explains how to migrate from existing message control systems (MCSs) to the current release of COMS. This guide is written for system administrators and system programmers who are responsible for the migration of their site to COMS.

***A Series Communications Management System (COMS) Operations Guide* (form 8600 0833)**

This guide explains how to perform terminal-based COMS functional tasks and serves as a reference to COMS commands. Syntax diagrams of COMS commands are provided with explanations and examples of how the commands can be used. This guide is written for terminal operators and computer operators.

Contents

Section 4. Accessing Service Functions

Section 5. Processing Items

Section 6. Interactive Recovery

Section 7. Batch Recovery

Section 8. Security

Section 9. Communicating with COMS through Remote-File Windows

Appendix A. Tables of Values and Mnemonics

Appendix B. COMS Header Layout

Appendix C. Sample COBOL74 Programs

Appendix D. Sample Processing Items

Appendix E. Sample COBOL74 Processing-Item Interface

Appendix F. Service Functions for Previous Releases

Tables

Section 1
Introduction to COMS

The Unisys Communications Management System (COMS) provides you with an extremely flexible and dynamic message control system (MCS) for the Unisys A Series systems.

COMS Versions

Two versions of COMS are available to you, and both use the COMS configuration file (which defines the characteristics of the COMS network) to provide the message control environment.

- COMS (Full-Featured) enables you to manipulate the configuration file by using the COMS Utility program. Additional features that you receive and can control directly are processing, routing, and some security features.

  The full-featured version of COMS also enables you to develop applications by using the transaction code (trancode) routing feature, the trancode security feature, the statistics feature, and the synchronized recovery feature for Data Management System II (DMSII) and Semantic Information Manager (SIM) databases.
- COMS (Kernel) creates a predefined configuration file, which cannot be manipulated. The COMS (Kernel) configuration file enables you to use the window feature with four windows: a Menu-Assisted Resource Control (MARC) window with eight dialogues, a Command and Edit (CANDE) window with two dialogues, a Generalized Message Control System (GEMCOS) window with one dialogue, and a printing window used to support the Remote Print System (ReprintS). Additionally, you can communicate with remote-file programs.

COMS Features

The main features that you should be familiar with before using COMS are

- Windows
- Message processing
- Message routing
- Security
- Statistics window
- Database recovery

Windows

COMS allows you to execute multiprogram environments from one station. Each program environment is referred to as a window. Windows do not allow you to see several program environments at the same time, but instead to move from one program environment to another while processing continues uninterrupted.

Thus, if payroll, accounts receivable, and inventory control applications are defined in separate windows, an entry clerk with the appropriate security clearances could run a payroll data entry application in one window, an accounts receivable application in another, and an inventory control application in a third window. This clerk could move from window to window without having to wait for processing in each window to stop.

Additionally, the window feature allows you to have up to eight dialogues in each defined window. A dialogue is a single access point into a given program environment (window). With multiple dialogues of a single window, the data entry clerk (who perhaps works for a service bureau and needs to concurrently process six customers using a single accounts receivable application) can change freely from one customer's dialogue to another without interrupting the processing in other copies of the accounts receivable application.

COMS provides three kinds of windows:

- Direct windows
- MCS windows
- Remote-file windows

A direct window allows you to route messages to programs defined to COMS while using all of COMS preprocessing and postprocessing capabilities. An MCS window provides access to a subordinate message control system, such as CANDE or GEMCOS. A remote-file window is a window established by COMS to allow programs to communicate interactively with data communication stations (remote files).

Declared remote files are defined in the configuration file, while dynamic remote files are opened by COMS to provide access to programs that are not defined in the configuration file.

For information about creating, modifying, and deleting windows in your COMS environment, refer to the *A Series Communications Management System (COMS) Configuration Guide.* For information on how to move from window to window, refer to the *A Series Communications Management System (COMS) Operations Guide.*

Message Processing

Direct windows provide considerable flexibility and processing power. COMS provides two ways of processing data found in direct window messages: by means of direct-window programs and by using processing items. You can arrange these methods in different sequences and use them to complement each other.

A direct-window program can be written in ALGOL, COBOL74, Pascal, or RPG, and can manipulate message data. All these languages, except Pascal, can be used to update DMSII and SIM databases. For more information about DMSII, see the *A Series DMSII Application Program Interfaces Programming Guide.* For more information about SIM, see the *A Series InfoExec Semantic Information Manager (SIM) Object Manipulation Language (OML) Programming Guide.*

A processing item is written in ALGOL and typically addresses a unique task that can be used by multiple applications and can be used many times within an application. For example, if your site requires that certain audits be done on all messages sent through the system, you can program that task in a processing item so that all applications developed in the COMS direct-window environment can use this processing item.

Processing items can be used to preprocess messages before they reach a direct-window program and postprocess messages after they leave a direct-window program. For example, you can use preprocessing to format a menu of an application, while you might use postprocessing to format and print a bill.

For information about COMS direct-window programming techniques and how to write processing items, refer to Section 5, "Processing Items," in this guide and to the *A Series ALGOL Programming Reference Manual, Volume 2: Product Interfaces.* For information on how to define processing items, see the *COMS Configuration Guide.*

For MCS and remote-file windows, COMS passes messages directly to and from the MCS or remote file.

Message Routing

COMS provides two methods of routing direct-window messages: the agenda method and the specific destination method.

An agenda is a mechanism for routing and processing messages. Processing items become a part of an agenda by associating them with an agenda in the configuration file. If you use an agenda to control message routing, you can send a message to that agenda for processing (or routing) before it reaches or after it leaves a direct-window program.

Trancodes can also be associated with an agenda. Trancodes are available only with the full-featured version of COMS. A trancode is a message identifier that can be used by COMS or by user applications to differentiate one message type from another. Thus, if INQ is the trancode for inquiry in a given application window, any message given that trancode would be routed to the agenda associated with that trancode. This method of routing is called trancode routing or transaction-based routing (TBR). One of the advantages of using trancodes for routing is that trancodes enable a direct window to have agendas assigned to it for various processing and routing requirements.

The specific destination method is used by direct-window programs that directly identify output message destinations. This provides you with additional flexibility by allowing you to change the agenda destination at run time to direct the message to a different destination.

For information about defining agendas and trancodes, refer to the *COMS Configuration Guide.*

For information about applying an agenda to a message or specifying a destination in a direct-window program, refer to Section 3, "Communicating with COMS through Direct Windows."

For MCS and remote-file windows, COMS passes messages directly to and from the MCS or remote file.

Security

COMS allows you to define security measures for direct windows through the COMS configuration file, through special processing items, and through direct-window programs.

COMS directly controls access to various parts of the network (for example, to MCS windows and declared remote-file windows) through the use of authorized usercodes and authorized stations. These options are discussed in the *COMS Configuration Guide.*

COMS also controls access through user-written processing items that perform security checking on usercodes and station names. See Section 5, "Processing Items," for more information.

In addition, you can write application programs that control security, based on information obtained from the COMS configuration file. For a complete discussion of programmatic control of the security of your COMS application in a direct-window program, refer to Section 8, "Security."

COMS controls access to MCS windows and declared remote-file windows through the use of authorized usercodes and authorized stations.

Statistics Window

Once you have installed your COMS direct-window environment, you can monitor system performance using the COMS Statistics window. COMS allows you to gather statistical summaries in the form of up-to-the-moment, online reports or in the form of much more extensive printed reports. See Section 4, "Accessing Service Functions," for more information.

Database Recovery

The database recovery feature (available for direct windows only, and only in the full-featured version) allows COMS to automatically resubmit transactions to your DMSII or SIM database after a transaction-state abort, system crash, or rollback. Recovery is performed for both interactive and batch updates. In addition, if you have implemented the protected input queue feature, recovery also causes messages in database program queues to be recovered and processed.

For information on writing a direct-window program that creates transaction trails, refer to Section 6, "Interactive Recovery." For information on protected input queues and on identifying the DMSII databases to COMS, refer to the *COMS Configuration Guide.* For information on controlling transaction trails for a DMSII database, refer to the DATABASE command description in the *COMS Operations Guide.* For information on SIM, see the *A Series InfoExec Semantic Information Management (SIM) Technical Overview.*

Section 2
Creating Your COMS Application

To create a COMS application in a direct window or convert an application to the COMS environment, you must

- Write any processing items that are needed.
- Write the direct-window programs.
- Modify the configuration file to define a direct window, and link to it the programs and processing items for which it is intended.

Suppose you wish to create a simple echo program to be run through a COMS direct window and to supply that program with one processing item that audits the messages and returns a duplicate of each message to the initiating station. To carry out this intention, you must do the following:

1. Decide on a processing-item library convention and write the processing item to audit the message before coming to the direct-window program. Refer to Section 5, “Processing Items,” for information on doing these tasks.
2. Write the direct-window program that receives and sends the messages. Refer to Section 3, “Communicating with COMS through Direct Windows,” for information about writing direct-window programs.
3. Use the COMS Utility to modify the configuration file as follows:
   a. Define the processing-item library.
   b. Define the direct-window program.
   c. Name and define the characteristics of the direct window.
   d. Define the processing item and include that processing in a processing-item list.
   e. Define the agenda and include the window name, processing-item list name, and the direct-window program.

Running an Application

When you submit the command *ON <your window name>*, COMS will start your program. When the program executes a RECEIVE statement and COMS has a message for it, COMS constructs the input header, applies the specified agenda, and then exits.

When your program sends a message, COMS applies the specified agenda (if any), routes the message, and then exits.

Closing your window does not cause COMS to terminate your program; you must disable the program or the window.

Some Useful COMS Commands

The following commands are useful for sending and receiving messages:

- DISABLE PROGRAM <program name>

  COMS submits a null message to the program with a value of 99 in the Status Value field. For this command to work, the program must receive the message, and it must go to end of task (EOT) whenever it detects a Status Value message of 99. A disabled program cannot be started up, and messages for it are rejected.

- DISABLE WINDOW <window name>

  COMS sends messages with a value of 99 in the Status Value field to all programs running in this window. The programs are not themselves disabled.

- ENABLE PROGRAM <program name>/WINDOW <window name>

  The program or programs defined in the window can be started up to receive messages. Note that there is no enable library command. Libraries are started again as soon as a linkage is requested.

Accessing Applications

As a COMS user, you can choose to access applications in any of the following ways:

- Through an existing message control system (MCS) window
- As remote-file programs
- By writing new programs or converting existing applications for use in a direct-window environment

MCS Window Applications

Any of your applications that currently run in CANDE can be initiated from the CANDE window or from the MARC window. You can also run these CANDE applications as COMS-declared remote-file programs. In addition, you can define customized CANDE windows with some characteristics tailored to the programs you plan to run in those windows. See the *COMS Configuration Guide* for more information on defining new windows. Refer also to the *A Series CANDE Configuration Reference Manual* and the *A Series Menu-Assisted Resource Control (MARC) Operations Guide* for information about initiating programs in a CANDE or a MARC environment.

For information about running GEMCOS applications through the GEMCOS window or installing your own MCS in the MCS window, refer to the *A Series Communications Management System (COMS) Migration Guide.*

Remote-File Programs

If you have existing programs that use the GEMCOS (or some other) remote-file interface, or if you have applications that can only receive input from one station per copy of the program, you can run these programs in COMS remote-file windows. For information about running remote-file programs in the COMS environment, refer to Section 9, "Communicating with COMS through Remote-File Windows."

Direct-Window Applications

To take advantage of COMS flexibility, you can convert your existing applications to a COMS environment or develop new applications for use in the COMS environment.

To create a COMS direct-window application or to convert an application to run in a COMS direct-window environment, you need to make a number of decisions relating to the following issues:

- Processing items
- Your security scheme
- Trancode routing
- Synchronized recovery
- Program use of COMS features
- Modifications to the COMS configuration file

Processing Items

Some decisions you should make about processing items are

- What processing items, if any, should be written for your COMS applications in general?
- What processing items are needed for the application you are designing?
- How should these processing items be linked together in agendas?
- What processing-item library convention should be used?

Refer to Section 5, "Processing Items," for information about programming a processing item. Refer to the *COMS Configuration Guide* for information about defining processing items in the configuration file.

Your Security Scheme

When planning your security scheme, consider the following questions:

- What combination of COMS and application-based security measures will you use?
- If you do use COMS security, what features (including control of station access, program access, and usercode access to windows, trancodes, and stations) are appropriate for your needs? (Remember that trancode security measures are available only with full-featured COMS.)
- If you control station access, what trancodes will be permitted from a given window?

- If you control program access, what trancodes will be embedded in the output messages of a given program?
- If you control access of usercodes, what stations and windows can a given usercode be logged on to, and what trancodes can that usercode use?
- To what stations, if any, do you wish to assign the continuous log-on capability?
- If COMS security features do not fill all your security needs, then what processing items or security routines in your direct-window programs can add needed security to your COMS applications?

Refer to Section 8, "Security," for information on programming for security, and to the *COMS Configuration Guide* for more information on the configuration file security measures.

Trancode Routing

If you are considering using trancode routing, some basic questions that need to be answered are

- Where in the message will you place the trancode?
- Will you be using the module function index (MFI) feature in your direct-window programs?
- What trancodes will you use?
- What security categories will you apply to the trancodes?
- What trancodes will you allow to be used by what usercodes?
- What agenda routing do you plan for each trancode?

See Section 3, "Communicating with COMS through Direct Windows," for more information about programming for trancodes. For additional information on trancodes and security categories, refer to the *COMS Configuration Guide.*

Database Recovery

The following questions should be asked if you have full-featured COMS and will be updating DMSII databases:

- Do you plan to use synchronized recovery?
- Do you plan to use protected input queues?
- Are you presently using the DMSII single-abort feature?
- Do your programs use concurrency control?

If you are updating SIM databases, the following questions should be asked:

- Do you plan to use protected input queues?
- Do you plan to use archival recovery?

Refer to Section 6, “Interactive Recovery,” and to the *COMS Operations Guide* for details of synchronized recovery.

Program Use of COMS Features

Consider the following questions as you are planning how to create new direct-window applications or convert existing applications:

- Does the program have message areas to receive and send messages?
- How will the program be initialized?

Refer to Section 3, “Communicating with COMS through Direct Windows,” for information about writing direct-window programs.

Modifying the COMS Configuration File

The following questions are important in determining how to change the COMS configuration file:

- What processing items need to be defined to COMS?
- How are they to be combined into agendas?
- What windows need to be defined?
- What agendas will be associated with what windows?
- What trancodes need to be defined?
- What programs need to be defined to COMS?
- What access will you allow to windows, trancodes, and stations?
- What stations, usercodes, and programs will be given access?
- What databases need to be defined?

These questions are only a sample of the decisions that need to be made. For more information about modifying the COMS configuration file, refer to the *COMS Configuration Guide.*

Section 3
Communicating with COMS through Direct Windows

When you use the direct-window interface of COMS, you have access to all the COMS features and functions, including the following:

- Service functions that let you translate names into designator values and designator values into names to manipulate the entities of the COMS environment
- Security checking of messages that programs receive and send
- Processing items that can process message data before and after programs receive and send it
- Dynamic opening of direct windows to terminals and dynamic communication over a modem

If you are using the full-featured version of COMS, you also have access to the following features:

- Message routing by transaction codes (trancodes)
- Synchronized recovery for multiple database-processing programs that are running asynchronously

This section presents each of the programming tasks necessary for communicating with COMS through direct windows and discusses the following topics:

- Preparing a message area
- Initializing a program
- Creating a designator table
- Programming to receive messages
- Programming to send messages
- Dynamically attaching to and detaching from stations

The following program example provides the program flow for a direct-window program that does not interact with a database, and shows in pseudolanguage the necessary programming steps discussed throughout this section. Indentation is used in the pseudolanguage to indicate the scope of each statement.

```
  ********************
  * DECLARATION PART *
  ********************

COMS HEADER CDIN;
COMS HEADER CDOUT;
DATA AREA MSG;
  *********************
  * PROGRAM STRUCTURE *
  *********************

     INITIALIZE_COMS;
     WHILE CDIN.STATUS NOT = 99 DO
          RECEIVE CDIN INTO MSG;
          IF CDIN.STATUS NOT = 99 THEN
               HANDLE_MSG;
               SEND CDOUT FROM MSG;
     EXIT_COMS;

  *******************
  * INITIALIZE_COMS *
  *******************
%Set up library title for COMS; Call to enable input%

     ESTABLISH_COMS_LINK;
     ENABLE INPUT COMS "ONLINE";

  *************
  * EXIT_COMS *
  *************
%No handling required%

     EXIT PROGRAM;
```

Preparing a Message Area

To receive and send messages, you must define a message area in which your program can build messages. You need to define the message area with a size and format appropriate to the data your program sends or receives. If a message area is too small to contain all the text for a message being sent or received, COMS truncates the message.

COMS uses input and output headers to control the routing and description of your message. The available fields of these headers are presented later in this section in “Programming to Receive Messages” and “Programming to Send Messages.”

In a program that performs updates with Screen Design Facility (SDF) forms, the message area can receive the SDF form if you take the following steps:

1. Define matching offsets for the message area and the SDF form. (The SDF form that you define with the SDF system must be at the same offset.)
2. Name the SDF form instead of the message area record when using statements that receive or send a message.

For examples of message areas declared in your programming language, refer to the appropriate language manual.

Initializing a Program

After you have prepared your message area, the next step required to prepare a program to send and receive messages is to provide for program initialization.

To initialize your application program, you must do the following:

1. Link to COMS.
2. Provide for initializing the COMS interface, either in interactive or batch mode.

For the specific program statements in your programming language, see the section on COMS program initialization in the appropriate language manual.

Creating a Designator Table

Often a program needs the designators that are associated with elements of the configuration file, such as agendas, data communication devices, programs, security categories, or usercodes. For this reason, you might want to include a table of element names and corresponding designators in your program. By using a table of this sort, your program can avoid making a call on COMS on each input for the designators the program needs. Each entity in the configuration file has a designator that can be used to reference the entity. This designator remains the same throughout an execution. If, however, you do a dump and load of the configuration file, the designators will change.

Because the layout of COMS designators may change with each software release, a program should never preserve any designator across executions. Do not, for example, use designators as keydata in a database, except in a restart data set.

To create a designator table, do the following:

1. Pass a name to get a designator by using the appropriate COMS service function. Refer to Section 4, “Accessing Service Functions,” for additional information on COMS service functions.
2. Create a table of configuration file element names and corresponding designators.

Programming to Receive Messages

The following paragraphs provide information about receiving messages, as well as instructions for determining the message origin from fields in the input header.

Using the Input Header

If you want your program to receive a message from COMS, you must use an input header. A header (or message header), which is a sequence of characters separated from the data message itself, provides routing and descriptive information for a message. The header is an enhanced version of the communication descriptor that was used in the previous releases of COMS. The input header can be accessed both in processing items and in direct-window programs.

Caution

When you set up the receive area of your message header, include enough space for the characters of the message and all associated trancodes. If you do not allow adequate space, the message will be truncated.

Depending on the language in which you are programming, the header name can either be defined by COMS or be a variable name that you choose. For specific information on using your programming language to define a header, see the appropriate language manual.

Input Header Fields

COMS places values into the input header fields when a direct-window program executes a program statement that receives or accepts a message or that enables input to a terminal. The values describe the status of, and the circumstances encountered by, each message received by the program. Exceptions to this rule are as follows:

- If your program is updating a Data Management System II (DMSII) database, COMS places a designator representing the last audited message into the Restart field when the program executes a BEGIN-TRANSACTION with <text> statement or a MID-TRANSACTION statement.
- COMS passes to the receiving program any data that a processing item has placed in the Conversation Area field of the input header. For information on processing items, see Section 5, "Processing Items."

The values COMS places into the input header fields are designators or integers that are part of an internal code understood by COMS and used in the COMS table structure. For most of the designators placed in the input header, you can use a service function of the COMS library to translate the designator to a name representing a COMS entity.

Service functions also allow you to translate names representing COMS entities to designators you use in output headers.

The following is a list of input header fields and a description of their contents. To see the layout of COMS headers, see Appendix B.

Program Designator Field

This field can contain the following designators:

- When a program or terminal is enabled, this field contains the program designator that COMS assigns to the program. COMS uses this information for database recovery operations. For more information on database recovery, see Section 6, "Interactive Recovery."
- When a message is received, this field contains the program designator of the program that originated the incoming message. If a station originated the message, this field contains a 0 (zero).

Function Index Field

This field contains a user-defined positive module function index (MFI) value that can be used for routing by trancode and security checking.

A mnemonic is provided in Pascal and RPG for a Function Index field value of 0 (zero). See Table A–1 in Appendix A for a list of these mnemonics.

Function Status Field

This field can contain one of the following values:

- A COMS-defined error value
- Values that report on the status of a dynamic attachment to another terminal
- Values that are results of delivery confirmation requests for output messages
- Values that are COMS notifications to direct windows of on/open activity, closure of the window, or a break condition in output from the window
- Values that indicate the specific reason COMS has told a program to terminate

For information on the meaning of specific values or mnemonics in the Function Status field, see Appendix A.

Usercode Designator Field

This field contains a designator representing the usercode associated with the program or station that originated the message.

Security Designator Field

This field contains a session security designator.

VT Flag Field

This field is returned by COMS on input after the program has set the VT flag on output. The VT flag should be used only within a CP 2000 environment. See "Setting the VT Flag" later in this section for more information.

Transparent Field

This field indicates whether the input message is being passed in transparent mode (that is, with no data formatting or translation).

Timestamp Field

This field contains the time and date (in the TIME(6) system format) that a message was first encountered by COMS. COMS audits the transaction trail for the time and date appearing in this field when the program executes a MID-TRANSACTION statement.

Station Designator Field

This field can contain the following designators:

- If a program receives a message, this field contains a designator representing the station that originated the incoming message.
- If a program originated the message, this field contains the station designator found in the Station Designator field of the input header of the originating program.
- If a program dynamically attaches or detaches a terminal, the station designator in the Station Designator field represents the attached or detached terminal.

Text Length Field

This field can contain the following values:

- When a program receives a message, this field contains the number of characters in the text of the incoming message.
- When the program enables an input terminal with the DIAL option, this field contains the length of the phone number of the destination.
- When a program requests delivery confirmation on output, this field contains the length of the delivery confirmation result returned by COMS on input.
- This field contains a value of 0 (zero) when COMS notifies a direct window of on/open activity with no text, closure of a window, or a break in output from the window.

Status Value Field

This field contains a numeric value that provides the following information:

- Confirmation as to whether an output message successfully reached its destination
- Identification of a synchronized recovery message
- Status of a message after it is processed by a processing item
- Indication that a program dynamically attached or detached a terminal, or was unable to do so
- Indication that a program was asked to terminate

For information on the meanings of the values and associated mnemonics in this field, see Appendix A.

Message Count Field

When a program executes a statement to determine whether messages from COMS are waiting in the queue of the program, this field contains the number of queued messages.

Restart Field

When a direct-window program enters the transaction state while updating a DMSII database for a previous release, this field contains a designator representing the last message that COMS audited in the transaction trail. The Pascal programming language does not support a COMS DMSII interface and therefore does not use this field.

Agenda Designator Field

When a program receives a message, this field contains the designator of the most recently applied input agenda.

When you use a processing item to call OUTPUT_PROC with an input header and you want to specify an agenda, the agenda designator must be placed in this field.

SDF Information Field

COMS uses this field internally.

Conversation Area Field

This field is optional and is the only user-defined field in the header. For the correct syntax to use when defining the field, see the appropriate language manual. This field can contain the following information:

- Information passed by a program to processing items, by processing items to other processing items, and by processing items to a program.
- The phone number of the destination, if a direct-window program enables an input terminal with the DIAL option.
- Information that a program puts in the transaction trail for its own DMSII or SIM recovery purposes. Recovery data is passed back to the application program when transactions are resubmitted by COMS. Recovery data is unique to a particular transaction and must not be used to retain information between transactions, such as storing running totals in a program.

Detecting Queued Messages

You can use a statement in your program to determine whether messages from COMS are waiting in the queue of the program. See the appropriate language manual for this program statement. If messages are queued, the program can perform a routine that receives messages from COMS. If the program queue does not contain any messages from COMS, the program can perform a routine that processes messages from other sources.

When a program executes the statement, COMS places the number of queued messages in the Message Count field of the input header. If more than one copy of your program is running, the value obtained by executing the statement is the sum of messages queued for all the copies. Since your program might not be able to receive all the queued messages reported by the statement, you must not use the Message Count field as a loop controller.

After executing the statement, the program queries the Status Value field of the input header. Table 3–1 shows the four possible values the field can contain.

Table 3–1. Possible Values for the Input Header Status Value Field

Value	Description
0	The Message Count field contains the number of messages queued for the program.
92	This value indicates that COMS is currently performing a synchronized recovery on a DMSII database. The program executes the RECEIVE statement to handle the recovery transactions.

continued

Table 3–1. Possible Values for the Input Header Status Value Field (cont.)

Value	Description
93	The program has aborted and then restarted. The Message Count field contains a nonzero value. This value does not indicate the total number of messages queued for the program. The program should execute RECEIVE statements to handle the recovery.
99	The program will be directed to terminate, but it should execute a RECEIVE statement to get its transactions and its status 99 message.

For mnemonics associated with these values, see Appendix A in this guide.

In some programming languages, the program does not need to query the Status Value field. The call itself returns the value. See the appropriate language manual for further information.

Waiting for COMS Input and Task Events

COMS provides a library entry point, the DCIWAITENTRYPOINT, that programs can call when waiting for an event to happen. Event-driven ALGOL programs can wait for two COMS events. In some cases, the nature of the transaction-processing program is such that it requires more awareness of external events than those handled by the COMS DCI interface. In these cases, in ALGOL, a program can wait for other time periods and required events in addition to the COMS events.

For a given transaction processor (TP), COMS causes either an input event or a task event to happen. One input event is shared among all copies of a program. The input event remains set to HAPPENED if messages are in the input queue of the program. All copies of the program waiting for the input event are *awakened*. On the other hand, each copy of a program has its own task event. COMS causes the task event to happen for a specific copy of a program when COMS wants to perform a task-specific function, such as giving a copy instructions to go to end of task (EOT) or to process a message during recovery.

TPs must not reset either of these events. COMS causes the event to happen at the appropriate time and resets the event when it is no longer needed.

The program must declare a real value procedure that contains the statement with the WAIT option set. The procedure must also take the two events as formal parameters, and other events and variables must be visible to this procedure. The following is an example of a real value procedure that contains the value returned by the DCIWAITENTRYPOINT function:

```
REAL PROCEDURE DCIWAITENTRYPOINT (  WAIT PROC )
      REAL PROCEDURE WAIT_PROC ( COMS_INPUT_EVENT ,
                                 COMS_TASK_EVENT ) ;
            EVENT
               COMS_INPUT_EVENT ,
               COMS_TASK_EVENT ;
         FORMAL ;
   LIBRARY COMS_DCI_LIBRARY ;

INTEGER
   MY_TIMEOUT ;
FILE
   MY_REMOTE_FILE ( KIND = REMOTE ) ,
   MY_PORT_FILE ( KIND = PORT ) ;

EVENT
   MY_COROUTINE_EVENT ;

REAL PROCEDURE MY_WAIT ( COMS_INPUT_EVENT ,
                         COMS_TASK_EVENT ) ;
      EVENT
         COMS_INPUT_EVENT ,
         COMS_TASK_EVENT ;
   BEGIN
   MY_WAIT := WAIT ( ( MY_TIMEOUT ) ,
                     COMS_INPUT_EVENT ,
                     COMS_TASK_EVENT ,
                     MY_REMOTE_FILE.INPUTEVENT ,
                     MY_PORT_FILE.INPUTEVENT ,
                     MY_PORT_FILE.CHANGEEVENT ,
                     MY_COROUTINE_EVENT ) ;
   END MY_WAIT ;

CASE DCIWAITENTRYPOINT ( MY_WAIT ) OF
   BEGIN
   ; % NEVER RETURNS (Ø).
   HANDLE_TIMEOUT ;
   HANDLE_COMS_INPUT ;
   HANDLE_COMS_TASK ;
   HANDLE-REMOTE_FILE_INPUT ;
   HANDLE_PORT_FILE_INPUT ;
   HANDLE_PORT_FILE_CHANGE ;
   HANDLE_COROUTINE_EVENT ;
   END ;
```

If the DCIWAITENTRYPOINT result indicates that either a COMS input is present or a COMS task event has happened, the program performs a conditional COMS *RECEIVE* function (that is, a RECEIVE statement with the DONTWAIT option set).

More than one copy of a program might be running at the same time. Every copy of a program that has called the DCIWAITENTRYPOINT function is awakened by the triggering of the input event, but only one copy receives the message that caused the event to be triggered. As a result, all other copies could wait indefinitely for the next transaction if they request a COMS *RECEIVE* function with the WAIT option set.

The DCIWAITENTRYPOINT function can be used only by ALGOL programs because all other programming languages do not allow procedures to take events as formal parameters when calling a subroutine.

Receiving a Message

A RECEIVE statement in your program informs COMS that you are ready to process a message. COMS, in turn, sends the message to your message area.

You can use a RECEIVE statement as many times as needed in 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.

The syntax for the use of the RECEIVE statement depends upon the programming language you are using. See the appropriate language manual for further information on using the RECEIVE statement in your program.

Determining the Origin of a Message

To determine the origin of a message, check the Program Designator field and the Station Designator field in the input header. In addition, check the Status Value field of the input header to determine the message status. You are not required to know the origin of a message, but it is strongly recommended to check the message status.

Refer to "Using the Input Header" in this section for additional information on the input header.

To determine a message origin from the input header, use the following procedure:

1. Check the Program Designator field. When a program executes a RECEIVE statement, the Program Designator field of the input header contains one of the following:

   - A designator representing the program that originated the message
   - The value 0 (zero) or a null designator to indicate that the message came from a station

   If this field contains a value of 0 (zero), you must check the Station Designator field for the originating station.

2. Check the Station Designator field. When a program executes a RECEIVE statement and a station originated the message, the Station Designator field of the input header contains a designator representing the station that originated the message.

   If a program originated the message, this field contains the station designator found in the Station Designator field of the input header of the originating program.

Using Module Function Indexes with Input

To facilitate the processing of messages using the transaction code routing method, you can associate a module function index (MFI) with each trancode or group of trancodes. This association is made in the configuration file. For instructions on using the COMS Utility to assign these MFIs, refer to the *COMS Configuration Guide.*

Once these MFIs are defined, any time a program executes a RECEIVE statement, the Function Index field of the input header contains the MFI assigned to the trancode associated with the incoming message, unless the message came from another program. The MFI can be used for routing messages to particular transaction-processing routines within a program, assuming that the value of the MFI can be matched to a routine in a program.

COMS checks the validity of the trancode before reaching your program. If the trancode is invalid, COMS applies the default input agenda to the message. If no default input agenda has been defined, COMS rejects the message and the station receives an error message. The Function Status field and the Status Value field of the input header should be checked for error messages. Refer to Appendix A for the possible error values that COMS returns.

Obtaining Direct-Window Notifications

COMS automatically returns values to certain input header fields of direct-window programs through the default input agenda when a *?ON <window name>* or *?CLOSE <window name>* command is issued to the direct window, or when a break in output from the direct window is detected. These values appear in the input header the next time the destination program of the default input agenda receives a message with a RECEIVE statement.

When you define a direct window with the COMS Utility program, you can specify Open Notification and On Notification text that the window receives for an initial opening and for every subsequent opening. If you do not provide any text for On and Open Notification (blanks are the default value), the Text Length field of the input header contains a value of 0 (zero). For information on values that appear in the Function Status field when you do not provide any text for On and Open Notification, see Appendix A.

When a break condition causes output from a direct window to be discarded, COMS reports this to the direct window by routing an input to the default input agenda of the direct window and by placing a value of 0 (zero) in the Text Length field of the input header. See Appendix A for values placed in the Function Status field.

The break notification is the only message that will inform the program that messages were discarded. Even if delivery confirmation has been requested on any of the discarded messages, a delivery confirmation rejection message is not sent with that message.

When you close a direct window with the *?CLOSE <window name>* command, COMS reports the closure by routing an input to the default input agenda of the direct window and by placing a value of 0 (zero) in the Text Length field of the input header. See Appendix A for values placed in the Function Status field.

Manipulating Closed Window Dialogues

After you close a window dialogue, any message sent to that dialogue is discarded. You are then assigned another current window based on the type of window you are closing and how the window is defined in the configuration file, as follows:

- If the window you are closing is a dynamic remote-file window, you are assigned to the dialogue of the MARC window from which the dynamic remote-file window was initiated.
- If the window you are closing is a direct window (defined in the configuration file), an MCS window, or a declared remote-file window, then the action COMS takes when closing the window is determined by the configuration file. You can specify which window you want to be transferred to when your current window closes. The close action value that is specified for your station or your usercode in the configuration file determines the action that is taken. Refer to the *COMS Configuration Guide* for information on possible close actions.

Checking the Status of Input Messages

You can determine the status of an input message by checking the Status Value field in the input header. When a RECEIVE statement is executed, a value is returned in the Status Value field to indicate the status of a message. When the Status Value is 99, the program should perform its end routines and go to end of task (EOT).

The meanings of the values are described in Appendix A.

Note: *Future releases of COMS might add values to the Status Value field of the input header. You need to keep this in mind when writing a program that makes use of the values this field can contain.*

Programming to Send Messages

You can use a SEND statement in your program to send messages to stations or to other programs, although you can send a message to only one destination with each SEND statement.

You can send a message to multiple stations or programs in two ways. The first method uses multiple send commands; each one specifies a different destination. The second method uses an agenda that includes a processing item that calls the OUTPUT_PROC procedure to send the message to multiple destinations. Refer to "OUTPUT_PROC Parameter" in Section 5, "Processing Items," for additional information about the OUTPUT_PROC procedure.

If you plan to have your program send a message using COMS, you must use an output header. The following pages describe the output header and its fields.

Using the Output Header

A header, or message header, is a sequence of characters, separated from the data message itself, that provides routing or descriptive information for a message.

Depending on the language in which you are programming, the header name can either be defined by COMS or be a variable name that you choose. For specific information on using your programming language to define a header, see the appropriate language manual.

Output Header Fields

Place designators into the output header fields to route outgoing messages and describe their characteristics. You can obtain designators by calling service functions of the COMS library. If an error occurs when a program sends an outgoing message, COMS returns an error value to the Status Value field described later in this section. For the layout of COMS headers, see Appendix B.

Destination Count Field

This integer field specifies the number of destinations to which the program can send a message. COMS supports only one destination.

Text Length Field

Use this field to specify the number of characters contained in the text of an outgoing message; it is the only way to inform COMS about the length of your message.

Status Value Field

With this field you can specify an agenda designator for postprocessing of the message that a program is sending. It is recommended that you use the Agenda Designator field for this purpose. If your window has a default output agenda, you do not need to specify an agenda for postprocessing.

After the program sends a message, COMS returns an integer value to this field to indicate whether the message was successfully sent to its destination or if an error occurred.

For information on specific values and mnemonics, see Appendix A.

Carriage Control Field

This field can be accessed either by a processing item or a program. A program can modify this field either directly or by means of an explicit carriage-control specification in the SEND statement. If there is no carriage-control specification in the SEND statement, COMS does not alter the value that might have been entered directly into this field.

A processing item can modify this field to change the carriage control specification for an output message that was initially provided by a program either directly or in its SEND statement.

Delivery Confirmation Flag Field

Use this field to request delivery confirmation of an output message.

Delivery Confirmation Key Field

This field is used by a program requesting delivery confirmation to uniquely identify each message. When COMS subsequently confirms or denies delivery, this identifying value is returned as part of the confirmation message. See "Requesting Delivery Confirmation" in this section for more information on confirming delivery of a message.

VT Flag Field

Use this field to set the virtual terminal flag on an output message to a CP 2000 station. See "Setting the VT Flag" later in this section for more information.

Transparent Field

Use this field to specify transparent mode for an output message to a CP 2000 station. Transparent mode is an operating mode in which there is no data formatting or translation.

Destination Designator Field

This field specifies an optional destination for a message.

Next Input Agenda Designator Field

If the Set Next Input Agenda field is equal to 1 (TRUE), this field specifies the agenda that is to be applied to the next input to the current dialogue of the destination station. For information on specifying the next input agenda, see "Program-Specified Input Agendas" in this section.

Set Next Input Agenda Field

This field specifies whether COMS is to use the contents of the Next Input Agenda Designator field to change the agenda for the next input to the current dialogue of the destination station. For information on specifying the next input agenda, see "Program-Specified Input Agendas" in this section.

Retain Transaction Mode Field

This field specifies whether transaction mode is to be retained for the current dialogue. A value of 1 (TRUE) indicates that transaction mode is not to be cleared for the dialogue when the output is delivered. A value of 0 (FALSE) indicates that transaction mode is to be cleared. For information on the use of transaction mode, see "Using Agendas for Message Routing" in this section.

Casual Output Field

Use this field to produce casual output for a protected transaction whose output is protected by default. Enter *1* in this field to produce casual output. If you enter *0*, the type of output produced depends on whether or not the transaction is protected. (If the transaction is protected, the output is protected.) COMS does not change the value of the field once it has been enabled. Therefore, be sure to change the value back to 0 if you are using the same COMS output header with subsequent output messages that you want protected.

Casual output is any output that is not protected. Casual output is delivered immediately except for delays when output tanking is necessary; the casual output can be lost if delivery is interrupted by a system failure. All COMS output for transaction that are not protected is casual output.

Agenda Designator Field

With this field you can specify an agenda for postprocessing of the message that a program is sending. The Status Value field can still be used for this purpose, but use of the Agenda Designator field is recommended.

If processing items do not change the contents of the Agenda Designator field, it does not need to be reloaded every time the program sends a new message. The Status Value field does not offer this feature.

SDF Information Field

This field is used internally by COMS.

Conversation Area Field

This field is optional and is the only user-defined field in the header. For the correct syntax to use when defining the field, see the appropriate language manual.

Use this field to pass information (in addition to the message data) to processing items. For example, to use an SDF form with a COMS direct-window program, place a form key in the first word of the Conversation Area field.

Declaring Multiple Input and Output Headers

You can declare multiple input and output headers in your program to move messages in and out of different processing states. By declaring multiple headers, you can maintain the flow of incoming and outgoing messages.

The following two examples illustrate the flow of programs that use multiple headers. The first example contains one input header and two output headers. The second example contains multiple input and output headers.

Example 1

```
000400 IDENTIFICATION DIVISION.
000500 PROGRAM-ID. TEST-FILE.
000600 ENVIRONMENT DIVISION.
000700 CONFIGURATION SECTION.
000800 SOURCE-COMPUTER. A-SERIES.
000900 OBJECT-COMPUTER. A-SERIES.
001000 INPUT-OUTPUT SECTION.
001100 FILE-CONTROL.
001200     SELECT PRT-FILE ASSIGN TO PRINTER.
001300 DATA DIVISION.
001400 FILE SECTION.
001500 FD  PRT-FILE.
001600 01  PRINT-REC.
001700     02 FILLER           PIC X(132).
001800 WORKING-STORAGE SECTION.
001900 01  MY-AREA.
002000     02  JUNK-IN         PIC X(20).
002100 01  JUNK-1.
002200     02  JUNK-2          PIC 9(5) USAGE BINARY.
002300     02  JUNK-OUT        PIC S9(11) USAGE BINARY.
002400 01  WS-AREA.
002500     02  FILLER          PIC X(100).
002600     02  FILLER          PIC X(20)  VALUE  "FIRST  MESSAGE IS  :".
002700     02  WS-MSG1         PIC X(20).
002800     02  FILLER          PIC X(40).
002900     02  FILLER          PIC X(20)  VALUE  "SECOND MESSAGE IS  :".
003000     02  WS-MSG2         PIC X(20).
003100     02  FILLER          PIC X(40).
003200     02  FILLER          PIC X(20)  VALUE  "THIRD  MESSAGE IS  :".
003300     02  WS-MSG3         PIC X(20).
003400     02  FILLER          PIC X(1620).
003500 01  XYZ.
003600     02  ALWAYS          PIC X(160).
003700     02  THETWO          PIC X(1760).
003800 COMMUNICATION SECTION.
```

```
ØØ39ØØ INPUT HEADER COMS-IN
ØØ4ØØØ PROGRAMDESG              IS COMS-IN-PROGRAM
ØØ41ØØ FUNCTIONSTATUS           IS COMS-IN-FUNCTION-STATUS
ØØ42ØØ FUNCTIONINDEX            IS COMS-IN-FUNCTION-INDEX
ØØ43ØØ USERCODE                 IS COMS-IN-USERCODE
ØØ44ØØ SECURITYDESG             IS COMS-IN-SECURITY-DESG
ØØ45ØØ TRANSPARENT              IS COMS-IN-TRANSPARENT
ØØ46ØØ VTFLAG                   IS COMS-IN-VT-FLAG
ØØ47ØØ TIMESTAMP                IS COMS-IN-TIMESTAMP
ØØ48ØØ STATION                  IS COMS-IN-STATION
ØØ49ØØ TEXTLENGTH               IS COMS-IN-TEXT-LENGTH
ØØ5ØØØ STATUSVALUE              IS COMS-IN-STATUS-KEY
ØØ51ØØ AGENDA                   IS COMS-IN-AGENDA
ØØ52ØØ SDFINFO                  IS COMS-IN-SDF-INFO.
ØØ53ØØ OUTPUT HEADER COMS-OUT
ØØ54ØØ DESTCOUNT                IS COMS-OUT-COUNT
ØØ55ØØ TEXTLENGTH               IS COMS-OUT-TEXT-LENGTH
ØØ56ØØ STATUSVALUE              IS COMS-OUT-STATUS-KEY
ØØ57ØØ TRANSPARENT              IS COMS-OUT-TRANSPARENT
ØØ58ØØ VTFLAG                   IS COMS-OUT-VT-FLAG
ØØ59ØØ CONFIRMFLAG              IS COMS-OUT-CONFIRM-FLAG
ØØ6ØØØ CONFIRMKEY               IS COMS-OUT-CONFIRM-KEY
ØØ61ØØ DESTINATIONDESG          IS COMS-OUT-DESTINATION
ØØ62ØØ NEXTINPUTAGENDA          IS COMS-OUT-NEXT-INPUT-AGENDA
ØØ63ØØ CASUALOUTPUT             IS COMS-OUT-CASUAL-OUTPUT
ØØ64ØØ SETNEXTINPUTAGENDA       IS COMS-OUT-SET-NEXT-INPUT-AGENDA
ØØ65ØØ RETAINTRANSACTIONMODE    IS COMS-OUT-SAVE-TRANS-MODE
ØØ66ØØ AGENDA                   IS COMS-OUT-AGENDA
ØØ67ØØ SDFINFO                  IS COMS-OUT-SDF-INFO
ØØ68ØØ CONVERSATION AREA.
ØØ69ØØ     Ø2 MSG-AREA            PIC X(192Ø).
ØØ7ØØØ OUTPUT HEADER COMS-OUT1
ØØ71ØØ DESTCOUNT                IS COMS-OUT-COUNT
ØØ72ØØ TEXTLENGTH               IS COMS-OUT-TEXT-LENGTH
ØØ73ØØ STATUSVALUE              IS COMS-OUT-STATUS-KEY
ØØ74ØØ TRANSPARENT              IS COMS-OUT-TRANSPARENT
ØØ75ØØ VTFLAG                   IS COMS-OUT-VT-FLAG
ØØ76ØØ CONFIRMFLAG              IS COMS-OUT-CONFIRM-FLAG
ØØ77ØØ CONFIRMKEY               IS COMS-OUT-CONFIRM-KEY
ØØ78ØØ DESTINATIONDESG          IS COMS-OUT-DESTINATION
ØØ79ØØ NEXTINPUTAGENDA          IS COMS-OUT-NEXT-INPUT-AGENDA
ØØ8ØØØ CASUALOUTPUT             IS COMS-OUT-CASUAL-OUTPUT
ØØ81ØØ SETNEXTINPUTAGENDA       IS COMS-OUT-SET-NEXT-INPUT-AGENDA
ØØ82ØØ RETAINTRANSACTIONMODE    IS COMS-OUT-SAVE-TRANS-MODE
ØØ83ØØ AGENDA                   IS COMS-OUT-AGENDA
ØØ84ØØ SDFINFO                  IS COMS-OUT-SDF-INFO
ØØ85ØØ CONVERSATION AREA.
ØØ86ØØ     Ø2 MSG-AREA-1          PIC X(192Ø).
ØØ87ØØ PROCEDURE DIVISION.
ØØ88ØØ INITIAL-PARA.
ØØ89ØØ     CHANGE ATTRIBUTE LIBACCESS OF "DCILIBRARY" TO BYFUNCTION.
ØØ9ØØØ     CHANGE ATTRIBUTE FUNCTIONNAME OF "DCILIBRARY" TO
```

```
009100          "COMSSUPPORT".
009200     ENABLE INPUT COMS-IN KEY "ONLINE".
009300 HK-PARA.
009400     MOVE "COMS" TO PRINT-REC.
009500     MOVE "IT IS OK" TO XYZ.
009600     RECEIVE COMS-IN MESSAGE INTO XYZ.
009700     MOVE 1920 TO COMS-OUT-TEXT-LENGTH OF COMS-OUT1.
009800     MOVE "HELLO YOU ARE IN. " TO ALWAYS.
009900     MOVE "PLEASE GIVE THREE MEANINGFUL MESSAGES." TO THETWO.
010000     SEND COMS-OUT1 FROM XYZ.
010100     RECEIVE COMS-IN MESSAGE INTO MY-AREA.
010150     MOVE JUNK-IN TO WS-MSG1.
010200     MOVE 1920 TO COMS-OUT-TEXT-LENGTH OF COMS-OUT.
010300     SEND COMS-OUT FROM JUNK-IN WITH ESI.
010400     RECEIVE COMS-IN MESSAGE INTO MY-AREA.
010450     MOVE JUNK-IN TO WS-MSG2.
010500     SEND COMS-OUT FROM JUNK-IN WITH ESI.
010600     RECEIVE COMS-IN MESSAGE INTO MY-AREA.
010650     MOVE JUNK-IN TO WS-MSG3
010700     SEND COMS-OUT FROM JUNK-IN WITH ESI.
010800     MOVE 1920 TO COMS-OUT-TEXT-LENGTH OF COMS-OUT.
010900     SEND COMS-OUT FROM WS-AREA WITH EGI AFTER ADVANCING 10 LINES.
011000     IF COMS-IN-STATUS-KEY = 99 GO TO EOJ.
011100     DISPLAY XYZ.
011200     ACCEPT COMS-IN MESSAGE COUNT.
011300 EOJ.
011400     STOP RUN.
```

Example 2

```
000100$ RESET FREE
000150******* $ SET LISTDOLLAR
000200******* $ SET LIST WARNSUPR
000300 IDENTIFICATION DIVISION.
000400 PROGRAM-ID. GOOD-SYNTAX.
000500*
000600*  ALL VALID SYNTAX PRODUCTIONS FOR COMS HEADERS.
000700*  THERE SHOULD BE NO SYNTAX ERRORS.
000800*
000900 ENVIRONMENT DIVISION.
001000 CONFIGURATION SECTION.
001100 SOURCE-COMPUTER. A3.
001200 OBJECT-COMPUTER. A3.
001300*
001400 DATA DIVISION.
001500*
001600 COMMUNICATION SECTION.
001700 INPUT HEADER IH1.
001800 INPUT HEADER IH2 PROGRAMDESG COMS-PROGRAMDESG.
001900 INPUT HEADER IH3 PROGRAMDESG IS COMS-PROGRAMDESG.
002000 INPUT HEADER IH4 CONVERSATION AREA CA SIZE 123.
002100 INPUT HEADER IH5 CONVERSATION AREA IS CA SIZE 123.
```

```
ØØ22ØØ INPUT HEADER IH6 CONVERSATION AREA IS CA SIZE IS 123.
ØØ23ØØ INPUT HEADER IH7 CONVERSATION AREA. Ø2 CA PIC X(123).
ØØ24ØØ INPUT HEADER IH8 CONVERSATION AREA.
ØØ25ØØ        Ø2 CA. Ø5 CA1 PIC X. Ø5 CA2 PIC X.
ØØ26ØØ INPUT HEADER IH9 VTFLAG IS V CONVERSATION AREA CA SIZE 123.
ØØ27ØØ INPUT HEADER IHA VTFLAG IS V CONVERSATION AREA IS CA SIZE 123.
ØØ28ØØ INPUT HEADER IHB VTFLAG IS V CONVERSATION AREA IS CA SIZE IS 123.
ØØ29ØØ INPUT HEADER IHC VTFLAG IS V CONVERSATION AREA. Ø2 CA PIC X(123).
ØØ3ØØØ INPUT HEADER IHD VTFLAG IS V CONVERSATION AREA.
ØØ31ØØ        Ø2 CA. Ø5 CA1 PIC X. Ø5 CA2 PIC X.
ØØ32ØØ INPUT HEADER IHE VTFLAG V CONVERSATION AREA CA SIZE 123.
ØØ33ØØ INPUT HEADER IHF VTFLAG V CONVERSATION AREA IS CA SIZE 123.
ØØ34ØØ INPUT HEADER IHG VTFLAG V CONVERSATION AREA IS CA SIZE IS 123.
ØØ35ØØ INPUT HEADER IHH VTFLAG V CONVERSATION AREA. Ø2 CA PIC X(123).
ØØ36ØØ INPUT HEADER IHI VTFLAG V CONVERSATION AREA.
ØØ37ØØ        Ø2 CA. Ø5 CA1 PIC X. Ø5 CA2 PIC X.
ØØ38ØØ*
ØØ39ØØ OUTPUT HEADER OH1.
ØØ4ØØØ OUTPUT HEADER OH2 AGENDA COMS-AGENDA.
ØØ41ØØ OUTPUT HEADER OH3 AGENDA IS COMS-AGENDA.
ØØ42ØØ OUTPUT HEADER OH4 CONVERSATION AREA CA SIZE 123.
ØØ43ØØ OUTPUT HEADER OH5 CONVERSATION AREA IS CA SIZE 123.
ØØ44ØØ OUTPUT HEADER OH6 CONVERSATION AREA IS CA SIZE IS 123.
ØØ45ØØ OUTPUT HEADER OH7 CONVERSATION AREA. Ø2 CA PIC X(123).
ØØ46ØØ OUTPUT HEADER OH8 CONVERSATION AREA.
ØØ47ØØ         Ø2 CA. Ø5 CA1 PIC X. Ø5 CA2 PIC X.
ØØ48ØØ OUTPUT HEADER OH9 VTFLAG IS V CONVERSATION AREA CA SIZE 123.
ØØ49ØØ OUTPUT HEADER OHA VTFLAG IS V CONVERSATION AREA IS CA SIZE 123.
ØØ5ØØØ OUTPUT HEADER OHB VTFLAG IS V CONVERSATION AREA IS CA SIZE IS 1.
ØØ51ØØ OUTPUT HEADER OHC VTFLAG IS V CONVERSATION AREA. Ø2 CA PIC X(1).
ØØ52ØØ OUTPUT HEADER OHD VTFLAG IS V CONVERSATION AREA.
ØØ53ØØ         Ø2 CA. Ø5 CA1 PIC X. Ø5 CA2 PIC X.
ØØ54ØØ OUTPUT HEADER OHE VTFLAG V CONVERSATION AREA CA SIZE 123.
ØØ55ØØ OUTPUT HEADER OHF VTFLAG V CONVERSATION AREA IS CA SIZE 123.
ØØ56ØØ OUTPUT HEADER OHG VTFLAG V CONVERSATION AREA IS CA SIZE IS 123.
ØØ57ØØ OUTPUT HEADER OHH VTFLAG V CONVERSATION AREA. Ø2 CA PIC X(123).
ØØ58ØØ OUTPUT HEADER OHI VTFLAG V CONVERSATION AREA.
ØØ59ØØ         Ø2 CA. Ø5 CA1 PIC X. Ø5 CA2 PIC X.
ØØ6ØØØ*
ØØ61ØØ PROCEDURE DIVISION.
ØØ62ØØ*
ØØ63ØØ MAIN-SECTION SECTION.
ØØ64ØØ MAIN-PARA.
ØØ65ØØ      MOVE 1 TO AGENDA OF IH1.
ØØ66ØØ      MOVE 1 TO COMS-PROGRAMDESG OF IH2.
ØØ67ØØ      MOVE 1 TO COMS-PROGRAMDESG OF IH3.
ØØ68ØØ      MOVE "HI" TO CA OF IH4.
ØØ69ØØ      MOVE "HI" TO CA OF IH5.
ØØ7ØØØ      MOVE "HI" TO CA OF IH6.
ØØ71ØØ      MOVE "HI" TO CA OF IH7.
ØØ72ØØ      MOVE "H" TO CA1 OF IH8. MOVE "I" TO CA2 OF IH8.
ØØ73ØØ      IF V OF IH9 NEXT SENTENCE.
```

```
ØØ74ØØ     IF V OF IHA NEXT SENTENCE. MOVE "HI" TO CA OF IHA.
ØØ75ØØ     IF V OF IHB NEXT SENTENCE. MOVE "HI" TO CA OF IHB.
ØØ76ØØ     IF V OF IHC NEXT SENTENCE. MOVE "HI" TO CA OF IHC.
ØØ77ØØ     IF V OF IHD NEXT SENTENCE. MOVE "H" TO CA1 OF IHD.
ØØ78ØØ     IF V OF IHE NEXT SENTENCE. MOVE "HI" TO CA OF IHE.
ØØ79ØØ     IF V OF IHF NEXT SENTENCE. MOVE "HI" TO CA OF IHF.
ØØ8ØØØ     IF V OF IHG NEXT SENTENCE. MOVE "HI" TO CA OF IHG.
ØØ81ØØ     IF V OF IHH NEXT SENTENCE. MOVE "HI" TO CA OF IHH.
ØØ82ØØ     IF V OF IHI NEXT SENTENCE. MOVE "H" TO CA1 OF IHI.
ØØ83ØØ*
ØØ84ØØ     MOVE 1 TO AGENDA OF OH1.
ØØ85ØØ     MOVE 1 TO COMS-AGENDA OF OH2.
ØØ86ØØ     MOVE 1 TO COMS-AGENDA OF OH3.
ØØ87ØØ     MOVE "HI" TO CA OF OH4.
ØØ88ØØ     MOVE "HI" TO CA OF OH5.
ØØ89ØØ     MOVE "HI" TO CA OF OH6.
ØØ9ØØØ     MOVE "HI" TO CA OF OH7.
ØØ91ØØ     MOVE "H" TO CA1 OF OH8. MOVE "I" TO CA2 OF OH8.
ØØ92ØØ     IF V OF OH9 NEXT SENTENCE.
ØØ93ØØ     IF V OF OHA NEXT SENTENCE. MOVE "HI" TO CA OF OHA.
ØØ94ØØ     IF V OF OHB NEXT SENTENCE. MOVE "HI" TO CA OF OHB.
ØØ95ØØ     IF V OF OHC NEXT SENTENCE. MOVE "HI" TO CA OF OHC.
ØØ96ØØ     IF V OF OHD NEXT SENTENCE. MOVE "H" TO CA1 OF OHD.
ØØ97ØØ     IF V OF OHE NEXT SENTENCE. MOVE "HI" TO CA OF OHE.
ØØ98ØØ     IF V OF OHF NEXT SENTENCE. MOVE "HI" TO CA OF OHF.
ØØ99ØØ     IF V OF OHG NEXT SENTENCE. MOVE "HI" TO CA OF OHG.
Ø1ØØØØ     IF V OF OHH NEXT SENTENCE. MOVE "HI" TO CA OF OHH.
Ø1Ø1ØØ     IF V OF OHI NEXT SENTENCE. MOVE "H" TO CA1 OF OHI.
Ø1Ø2ØØ     STOP RUN.
```

COMS Message-Routing Logic

When you use a SEND statement to send a message, the message is routed by COMS based on the presence and validity of the following items:

- A destination, as indicated in the Destination Designator field of the output header.
- An agenda, as indicated in the Agenda Designator field of the output header or a default output agenda.
- An agenda-specified destination, as indicated by the destination defined for the agenda in COMS Utility. Note that a destination specified in the Destination Designator field overrides an agenda-specified destination.

Table 3–2 is a decision table that indicates how COMS routes messages when these items either have assigned values (Y) or do not have assigned values (N), or when the presence of an item does not matter (–). An explanation of the action codes follows the decision table.

Table 3–2. Message-Routing Decision Table: Items with Assigned Values (Yes/No/–)

Item	Value Assigned							
Header Destination	Y	Y	Y	N	N	N	N	N
Header Agenda	Y	N	N	Y	Y	N	N	N
Agenda Destination	–	–	–	Y	N	–	–	–
Default Output Agenda	–	Y	N	–	–	Y	Y	N
Agenda Destination	–	–	–	–	–	Y	N	–
Action Code	1	1	2	3	4	3	4	5

The action codes (values from 1 to 5) in Table 3–2 are associated with the actions COMS takes during message routing. These actions are explained in Table 3–3.

Note: *Do not specify an agenda as the default output agenda if it contains a processing item that calls OUTPUT_PROC and passes an output header that does not specify an agenda. This combination can cause recursive calls on the same processing item, eventually causing a STACK OVERFLOW error. To prevent this error, a processing item that calls OUTPUT_PROC should specify an agenda other than the default output agenda in its output header.*

Table 3–3. Message-Routing Action Codes

Action Code	Action Taken
1	If the destination is a station, COMS executes any agenda-specified processing items associated with the message and sends the message to the destination. If the destination is a program, the message and the agenda designator are placed in the input queue of the program. When the program executes a RECEIVE statement, COMS removes the message from the queue, applies the agenda-specified processing items, and passes the message to the program.
2	COMS sends the message to the destination.
3	In this case, the agenda-specified destination is a program. As a result, the message and the agenda designator are placed in the input queue of program. When the program executes a RECEIVE statement, COMS removes the message from the queue, applies the agenda-specified processing items, and passes the message to the program.

continued

Table 3–3. Message-Routing Action Codes (cont.)

Action Code	Action Taken
4	COMS sends the message to the originating station of the last received message after executing any agenda-specified processing items associated with the message.
5	COMS sends the message to the originating station.

Sending a Message

A SEND statement initiates the sending of a message to a station or a program. You can use the SEND statement as many times as needed in your program.

Direct window programs normally run in a window. However, this does not have to be true in all cases. It is possible, using the COMS Utility, to define a program that has no window associated with it. This is done by not having any agenda with the program as its destination. If this "agendaless" program is run, COMS cannot associate it with any window, and functions for getting designators and for sending output to stations will not work in the same way as if the program ran in a window.

A program running outside a window can only send to a station if its input came from another program running in a window, and the destination is the same station as originated the message. The reason for this behavior is that COMS has to know to which terminal window to send the message.

Using Segmented Output

To send segmented messages, see Table 3–4 for the COMS options.

Table 3–4. Segmented Message Options

Send Option	Description
End-of-Message Indicator (EMI)	Unsegmented output that is sent immediately. The length of the message is not necessarily the entire length of the Message Area variable defined, but the length entered into the Text Length field.This option is the default value.
End-of-Group Indicator (EGI)	Segmented output that has been held with the ESI option is sent along with the current message. Any carriage controls that are used here are executed only with the current message.

continued

Table 3–4. Segmented Message Options (cont.)

Send Option	Description
End-of-Segment Indicator (ESI)	With this option, data is taken from the Message Area variable and put into a temporary holding space the size of the length entered into the Text Length field. This is not necessarily the size of the entire message area variable. The data is then held until one of the following takes place: • A RECEIVE statement is processed. At this time, all segmented output is sent to COMS. • A SEND statement is processed with the EGI option. At this point, all segmented output is sent to COMS in the order in which it was originally processed. • A SEND statement that contains the ESI option and directs a message to another station is processed. In this case, a SEND statement ends message queuing for a station because COMS queues messages for only one station at a time.

When your program uses segmented output, the MCS waits to transmit any portion of a message until the entire message is placed in the output queue. The appropriate EMI or EGI message indicator must be included at the end of a message or the MCS does not recognize the message and does not send it. After a run has stopped, messages without an EMI or EGI at the end are not sent, and are purged from the system.

If multiple SEND statements are processed with the ESI option, and a SEND statement with the EMI option is processed in the middle of these, the SEND statement with the EMI option is sent immediately, while the others must wait until one of the ESI output conditions is true. As a result, in some cases the messages might appear to be sent in an incorrect order.

The syntax for the use of your SEND statement varies according to the programming language you are using. For information on using the SEND statement in your programming language, see the appropriate language manual.

Routing Messages by Specifying a Destination

When you include a SEND statement in your program, you can also specify a message destination in the output header, unless you want the message to be returned to its originating station.

To specify a destination, make entries in the output header fields as shown in Table 3–5.

Table 3–5. Output Header Field Descriptions

Field	Description
Destination Count field	This field must contain the value 1, since COMS supports a single destination per SEND statement.
Destination Designator field	You can enter a specific destination in this field for direct routing.
Agenda Designator field or Status Value field	These fields can contain an agenda designator from which a destination can be derived. The destination specified in the agenda is overridden by any destination specified in the COMS Destination Designator field.

When your program executes a SEND statement, COMS reads the value in the Destination Designator field to determine the destination for your outgoing message. To specify direct routing, enter a station designator or a program designator. You can specify only one destination per SEND statement.

Routing Messages by Using Agendas

Agendas are entities of the configuration file. Each agenda can consist of an optional list of processing items and an optional destination. Processing items cannot be accessed individually by name, but only by group name, that is, in a processing-item list. The processing of a message sent by a program before the message reaches its destination is referred to as postprocessing.

To use an agenda on output, you do not need to assign a destination to the message because the destination can be determined in the Destination Designator field or the predefined destination at installation time.

Using the Agenda Activity menu of COMS Utility, you can create a default output agenda, which might or might not specify a destination. If it does, the destination must be a program (or INPUT_ROUTER, for transaction-based routing). There can be only one default output agenda per window. For additional information on agendas, see the *COMS Configuration Guide.*

To apply an agenda to a message on output, do the following:

1. Get the agenda designator by passing the agenda name to the appropriate COMS service function. For additional information on COMS service functions, refer to Section 4, “Accessing Service Functions.”
2. Put the agenda designator in the Agenda Designator field of the output header.

3. Check the value in the Destination Designator field of the output header to see whether it contains the desired destination for the message. The destination specified in this field overrides the destination specified by the agenda in the Agenda Designator field of the output header. Move a null value or 0 (zero) into the Destination Designator field of the output header if you want to use the destination specified by the agenda designated in the Agenda Designator field of the output header.
4. Use a SEND statement to send the message and a copy of the output header.

If your window specification contains a default output agenda, and you want to have that agenda applied to your message, steps 1 and 2 are not necessary.

After you send an outgoing message, the Status Value field of the output header contains a value that indicates the status of the message. These values are listed in Appendix A.

If an error occurred and you failed to clear the error value from the Status Value field, the field functions as if you had set it to null (that is, cleared of information) the next time your program executes a SEND statement.

If there is not a designator in the Destination Designator field, the Agenda Designator field of the output header, or the Status Value field of the output header, the message is directed to the station originating the transaction.

If the destination of the message is determined to be a station, the processing items associated with the agenda are applied to the message immediately.

If the destination is a program, the message and the agenda designator are placed in the input queue of the program. When the message is removed from the queue by COMS as a result of the execution of the destination program of a RECEIVE statement, the processing items are applied to the message.

Note: *When using agendas to route messages, the program must execute a RECEIVE statement from a terminal—not from a program—before a default output agenda is applied to messages. Also, second default output agendas cannot contain processing items that call OUTPUT_PROC without supplying a valid agenda designator. Those that do are discontinued for exceeding resource limitations (R-DS).*

Using Transaction Mode

Transaction mode prevents a user from entering input until the response from the previous input has been received. There are two agenda attributes that control transaction mode: SET TRANSACTION MODE and TRANSACTION-MODE AGENDA. Input sent to an agenda that has the SET TRANSACTION MODE attribute set to Y in the COMS Utility causes the transaction-mode state for that dialogue to be set to 1 (TRUE). A TRANSACTION-MODE AGENDA attribute specifies an optional agenda that has a program that processes input from dialogues that are already in transaction mode. Only one TRANSACTION-MODE AGENDA attribute can be defined for a window. If no TRANSACTION-MODE AGENDA attribute is defined for a window, the transactions are routed to the dialogue of the MARC/1 station. MARC discards the input and sends the message “Previous input still in process; message rejected”.

When an initial input message is associated with an agenda configured for transaction mode, all subsequent input to that dialogue is processed as specified, by either the TRANSACTION-MODE AGENDA attribute for the current window or the dialogue of the MARC/1 station until the transaction mode is cleared. During this process, you can switch to another window or dialogue; switching does not affect the transaction mode state of the previous dialogue, unless that dialogue is explicitly closed.

For information on specific values and mnemonics associated with transaction mode, see Appendix A. For more information on transaction mode, see the *COMS Configuration Guide.*

To clear transaction mode, do one of the following:

- When you program your transaction processor (TP) or processing item to send a message to a dialogue, place a value of zero (FALSE) in the Retain Transaction Mode field of the output header. When COMS receives confirmation that the message has been delivered, routing is returned to the normal specifications, unless a previous SEND statement under transaction mode has set the next input agenda for this dialogue.
- Execute another RECEIVE statement without an intervening SEND statement. The transaction mode is cleared and the dialogue is returned to normal routing specifications.
- Close your dialogue so that the dialogue that was in transaction mode no longer exists.

If you do not want your TP or processing item to clear transaction mode, place a value of 1 (TRUE) in the Retain Transaction Mode field of the output header when the SEND statement is executed.

The user is responsible for maintaining the transaction-mode state of any dialogue. The TRANSACTION-MODE AGENDA attribute merely permits a second program to process when transaction mode is already set, and permits the user to clear transaction mode when desired.

If transaction mode, without an associated TRANSACTION-MODE AGENDA remains set after the first program terminates, the user must close the window to clear the condition.

If the second program terminates, the user must close the window to clear the condition.

Routing Messages by Trancode

You can use the Agenda Designator field of the output header to route messages by the trancode (transaction code) contained in the message. A trancode is a sequence of characters that can be included in a message. You can also use trancodes to invoke processing tasks in an application program within a direct window.

To route messages by trancode, place in the Agenda Designator field of the output header a designator representing an agenda whose destination is INPUT_ROUTER. INPUT_ROUTER is an entry point in the Router library, which is an internal COMS library. Refer to the *COMS Configuration Guide* for more information about trancodes.

Because COMS program entities do not have a Trancode Position field, any message routed to INPUT_ROUTER from a program defaults to a trancode position of 1. Only messages routed directly from a station use the trancode position value indicated in the COMS configuration file station record. Refer to "Determining the Origin of Messages" earlier in this section for more information about message origin.

To use Trancode Security when routing messages by trancode, (as defined on the Trancode screen in the COMS Utility), you must establish a current session security for the program executing the SEND statement. Session security is established when the program executes its first RECEIVE statement. The session security is the intersection of the station security and the usercode security of the incoming message. If you execute a SEND statement before first executing a RECEIVE statement, the current session security is null and COMS rejects the message by returning a value of 98 in the COMS-OUT-STATUS-KEY field of the output header.

Program-Specified Input Agendas

Your program can specify the agenda to be applied to the next input message received from a dialogue at a particular station. To do this, the Set Next Input Agenda field of the output header must be set to 1 (TRUE). At the same time, the Next Input Agenda Designator field of the header should contain the designator of the desired agenda. COMS changes the agenda when it receives confirmation that the output message was delivered to the station. If the Set Next Input Agenda field is 1 (TRUE) and the Next Input Agenda Designator field contains a 0 (zero), COMS resets the routing of inputs to whatever is specified in the window configuration.

If the target dialogue is in transaction mode, you can encounter one of the following situations:

- If the SEND statement also clears transaction mode, the next input received from the dialogue (after receipt of confirmation) is processed by the agenda in the Next Input Agenda Designator field of the SEND statement.
- If the SEND statement does not clear transaction mode, input continues to be routed according to transaction-mode specifications until a CLEAR is sent. After your program receives confirmation of a clearing message, the next input to the dialogue is processed by the previously established input agenda. If more than one SEND statement sets the next input agenda, the last one for which confirmation has been received goes into effect when transaction mode is cleared.

Once your program sets the input agenda for a dialogue, that specification remains in effect until it is set to another agenda or is reset to normal routing, or until the station closes the dialogue. If you have configured this new agenda for transaction mode, this agenda is returned to when transaction mode is cleared. For more information on transaction mode, see "Using Transaction Mode" in this section.

If your destination program aborts, normal COMS error handling takes effect, but the programmatically set input agenda is not canceled.

If your program requests delivery confirmation when it establishes a new input agenda for a dialogue, the confirmation message is sent to the destination of the new input agenda, not the default agenda. For more information on delivery confirmation, see "Requesting Delivery Confirmation" in this section.

Setting the VT Flag

The VT (virtual terminal) flag of the output header can be used with a COMS direct window, which can have a virtual terminal name when it is used within a CP 2000 environment. The virtual terminal name describes to BNA how the direct window has formatted the output.

For information on setting the flag using your programming language, see the appropriate language manual.

For information on virtual terminals, refer to the *A Series BNA Version 2 Capabilities Overview.*

Requesting Delivery Confirmation

Delivery confirmation is available for network support processor (NSP) stations and for CP 2000 stations. This feature of COMS lets a direct window know when a station has received a particular message the window has sent. To request delivery confirmation for an output message, set the Delivery Confirmation flag to 1 (TRUE) and place unique values of your choice in the Delivery Confirmation Key field of the output header before executing a SEND statement.

For information on field names for your programming language, see Appendix B.

When a program sends a message for which delivery confirmation is requested, COMS returns a confirmation result to either the default input agenda of the window that generated the output or the agenda established by that message, if the program specified a program-specified input agenda. For more information on program-specified input agendas, see "Program-Specified Input Agendas" in this section. The confirmation result is contained in certain fields of the input header, and in the first three bytes of the message area of the destination program of the default input agenda. As soon as COMS confirms successful delivery of the message, COMS sends the confirmation result to the destination program of the default input agenda.

If the destination program of the default input agenda is not the same program that requested delivery confirmation, a program-to-program message can be sent to the requesting program to provide the confirmation result.

Results for Successful Messages

For a message that has successfully reached its destination, COMS confirms the delivery of the message by returning a value to the input header in the Function Status field. For the meaning of these values and their corresponding mnemonics, see Appendix A.

COMS also returns a value of 3 to the Text Length field of the input header, and returns in the first three bytes of the message area the unique value that was placed in the Delivery Confirmation Key field of the output header before the original message was sent.

Results for Rejected Messages

If the CP 2000 terminal gateway rejects an output message for which delivery confirmation was requested, COMS returns a value of 6 in the Text Length field of the input header and a value to the Function Status field. For the meaning of the Function Status field values and their corresponding mnemonics, see Appendix A.

A value of 6 is returned to the Text Length field of the input header, indicating that the rejection message is 6 bytes long. The first 3 bytes of the rejection message contain what was in the Delivery Confirmation Key field when the original message was sent.

If a break condition causes output from a direct window to be discarded, a break notification is sent, but no separate delivery confirmation rejection message is sent for each discarded message, even if delivery confirmation was requested.

Bytes 4, 5, and 6 of the rejection message contain the following information:

- A value in byte 4 is an error code denoting the reason for the rejection of the message. Error values currently defined are the following:

Value	Description
1	Invalid virtual terminal data
2	Truncated data
3	Virtual terminal not supported
4	Out-of-band error

- Values in bytes 5 and 6 describe the location and origin of the message data that the CP 2000 terminal gateway was processing when it detected the error.

Checking the Status of Output Messages

When your program executes a SEND statement, COMS places a value in the Status Value field of the output header to indicate whether an error has occurred.

Your program should check this value to determine whether an error has occurred in the SEND statement or the output header. The meanings of the values and their corresponding mnemonics are described in Table A–3.

Note: *Future releases of COMS might add values to the Status Value field of the output header. You need to keep this in mind when writing a program that makes use of the values this field can contain.*

Attaching Dynamically to Stations

You can open dynamically a direct window to a station not currently attached to COMS. COMS supports both CP 2000 and network support processor (NSP) stations, which might exist on the same system and use the same direct windows.

Before executing the ENABLE statement, the program must specify the station to which the window will be opened. To do this, the program must move a station designator for the destination station into the Station Designator field of the input header.

If the station specified as the destination is currently attached to another program, the window of the station is opened but is not made the current window. If you use a CP 2000 station or use the DIAL option on an NSP, the window of the station is made the current window.

After COMS has successfully attached a station, the direct window that initiated the attachment becomes the default window for the station, overriding all other default window settings. This means that the station automatically starts communication with the window that caused COMS to initiate the attachment after the user log-on sequence. If the station has a default usercode, the log-on sequence is bypassed entirely.

COMS is unable to attach an X.25 station or to initiate an X.25 call until the link layer has been established. The link layer must be established outside of COMS.

Using Key Options on Attachment

Key options are literals defined for use with COMS. Enclose the literal in quotation marks when you use it in the statement that enables the input terminal, or when you place a literal into a data name you have declared. The key options are

- KEY DIAL
- KEY WAIT
- KEY NOWAIT
- KEY WAITNOTBUSY
- KEY WAITDIALOUT

Use the KEY DIAL option only with NSP stations on switched lines. For attaching NSP stations not on switched lines, any option other than DIAL can be used. KEY DIAL allows a program to communicate dynamically over a modem with a station, if you place values in three fields of the input header as follows:

1. Move a station designator for the destination station into the Station field of the input header.
2. Move the phone number of the destination into the Conversation Area field of the input header.
3. Move the length of the phone number into the Text Length field of the input header.

The other options connected with enabling a station, KEY WAIT, KEY NOWAIT, KEY WAITNOTBUSY, and KEY WAITDIALOUT, are significant for CP 2000 stations. Use these options to specify how COMS should wait to attach a station and, optionally, the hostname on which the station is located.

If no option is specified or if KEY WAIT is specified, COMS waits for both a physical attachment (a dialout) to the station, and for the station to enter a "not busy" state.

If KEY NOWAIT is specified, COMS attaches the station only if it is not busy and is already physically attached.

If KEY WAITNOTBUSY is specified, COMS waits for the station to enter "not busy" state, but does not wait for a physical attachment.

If KEY WAITDIALOUT is specified, COMS waits for a physical attachment to be made, but not for the station to enter "not busy"state.

If a hostname is specified, COMS attaches the program to a station on that host. If a hostname is not specified, COMS attaches to a station on the host defined for the station in the input header of the program.

For information on the syntax of the ENABLE statements and key options in your programming language, see the appropriate language manual.

Checking Attachment Status

To find out the status of a request to attach a station programmatically, write your program to query values in the Status Value field and the Function Status field of the input header after the program enables a station. The meanings of the values are described in Appendix A.

Detaching Dynamically from Stations

You can close dynamically a window to a station, or disconnect a station reached through a modem. The station whose window will be closed is the station identified by the station designator in the Station Designator field of the input header.

COMS requests that the CP 2000 terminal gateway detach a station if the following conditions prevail:

- The program disabling a station must be a program in the current window of the station being detached.
- No dialogues to other windows are open, except dialogue 1 of the MARC window.

Using Key Options on Detachment

The key options are literals defined for use with COMS. Enclose a literal in quotation marks when you use it in a statement that disables an input terminal or when you place a literal into a data name you have declared. The key options are

- KEY RETAIN
- KEY RELEASE
- KEY DIAL
- KEY DONTCARE

The KEY RETAIN, KEY RELEASE, and KEY DONTCARE options are for CP 2000 stations. The KEY DIAL option is for NSP stations on switched lines. For NSP stations not on switched lines, any key option other than KEY DIAL can be used.

If you use the KEY RETAIN option, the CP 2000 terminal gateway retains the physical attachment of the station and terminates only the logical attachment.

If you use the KEY RELEASE option, the CP 2000 terminal gateway terminates the logical attachment and releases the physical attachment (that is, it hangs up the phone).

If you use neither the KEY RETAIN nor the KEY RELEASE options, the CP 2000 terminal gateway decides whether to retain or release the physical attachment to the station.

If you use the KEY DIAL option in the ENABLE statement, you should use KEY DIAL in the DISABLE statement to detach the station that was reached through the modem.

If you use the KEY DONTCARE option, the CP 2000 terminal gateway decides whether to retain or release the physical attachment to the station.

For information on the syntax of the DISABLE statements and key options in your programming language, see the appropriate language manual.

Checking Detachment Status

To find out the status of a request to detach a station programmatically, write your program to query values in the Status Value field and Function Status field of the input header after the program executes the DISABLE statement. The meanings of the values are described Appendix A.

Break Condition

Standard Unisys data comm software issues a break condition when a user enters *?BRK* or presses the break key at a station. COMS then processes the break for the current and resumed windows of that station. For each such window, any tanked output is discarded and a break condition message is routed to the default input agenda of the window. See Appendix A for the tables of values and mnemonics for the value placed in the Function Status field of the input header to indicate a break condition message. A direct window program obtains this message when it executes a RECEIVE statement and there are no other prior queued messages for the window.

A direct-window program can continue to send output without checking whether a break condition message has been routed to it by COMS. As soon as COMS has queued the break condition message to the program, COMS stops discarding any messages sent by the program. Therefore, any output sent by the direct window program after this point is displayed at the station.

Section 4
Accessing Service Functions

This section applies to the full-featured version of COMS. If you have developed your programs using a previous release of COMS, see the service functions in Appendix F. For further information about the entities and designators discussed in this section, see the *COMS Configuration Guide.*

This section describes the service functions or entry points that COMS provides. Entry points are procedures that allow access to a library code file. Service functions allow you to obtain information on all of the entities in the configuration file.

All entities in the COMS configuration file have designators and can be used in service function calls. For each entity, you can use your program to receive information about the following:

- Entity name
- Installation data

For some entities, you can receive the following additional information:

Entity	Information Available
Program	Current input queue depth
	Mix numbers for active copies
	Response time aggregate
	Response time for last transaction
	Security designator
	Total number of input messages handled
Station	Device designator
	Logical station number (LSN)
	Screen size
	Security designator
	Virtual terminal (VT) type
Station List	Stations in list
Usercode	Security designator
Window	Current number of users
	Maximum number of users

Application programs can communicate with COMS through a direct-window interface by using a COMS header. Refer to Section 3, "Communicating with COMS through

Direct Windows" for information on this procedure. This interface also allows communication between COMS and application programs to be enhanced by the use of designators. A designator is a binary data type that can be included in a program to control messages symbolically rather than directly with entities in the data comm environment.

Each COMS service function is an integer procedure of the COMS Library. When you use a service function, you can exchange either a name that represents an entity for a designator or a designator for a name that represents an entity.

Note: *Always use the space character to initialize arrays for entity names in your programs. When COMS returns an entity name in response to a service function call, it uses space characters to blank-fill the remainder of the array. Thus, programs that initialize and scan for null characters will fail.*

Before you call service functions, you need a general understanding of how they use designators and names, and what their input and output parameters are.

Designators and Names

The service functions use designators to constitute an internal code understood by COMS. The designators are used in the table structure of the system. Because the layout of COMS designators may change with each software release, a program should never preserve any designator across executions. Do not, for example, use designators as keydata in a database.

The COMS service functions enable you to

- Translate designators to names that represent COMS entities.
- Translate names that represent COMS entities to designators.
- Obtain additional information about the designator passed to the service function.

You can obtain designators from the message header of an application program or from those service functions that allow you to translate names to designators. (See Section 3, "Communicating with COMS through Direct Windows.")

Input and Output Parameters

When you pass a name or a designator to a service function, the name or designator is used as an input parameter. The COMS Library returns output parameters and function values. All service functions return an integer specifying the result of the call in an area you define to receive your result. For the syntax for your programming language, see the appropriate language manual. For information on the specific result values and mnemonics of service function calls, see Appendix A.

You can get designators from several different places. You can get them from various fields of the input header. (See Section 3, "Communicating with COMS through Direct

Windows," for a discussion of header fields.) You can also get designators by using the following service functions:

- GET_DESIGNATOR_USING_NAME
- GET_DESIGNATOR_USING_DESIGNATOR
- GET_DESIGNATOR_ARRAY_USING_DESIGNATOR

Agenda, trancode, installation data, and station designators are different from designators that represent other COMS entities. Whereas a program, for example, can always obtain the same valid designator that identifies only the program itself, each designator for the agenda, trancode, and installation data entities must uniquely identify a particular combination of a window and that entity. In the same way, each station designator uniquely identifies a particular combination of a window, a dialogue, and a station.

Always comply with the following restrictions to make sure that you get valid designators when using the GET_DESIGNATOR_USING_NAME service function for agendas, trancodes, and installation data.

- Call this service function only from direct-window programs.
- Call the service function only after a direct-window program has enabled an input terminal.
- Do not allow a processing item to call the service function until the library code of the processing item has executed a FREEZE statement.

Use of Installation Data

Installation data are used to attach special data to specific configuration elements. The data can take the form of comments or program-accessible fields to be used by user-created applications. Typical uses of installation data would include:

- District or branch number for location of a station.
- Test or production mark for a program.
- A set of data to group entities in the configuration according to some installation-specific rules.

To the programmer, installation data items are handled in the same manner as the other items in the COMS configuration. Some differences, however, need to be explained.

The items for an installation data entity are accessed using the following service function calls:

Service Function Call	Items Accessed
GET_INTEGER_USING_DESIGNATOR	Accesses the following items one at a time: • Installation_Integer_1 • Installation_Integer_2 • Installation_Integer_3 • Installation_Integer_4
GET_INTEGER_ARRAY_USING_DESIGNATOR	Uses Installation_Integer_All to concatenate all the items listed under GET_INTEGER_USING_DESIGNATOR.
GET_STRING_USING_DESIGNATOR	Accesses the following items one at a time: • Installation_String_1 • Installation_String_2 • Installation_String_3 • Installation_String_4 • Installation_Hex_1 • Installation_Hex_2

The designator passed to these service functions can be one of the following:

- An installation data entity designator, which you can obtain from an installation data name by using the GET_DESIGNATOR_USING_NAME service function or using the GET_DESIGNATOR_USING_DESIGNATOR service function. In this case, the entity is used in a direct manner to access the data.
- Any other entity designator that has a link to an installation data entity. In this case, the link is used as the path to the data.

For example, these two methods of getting an installation integer for a station designator produce the same result:

Method 1

1. Have a station designator.
2. Use the GET_INTEGER_USING_DESIGNATOR service function with the station designator.

Method 2

1. Have a station designator.
2. Use the GET_DESIGNATOR_USING_DESIGNATOR service function with the station designator and the request for Installation_Data_Link.

3. Use the GET_INTEGER_USING_DESIGNATOR service function with that installation data designator.

The following sequence of steps gets an installation integer for a station when the station is linked to an installation data item that in turn is linked to another installation data item containing the installation integer:

1. Have a station designator.
2. Use the GET_DESIGNATOR_USING_DESIGNATOR service function with the station designator and the request for Installation_Data_Link.
3. Use the GET_DESIGNATOR_USING_DESIGNATOR service function with the installation data designator received in step 2.
4. Use the GET_INTEGER_USING_DESIGNATOR service function with the installation data designator from step 3.

Because installation data are application dependent, each installation data entity resides in a particular window. However, an entity can have a window of "ALL" specified, in which case it resides in all windows. Therefore, if a program running in one window requests installation data, it might get different data than would a program running in another window. COMS keeps track of where each installation date entity resides so that the program does not have to have any any knowledge of the different windows.

Explanation of Mnemonics for Service Functions

For each service function, the allowable mnemonics for entities are listed later in this section. Table 4–1 provides brief descriptions of each mnemonic.

Table 4–1. Descriptions of Service Function Mnemonics

Mnemonic	Description
Aggregate_Response_Time	Sum of transaction response times. Accumulated by COMS.
Convention	Connection default convention. Assigned by MARC.
Current_User_Count	Number of users on a window. Kept by COMS.
Installation_Data_Link	An entity pointed to by another COMS entity. Assigned by the COMS Utility.
Installation_String_1, Installation_String_2, Installation_String_3, Installation_String_4, Installation_Hex_1, Installation_Hex_2	Strings of installation-specific data. Assigned by the COMS Utility.

continued

Table 4–1. Descriptions of Service Function Mnemonics (cont.)

Mnemonic	Description
Installation_Integer_1, Installation_Integer_2, Installation_Integer_3, Installation_Integer_4	Integers of installation-specific data. Assigned by COMS Utility.
Language	Connection default language. Assigned by MARC.
Last_Response_Time	Program response time in milliseconds for the last transaction handled. Kept by COMS.
LSN	Logical station number. Assigned by the data comm subsystem.
Maximum_User_Count	Maximum allowed number of users on a window. Assigned by the COMS Utility.
Mixnumbers	Array of mix numbers for the copies running for the program. Kept by the operating system.
Screen Size	Statically configured attribute that represents the size of the terminal attached to the system.
Statistics	A gathering of program statistics. Values accumulated by COMS.
Total_Transaction_Count	Number of transactions received by the program since COMS initialized. Kept by COMS.
Transaction_Queue_Depth	Number of transactions in queue for the program. Kept by COMS.
Virtual Terminal	Name of the terminal's text editor in the communications processor. Kept by COMS.

Calling Service Functions

You can call the COMS service functions with application programs and processing items. For the specific syntax for calling a service function using your programming language, see the appropriate language manual.

The service functions are

- 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

The station table service functions that appear in the previous list perform differently than the other service functions. Specifically, the station table functions provide the user-application program with a unique integer value for a given station designator. When provided a station designator, these functions perform the following tasks:

- Allocate a location (index) in a table
- Preserve the index within the table
- Supply the table index to the application program on request

The table index is owned by the TP and is passed to these station table service functions on each call. The table can be searched by multiple inquirers concurrently. This table can also be searched while it is being updated. However, the inquirer is responsible for preventing concurrent updates.

The table can be resized by the functions. However, the table size is generally more accurate when the caller declares an initial size based on the number of stations that might be added to the table.

The following subsections describe each service function and provide information on how to use the service functions to obtain information on specific entities.

CONVERT_TIMESTAMP

You can use this service function to convert a COMS timestamp, TIME(6), into a form that you can read.

The declaration in COMS is as follows:

```
INTEGER PROCEDURE CONVERT_TIMESTAMP(ENTY_TIMESTAMP,
                                    ENTY_MNEMONIC,
                                    ENTY_TIME);
```

Parameters

ENTY_TIMESTAMP is the TIME(6) timestamp that is returned in the Timestamp field of the input header.

ENTY_MNEMONIC represents the information you are requesting. The allowable mnemonics are as follows:

- The Date mnemonic returns the date in the form of MMDDYY.
- The Time mnemonic returns the time in the form of HHMMSS.

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

GET_DESIGNATOR_ARRAY_USING_DESIGNATOR

You can use this service function to get a designator list representing the list of stations associated with the station-list designator passed as the input parameter to this function. Because the station list is not a field of the header, you must first obtain the designator representing the station list by using the GET_DESIGNATOR_USING_NAME service function. Then use the station list designator supplied by that function as the designator variable in the GET_DESIGNATOR_ARRAY_USING_DESIGNATOR function.

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_DESIGNATOR_ARRAY_USING_DESIGNATOR
                                                  (ENTY_DESIGNATOR,
                                                   ENTY_DESGTOTAL,
                                                   ENTY_DESGVECTOR);
```

Parameters

ENTY_DESIGNATOR represents the structure. The allowable designator is station list, which returns a list of stations.

ENTY_DESGTOTAL is the total number of designators in the list by COMS.

ENTY_DESGVECTOR is the list in which the designators are returned.

GET_DESIGNATOR_USING_DESIGNATOR

You can use this service function to get a specific designator out of the structure represented by a designator. The designator to be passed as an input parameter can be any designator. COMS looks at the designator and the mnemonic passed and returns the designator information associated with the two.

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_DESIGNATOR_USING_DESIGNATOR(ENTY_DESIGNATOR,
                                                  ENTY_MNEMONIC,
                                                  ENTY_DESGRES);
```

Parameters

ENTY_DESIGNATOR represents the structure from which you want to get a designator.

ENTY_MNEMONIC is the designator type you are requesting. The allowable mnemonic and designator combinations are as follows:

Mnemonic	Entity
Device	Station
Installation_Data_Link	All entities
Security_Category	Station, program, and usercode

ENTY_DESGRES is the designator returned by COMS.

GET_DESIGNATOR_USING_NAME

You can use this service function to convert a COMS entity name to a COMS designator. If you have a name that represents an entity, you can get the designator associated with that name. For instance, you could use this service function if you want to send a message through an agenda for which you have the name. You cannot put the agenda name in the output header, but you can obtain the agenda designator by using this service function.

If a program requires the designator of the window under which it is operating, you can place an asterisk (*) in the first column of the parameter you pass for the window name and call GET_DESIGNATOR_USING_NAME. COMS will return the window designator for the window associated with the calling program.

If GET_DESIGNATOR_USING_NAME receives an input name of 17 blanks in a usercode inquiry, GET_DESIGNATOR_USING_NAME will return the superuser designator. For more detailed information on superuser usercode and superuser designator, refer to the *A Series Security Administration Guide.*

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_DESIGNATOR_USING_NAME(ENTY_NAME,
                                            ENTY_MNEMONIC,
                                            ENTY_DESIGNATOR);
```

Parameters

ENTY_NAME is the name of the entity. For the agenda, trancode, and installation data entities, the string for the entity name should include the window name if the program calling the service function is running in another window or is run outside of COMS. For example, when you pass the entity name, you should provide the following:

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

For the installation data entity, if a window is not specified, and if the window in which your program is running has no entity of the same name, the installation data with window value equal to “ALL” is picked up, since it is the default.

ENTY_MNEMONIC is the mnemonic for the entity you are obtaining. The mnemonics are listed in Appendix A. Mnemonics are allowed for all entities that have a designator.

ENTY_DESIGNATOR is the designator COMS returns to you.

GET_INTEGER_ARRAY_USING_DESIGNATOR

You can use this service function to get a list of integers from the structure represented by a designator.

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_INTEGER_ARRAY_USING_DESIGNATOR
                                                      (ENTY_DESIGNATOR,
                                                       ENTY_MNEMONIC,
                                                       ENTY_INTEGERTOTAL,
                                                       ENTY_INTEGERVECTOR);
```

Parameters

ENTY_DESIGNATOR represents the structure from which you want to get a list of integers.

ENTY_MNEMONIC describes which integer list you are requesting. The allowable mnemonic and designator combinations are as follows:

Mnemonic	Entity
Installation_Integer_All	All entities
Mixnumbers	Program

ENTY_INTEGERTOTAL is the number of integers returned in the vector.

ENTY_INTEGERVECTOR is the list itself.

GET_INTEGER_USING_DESIGNATOR

You can use this service function to get a specific integer out of the structure represented by a designator.

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_INTEGER_USING_DESIGNATOR(ENTY_DESIGNATOR,
                                               ENTY_MNEMONIC,
                                               ENTY_INTEGER);
```

Parameters

ENTY_DESIGNATOR is the designator representing the structure from which you want to get a specific integer.

ENTY_MNEMONIC describes which integer you are requesting. The allowable mnemonic and designator combinations are as follows:

Mnemonic	Entity
Aggregate_Response_Time	Program
Current_User_Count	Window
Installation_Integer_All	All entities
Installation_Integer_1	All entities
Installation_Integer_2	All entities
Installation_Integer_3	All entities
Installation_Integer_4	All entities
Last_Response_Time	Program
Logical_Station_Number	Station
Maximum_User_Count	Window
Mixnumbers	Program
Screen_Size	Station
Total_Transaction_Count	Program
Transaction_Queue_Depth	Program
Virtual_Terminal	Station

ENTY_INTEGER is the integer returned by COMS.

GET_NAME_USING_DESIGNATOR

You can use this service function to convert a COMS designator to the COMS name for that designator. For instance, you might want to know the name of the terminal that sent a message if you use the terminal names as a unique key.

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_NAME_USING_DESIGNATOR(ENTY_DESIGNATOR,
                                            ENTY_NAME);
```

If GET_NAME_USING_DESIGNATOR in a usercode inquiry receives the designator of the superuser, then GET_NAME_USING_DESIGNATOR returns a name of 17 blanks. For more detailed information on superuser, refer to the *Security Administration Guide.*

Parameters

ENTY_DESIGNATOR is the designator. All designators are allowed.

ENTY_NAME is a string of 1 to 17 characters, except for station names, which are up to 255 characters, where the name is to be returned by COMS. If the station name returned by COMS is shorter than the user array, COMS fills the user array with blanks.

GET_REAL_ARRAY

You can use this service function to get a data structure with no connection to any entity. You can monitor response times on your system through the use of statistics.

Note: *This service function cannot be used by the RPG programming language. However, the results provided by this service function can be covered by using the GET_INTEGER_USING_DESIGNATOR service function and performing a series of separate calls.*

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_REAL_ARRAY(ENTY_MNEMONIC,
                                 ENTY_REALTOTAL,
                                 ENTY_REALVECTOR);
```

Parameters

ENTY_MNEMONIC is the data structure that you are requesting. The allowable mnemonic is Statistics. It returns a table, each entry of which has six elements as follows:

- A designator for the entity for which the statistics are given
- The type of entity as follows:

Value	Entity
1	Direct-window program
2	Remote-file program
3	MCS window

- Transaction queue depth (direct window only)
- Total transaction count
- Last response time in milliseconds (direct window only)
- Aggregate response time in milliseconds (direct window only)

This information is also provided by the Statistics window. However, if you want to generate your own reports, the service function is available for your use. For more information on the Statistics window, see the *COMS Configuration Guide.*

ENTY_REALTOTAL is the total number of elements returned in the array – six times the number of table entries for statistics.

ENTY_REALVECTOR is the array where the information is returned from COMS. The size of the array is six words for each running direct-window program plus six words for each MCS window.

GET_STRING_USING_DESIGNATOR

You can use this service function to get an EBCDIC string out of the structure represented by the designator. This service function returns strings that you can have set up as installation data in the COMS Utility.

This service function is used to enable a multilingual program to communicate with multiple stations by determining the language and convention attributes of the stations. If an application program uses the GET_STRING_USING_DESIGNATOR service function to retrieve the attribute for a station, and if the station uses the default system attribute, then the service function returns an error value of 4 (UMBRELLA_NONDATA_ERRORV).

For more information on the language and convention attributes, refer to the *COMS Configuration Guide.*

The declaration in COMS is as follows:

```
INTEGER PROCEDURE GET_STRING_USING_DESIGNATOR(ENTY_DESIGNATOR,
                                              ENTY_MNEMONIC,
                                              ENTY_STRINGTOTAL,
                                              ENTY_STRING);
```

Parameters

ENTY_DESIGNATOR is the designator representing the structure from which you want to get the string.

ENTY_MNEMONIC describes which string you are requesting. The allowable mnemonic and designator combinations are as follows:

Mnemonic	**Entity**
CONVENTION	Station
Installation_String_1	All entities
Installation_String_2	All entities
Installation_String_3	All entities
Installation_String_4	All entities
Installation_Hex_1	All entities
Installation_Hex_2	All entities
LANGUAGE	Station

ENTY_STRINGTOTAL is the number of valid characters in the string returned by COMS.

ENTY_STRING is the string returned by COMS. This designator returns the requested LANGUAGE or CONVENTION attribute string.

If either the LANGUAGE or CONVENTION mnemonic is specified, the station designator obtains the corresponding station table index of the entry for that station.

After the station index has been determined, the station table entry is checked to determine if the station is logged on. Ensuring the log-on status requires checking that the STA_UCODEINX field is set to a nonzero value. If the STA_UCODEINX field is set to zero, the procedure returns a value indicating a failure. If the STA_UCODEINX field is set to a nonzero value, then the attribute string required is returned in ENTY_STRING. If no other attribute has been specified, the system default attribute is returned.

STATION_TABLE_ADD

This service function is used to add a station designator to the table. The table and the station designator are passed to the procedure and a unique index is returned.

Note: *This service function cannot be used by the RPG or Pascal programming languages.*

The declaration in COMS is as follows:

```
PROCEDURE STATION_TABLE_ADD (STATION_HASH,
                             STATION_DESIGNATOR);
```

Parameters

STATION_HASH is the table of station designators.

STATION_DESIGNATOR is the station designator to be added to the table.

STATION_TABLE_INITIALIZE

This service function is used to initialize the station table into which the station index values are placed. You pass to this procedure the table and a table modulus. (The table is implemented as a hash table.) The modulus is used to determine the density and access time of the table. For fast access and sparse table population, select a modulus that is twice the maximum number of table entries. For slow access and compact table population, select a modulus that is half the maximum number of entries.

Note: *This service function cannot be used by the RPG or Pascal programming languages.*

The declaration in COMS is as follows:

```
PROCEDURE STATION_TABLE_INITIALIZE (STATION_HASH,
                                    SHMOD);
```

Parameters

STATION_HASH is the table of station designators.

SHMOD is the table modulus.

STATION_TABLE_SEARCH

The STATION_TABLE_SEARCH service function is used to locate a service designator. This service function locates a service designator by receiving a station table and then searching the available service designators within the table. If the service function finds the service designator, the index to the table is returned. A return value of zero indicates that the station designator was not found.

Note: *This service function cannot be used by the RPG or Pascal programming languages.*

The declaration in COMS is as follows:

```
INTEGER PROCEDURE STATION_TABLE_SEARCH (STATION_HASH,
                                        STATION_DESIGNATOR);
```

Parameters

STATION_HASH is the table of station designators.

STATION_DESIGNATOR is the station designator to be located.

TEST_DESIGNATORS

You can use this service function to test whether a designator is part of a structure represented by another designator. For example, when you pass a security designator and a security-category designator to this service function, COMS tells you whether the security category represented by the security-category designator is valid for the session represented by the security designator. This service function can also be used for the security designators of stations and usercodes.

The declaration in COMS is as follows:

```
INTEGER PROCEDURE TEST_DESIGNATORS(ENTY_DESIGNATOR_1,
                                   ENTY_DESIGNATOR_2);
```

Parameters

ENTY_DESIGNATOR_1 and ENTY_DESIGNATOR_2 represent the two entities you are trying to match, or whose relationship you want to determine. The order in which you pass the designators does not matter. Allowable combinations of designators are as follows:

- Device-type list and device type
- Security category and security

Section 5
Processing Items

A processing item is a procedure you can use to process a message either before an application program receives it or after an application program sends it. Only programs that use the direct-window interface to COMS can receive and send messages that use processing items. Processing items reside in processing-item libraries you create. Processing items are written in ALGOL. However, any programming language that ALGOL can call can use the ALGOL shell to write processing items. For information on using the ALGOL shell to write processing items, see Appendix E.

To integrate processing items into your installation, you need to understand the functions and interrelationships of the entities in the COMS configuration file. For more information on COMS entities, see the Section 1, "Introduction to COMS." Also refer to the *COMS Configuration Guide* for more information about the COMS entities and what roles they play in message routing.

You can use processing items to preprocess and postprocess the messages that are received and sent by the stations and programs in a data communications network. To preprocess a message, the COMS configuration file must have been defined to apply a list of processing items to a message before it is received by a program. Refer to the *COMS Configuration Guide* for information on defining the configuration file. To postprocess a message, specify programmatically which agenda you want applied to the message a program is sending. Refer to Section 3, "Communicating with COMS through Direct Windows," for more information on the necessary programming steps.

The following are among the possible uses for processing items:

- Translating a message from one format into another
- Generating multiple messages to be received or sent from a single message
- Redirecting a message to a destination different from the one indicated by the system configuration or specified programmatically
- Segmenting long messages into several shorter messages, or grouping several short messages into one large message
- Performing security checks on messages
- Auditing messages
- Formatting input and output messages

For a processing item to process a message, the message must be received or sent through a COMS direct window and must be associated with an agenda. You can apply processing items only by using agendas, because programs you write cannot directly call processing items. Refer to Section 3, "Communicating with COMS through Direct Windows," for information about the direct-window interface to COMS and for the function of agendas in message routing, preprocessing, and postprocessing.

The next two parts of this section, "How Processing Items Alter Message Data" and "Creating a Processing-Item Library," provide a description of how processing items work and what structures are necessary before you can write processing items. For a discussion of how to write processing items, refer to "Creating a Processing Item Using the ALGOL Specification" in this section.

How Processing Items Alter Message Data

The actual preprocessing or postprocessing of a message occurs when the COMS Agenda Processor library passes the message data and the associated header to a processing-item library. The route a message takes to reach the Agenda Processor library differs depending on whether the message is to be preprocessed or postprocessed. The following subsections describe the routes that messages take to reach the Agenda Processor library and the role the library plays in message processing.

Routing of a Message for Preprocessing

When an incoming message first enters the COMS system, the COMS Router library determines which agenda to apply to the message. If the message contains a trancode, then the trancode specifies the agenda. If the message does not contain a trancode, then the Router library applies the default input agenda of the window to which the message is destined.

An agenda must specify the destination for the message and can specify a list of processing items to be applied to the message. According to the destination specified by the agenda, the Router library places the message in the queue of the destination program. When the destination program executes a RECEIVE statement, the transaction processor (TP) library or the database (DB) library is called automatically. Next, the TP or DB library determines whether any processing items need to be applied to the message in the queue. If processing items do need to be applied, the TP or DB library calls the COMS Agenda Processor library.

Refer to "How the Agenda Processor Handles a Message" later in this section to find out what actions the Agenda Processor library takes when it is called by the TP or DB library. The actions taken by the Agenda Processor library are the same regardless of whether a message is being preprocessed or postprocessed.

Routing of a Message for Postprocessing

When a program executes a SEND statement to send a message out to a station or another program, the TP or DB library is called automatically. Next, the TP or DB library determines what agenda is associated with the message. An agenda must have been specified either programmatically in the Agenda Designator field of the output header before the SEND was executed or by specification of a default output agenda for the window. If no agenda was specified, the TP or DB library applies the default agenda of the direct window that is sending the message.

Sending a program-to-station message is the simplest case of routing an outgoing message. In this case, if the specified agenda contains a processing-item list, the TP or

DB library calls the Agenda Processor library before sending the postprocessed message to its destination.

For a program-to-program message, the TP or DB library places the outgoing message into the queue of the destination program. When the destination program executes a RECEIVE statement, the TP or DB library calls the Agenda Processor library if the specified agenda contains a processing-item list.

If you route an outgoing message by trancode, you must have specified, either in the Agenda Designator field of the output header of the program or as the default output agenda for the window, an agenda with no processing-item list whose destination is INPUT_ROUTER. Next, the TP or DB library replicates the logic of the Router library to determine which agenda is associated with the trancode in the message. Once the agenda is known, the TP or DB library calls the Agenda Processor library if the agenda contains a processing-item list, before sending the message to its destination. Refer to "Routing Messages by Specifying a Destination" and "Routing Messages by Trancode" in the Section 3,"Communicating with COMS through Direct Windows," for more information about specifying INPUT_ROUTER as an agenda.

The following subsections describe what actions the Agenda Processor library takes when it is called by the TP or DB library. The actions taken by the Agenda Processor library are the same regardless of whether a message is being preprocessed or postprocessed.

How the Agenda Processor Library Handles a Message

Only the TP or DB library can call the Agenda Processor library in response to an incoming message entering COMS or a program executing a SEND statement. When the TP or DB library calls the Agenda Processor library, it passes to the Agenda Processor library the message data and associated header for the message being preprocessed or postprocessed.

If the specified agenda contains a processing-item list, the Agenda Processor library calls each processing item on behalf of the message, passing as parameters the message data and the associated header. Thus, the processing item can modify the memory areas, called the message data array and the header array, which are declared in the program that is receiving or sending a message.

When a single processing item completes all its processing tasks, it exits back to the Agenda Processor library. If the Agenda Processor library calls a second processing item, the second item sees the same memory areas as they were modified by the previous processing item. When all processing items in a list are completed, the Agenda Processor library exits back to the TP or DB library, which exits back to the program executing a RECEIVE statement or a SEND statement.

Example of Processing Items Used for Postprocessing

Following is an example of three processing items in a processing-item list that work together to postprocess a message:

- The first processing item fetches instructions from disk for creating a particular format.
- The second processing item uses the instructions to create the format and the message that the user receives.
- The third processing item stores the message on disk so it can be recalled if needed. When the third processing item completes its task, COMS automatically sends the format to the user's terminal.

Creating a Processing-Item Library

You must create a processing-item library before writing individual processing items. The following paragraphs discuss the conventions to follow when creating processing-item libraries and how to choose a library configuration.

Conventions for Creating Libraries

Follow these conventions when you create processing-item libraries for a multiple-program environment:

1. Write processing-item libraries in ALGOL only.
2. Use these ALGOL library attributes when writing the library code:

   ```
   $SET SHARING = SHAREDBYRUNUNIT
   ```

   ```
   FREEZE (PERMANENT)
   ```

   Refer to the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation* for information about writing ALGOL libraries.
3. One processing-item library cannot call another processing-item library that is known to COMS.
4. An application program cannot directly call a processing-item library. Only a COMS internal library called the Agenda Processor library can directly call a processing-item library.
5. A program written in a language other than ALGOL can make use of processing-item libraries if the language has a library interface. Although no program except the Agenda Processor library can directly call a processing-item library, a language must have a library interface in order to use the direct-window interface to COMS.

Choosing a Library Configuration

There are two basic configurations you can use when creating processing-item libraries:

- Multiple libraries that each contain only a few entry points
- A single library, or only a few libraries, that each contain numerous entry points

These configurations are described in the following text. Read this information before choosing a library configuration.

Creating Multiple Libraries Containing Few Entry Points

The advantages and disadvantages of creating multiple libraries that each contain only a few entry points are as follows:

- Multiple libraries are easy to maintain. If you include as few entry points as possible in each library, then changing, adding, or deleting libraries or entry points has little impact on other system structures.
- Multiple libraries require more memory than a single library that contains all combined entry points.

Creating a Single Library with Multiple Entry Points

The advantages and disadvantages of creating one library, or only a few libraries, containing multiple entry points are as follows:

- Using few libraries minimizes the number of memory stacks running on your system.
- If all functions that you want performed share a common state or common declarations, such as a common file declaration, then making all these functions entry points of the same library is efficient and might be mandatory in some cases.
- An entry point of a processing-item library can call other entry points or internal procedures of the same library. These entry points and procedures can share common logic. In contrast, an entry point of one processing-item library cannot share logic with or call an entry point of another processing-item library.

Example of a Single Library Containing Multiple Entry Points

Following is an example of a processing-item library with two entry points that can refresh a screen if for some reason the original screen is lost:

- One entry point is called right after the original screen is transmitted to the terminal of the user. This is a postprocessing entry point that saves a copy of the screen on disk.
- If the user's terminal must be turned off and then turned back on, the REFRESH command is entered.

- The second entry point of the processing-item library detects the entry of the REFRESH command, and retrieves the copy of the screen that is saved on disk. Then COMS automatically sends the copy of the screen to the user's terminal.

In this case, there are two entry points in one processing-item library because both entry points share a common declaration for the disk file.

Creating a Processing Item Using the ALGOL Specification

You should write your processing item in ALGOL. However, any programming language that ALGOL can call can use the ALGOL shell to write processing items. For information on using the ALGOL shell to write processing items, see Appendix E.

To create the processing item, use the following ALGOL specification:

```
REAL PROCEDURE PROC_ITEM(STATE,
                         HEADER,
                         USER_DATA,
                         TEXT_1,
                         TEXT_2,
                         OUTPUT_PROC);
      REAL STATE;
      ARRAY HEADER[Ø];
      EBCDIC ARRAY USER_DATA[Ø], TEXT_1[Ø],
                      TEXT_2[Ø];
      REAL PROCEDURE OUTPUT_PROC(STATE,
                                 HEADER,
                                 TEXT_1,
                                 TEXT_2);
           REAL STATE;
           ARRAY HEADER[Ø];
           EBCDIC ARRAY TEXT_1[Ø], TEXT_2[Ø];
           FORMAL;
      BEGIN
        .
        .
        .
      END;
```

Use this REAL procedure to declare the required parameters for each processing item you create. The required parameters are STATE, HEADER, USER_DATA, TEXT_1, TEXT_2, and OUTPUT_PROC. Although you can name the procedure and the parameters anything you wish, you must declare the parameters in the order shown above. The Agenda Processor library passes these parameters to your processing item when submitting a message for processing.

Whether or not you intend to use all the parameters shown in the specification, you must declare all of them, including the formal procedure called OUTPUT_PROC. Even if you want a processing item to do nothing but call OUTPUT_PROC, you must declare the entire PROC_ITEM procedure.

The Agenda Processor library passes each of the six parameters when calling a processing item so that the processing item can modify the message data. Each parameter is described in the following subsections.

STATE Parameter

The STATE parameter is a REAL variable. One of its purposes is to indicate which one of the following parameters holds the newest message data:

- USER_DATA
- TEXT_1
- TEXT_2

When the Agenda Processor library calls a processing item, it sets the value of STATE.[15:02] to tell the processing item where the message data is. The Agenda Processor library also sets STATE.[07:08] to a value of 0 (zero) or 1 to indicate whether the message is being received or sent. Finally, the Agenda Processor library sets STATE.[13:06] to the index of the first word of the Conversation Area field in the HEADER parameter.

A processing item must change the value of STATE.[15:02] to one of the values defined in Table 5–1 whenever it moves the newest message data from one data area to another. If it does not modify the message data at all, do not change the value of any field in the STATE parameter.

Table 5–1. State Parameter Values for Processing Items

Location	Value	Meaning
STATE.[07:08]	0	This message is received by an application program executing the RECEIVE <input header name> statement.
	1	This message is sent by an application program executing the SEND <output header name> statement.
STATE.[13:06]		This field contains the index of the first word of the input Conversation Area field in the HEADER parameter.

continued

Table 5–1. State Parameter Values for Processing Items (cont.)

Location	Value	Meaning
STATE.[15:02]	0	The USER_DATA parameter contains the newest message data. If a processing item is placing new message data into the USER_DATA parameter, then the processing item must set STATE.[15:02] to a value of 0 (zero) to inform the Agenda Processor library that that the newest message data are now in the USER_DATA parameter.
	1	The TEXT_1 parameter contains the newest message data. If a processing item places new message data into the TEXT_1 parameter, then the processing item must set STATE.[15:02] to a value of 1 to inform the Agenda Processor library that the newest message data are now in the TEXT_1 parameter.
	2	The TEXT_2 parameter contains the newest message data. If a processing item places new message data into the TEXT_2 parameter, then the processing item must set STATE.[15:02] to a value of 2 to inform the Agenda Processor library that the newest message date are now in the TEXT_2 parameter.
STATE.[23:08]		This field is reserved for use by COMS.
STATE.[47:24]		Processing items use this field to pass information to other processing items. COMS initializes this field to a value of 0 (zero) prior to calling the first processing item. Between calls to other processing items, COMS does not change the value of STATE.[47:24]. COMS preserves the changes made to the message data by one processing item until the next processing item is called.

HEADER Parameter

The HEADER parameter is an array designed to contain the header passed by the program on whose behalf the processing item has been called. When calling a processing item, the Agenda Processor library passes to the HEADER parameter a copy of the header in the program that is receiving or sending a message.

The Agenda Processor library passes a copy of the input header when the message has yet to be received by the program. The Agenda Processor library passes a copy of the output header when the message is being sent by the program. Refer to Section 3, "Communicating with COMS through Direct Windows," for more information about the the input and output headers.

A processing item can modify fields within the HEADER array so that the HEADER contains the correct descriptive information when the processed message is received or sent by a program.

Updating Input Header Fields

When the HEADER parameter is an input header, you must update the following fields of the input header if these conditions apply:

- The Text Length field of the input header if the processing item changes the length of the message data
- The Conversation Area field of the input header if you wish to pass additional information to a program or another processing item

Updating Output Header Fields

When the HEADER parameter is an output header, you must update the following fields of the output header if these conditions apply:

- The Text Length field of the output header if the processing item changes the length of the message data
- The Agenda Designator field of the output header if you wish to specify an agenda that differs from the agenda originally specified in output header of the application program
- The Destination field of the output header if you wish to specify a destination that differs from the destination originally specified in the output header of the application program
- The Conversation Area field of the output header if you wish to pass additional information to a program or another processing item

Refer to Section 3, "Communicating with COMS through Direct Windows," for descriptions of fields within the input and output headers. Refer to the definition of the OUTPUT_PROC parameter in this section for more information about specifying destinations for processed messages.

USER_DATA Parameter

The USER_DATA parameter is an EBCDIC array that can contain the message data that is to be preprocessed or postprocessed. When calling a processing item, the Agenda Processor library passes in the USER_DATA parameter a pointer to the message data in the destination program.

For a message that is to be postprocessed, do not modify the message data while it is in the USER_DATA parameter, because doing this would modify the message area of your program. Moreover, modifying the message data in USER_DATA could adversely affect subsequent sending of messages and reproducibility during a synchronized recovery. In addition, you might need to use the original data more than once to generate multiple messages for the same transaction.

Move the message data into the TEXT_1 and/or TEXT_2 parameters (described later in this section) when you want to modify it. Use the STATE parameter to inform the Agenda Processor library as to which parameter contains the newest message data at a particular time. When the value of STATE.[15:02] is 0 (zero), the USER_DATA parameter contains the newest message data.

TEXT_1 and TEXT_2 Parameters

The TEXT_1 and TEXT_2 parameters are EBCDIC arrays. You can use these parameters as scratch data areas for modifying the message data.

These parameters each have an initial size of one character. Each must be resized with the RESIZE verb prior to being used. A processing item should always check the size of the TEXT_1 or TEXT_2 parameter before placing message data into it, because the same arrays are reused for each copy of a program that is sending preprocessed messages.

If the processing item changes the length of the message data while it is in the TEXT_1 or TEXT_2 parameter, you must update the Text Length field in the appropriate header. Refer to the definition of the HEADER parameter earlier in this section for more information about updating fields in the header.

The processing item must change the value of STATE.[15:02] if the processing item edits or reformats the message data from the USER_DATA parameter to a scratch area, or from one scratch area to the other. Set STATE.[15:02] to one of the following values to let the Agenda Processor library know the new location of the newest message data:

- A value of 1 if the TEXT_1 parameter holds the newest message data
- A value of 2 if the TEXT_2 parameter holds the newest message data

OUTPUT_PROC Parameter

The OUTPUT_PROC parameter is a formal REAL procedure of the Agenda Processor library that must be declared in a processing item as follows:

```
REAL PROCEDURE OUTPUT_PROC(STATE,
                           HEADER,
                           TEXT_1,
                           TEXT_2);
     REAL STATE;
     ARRAY CD[Ø];
     EBCDIC ARRAY TEXT_1[Ø], TEXT_2[Ø];
     FORMAL;
```

When calling a processing item, the Agenda Processor library passes the name of the OUTPUT_PROC procedure as a parameter to the processing item. The primary purpose of OUTPUT_PROC is to generate multiple transactions based on a single message that the Agenda Processor library has passed to the processing item. A processing item must call OUTPUT_PROC multiple times to generate multiple transactions.

You can use OUTPUT_PROC to do the following:

- Redirect a message to a destination that differs from the destination specified in the header that the Agenda Processor library passed to the processing item. To do this, modify one of the fields in the header that can specify a destination.
- Transmit the segments of a segmented message to a single destination.
- Send a single message to multiple destinations by calling OUTPUT_PROC once for each destination. Specify the destinations by creating a header for each destination.
- Generate multiple messages and send them to a single destination by calling OUTPUT_PROC once for each message.
- Generate multiple messages and send them to multiple destinations by calling OUTPUT_PROC once for each message and creating a header to specify each destination.

Refer to "Passing an Input Header to OUTPUT_PROC" and "Passing an Output Header to OUTPUT_PROC" later in this section for more information about generating new transactions by calling OUTPUT_PROC multiple times. Having a program execute the SEND statement multiple times is an alternative to having a processing item call OUTPUT_PROC multiple times.

Calling OUTPUT_PROC and Transmitting a Message

When a processing-item list contains several processing items, you usually wait until the last item in the list has completed its tasks before transmitting the completely processed message to its destination. There are two ways in which a processing item can transmit a message to its destination:

- Call OUTPUT_PROC explicitly, passing the parameters described later in "Passing the Parameters to OUTPUT_PROC."
- Allow the Agenda Processor library to transmit the message automatically when the last processing item in a processing-item list completes its tasks.

It is unnecessary to explicitly call OUTPUT_PROC if you are transmitting a single message to a single destination, because the Agenda Processor library can do it automatically. This leaves you free to add, delete, or change the order of the items in a processing-item list and be assured that the message will always be transmitted at the end of the list.

Whether a processing item calls OUTPUT_PROC explicitly or allows the Agenda Processor library to transmit the message automatically, the message goes to one of the following destinations:

- The originating station that is specified in the Station field of the input header, if the processing item contains an input header in its HEADER parameter. This is the most common destination.
- The station or program that is specified in the Destination field of the output header, if the processing item contains an output header in its HEADER parameter.
- The destination associated with the agenda that is specified in the Agenda Designator field of the output header, if the processing item contains an output header in its HEADER parameter.

Passing the Parameters to OUTPUT_PROC

A processing item must pass the following parameters to OUTPUT_PROC:

- STATE of type REAL
- HEADER of type Array
- TEXT_1 of type EBCDIC Array
- TEXT_2 of type EBCDIC Array

The types and semantics of these parameters are the same as for the STATE, HEADER, TEXT_1, and TEXT_2 parameters of the PROC_ITEM procedure. You can pass to OUTPUT_PROC the same parameters that Agenda Processor library passes to the processing item, but you do not have to pass the same parameters. The processing item must make all desired modifications to the message data and the input or output header before calling OUTPUT_PROC.

You pass either an input header or an output header to the OUTPUT_PROC procedure, depending on whether the message being processed is to be received or sent by an application program. You can use OUTPUT_PROC to generate new transactions with either an input header or an output header, although different conventions apply, depending on which header you are passing.

Passing an Input Header to OUTPUT_PROC

When a processing item passes an input header to OUTPUT_PROC, the message can be transmitted to a program but not to a station. A processing item should perform the following tasks before calling OUTPUT_PROC:

- Set STATE.[07:08] to a value of 0 (zero).
- Set STATE.[13:06] to the index of the first word of the input Conversation Area field in the header being passed.
- Set STATE.[15:02] to a value of either 1 or 2 to indicate which parameter contains the newest message data.

To allow your program to specify or change a destination when passing an input header, choose one of the following methods:

- Place an agenda designator in the Agenda Designator field of the input header.
- Place a program designator in the Program Designator field of the input header.
- Place both an agenda designator and a program designator in the previously mentioned fields of the input header. With this method, the program designator specifies the destination program, while the agenda designator specifies the processing-item list.

Do not specify an agenda whose destination is INPUT_ROUTER when you want to send a message to another program. You must use one of the methods previously described to specify the destination program.

Passing an Output Header to OUTPUT_PROC

When a processing item passes an output header to OUTPUT_PROC, the message can be transmitted to a station or a program. A processing item should perform the following tasks before calling OUTPUT_PROC:

1. Set STATE.[07:08] to a value of 1.
2. Set STATE.[13:06] to the index of the first word of the output Conversation Area field in the header being passed.
3. Set STATE.[15:02] to a value of 0 (zero), 1, or 2 to indicate which parameter contains the newest message data.

To specify or change a destination when passing an output header, choose one of the following methods:

- Place an agenda designator in the Agenda Designator field of the output header.
- Place a program designator in the Destination field of the output header.
- Place both an agenda designator and a program designator in the Agenda Designator and Destination fields of the output header. Using this method, the program designator specifies the destination program, while the agenda designator specifies the processing-item list.

If you do not specify a destination in the output header, the Agenda Processor library transmits the message to the station or program that originated the current transaction. COMS derives the identity of the originator from the input header associated with the output header that was passed to the Agenda Processor library.

Caution

When you use a processing item to call OUTPUT_PROC, do not specify a default output agenda in the output header of the processing item. This procedure can cause the processing item to produce recursive calls that eventually lead to a stack overflow.

Formatting Output Messages

You can create a processing item that formats or reformats output messages in any way you desire before they reach their destinations. However, COMS provides a simple, predefined method for specifying or altering carriage control of output messages.

Altering Carriage Control for Output Messages

A processing item can specify or alter carriage control for an output message before the message reaches its destination, regardless of whether the program or station that sent the message has specified carriage control.

A direct-window program can specify carriage control or allow the COMS default setting to be used. A COBOL74 application program can use the BEFORE/AFTER ADVANCING option with the SEND statement to specify carriage control, or allow the COMS default setting (advancing after one line) to be used.

Carriage Control Field Values

A processing item can reset bits [47:16] in the carriage-control field of the output header of a direct-window program when the output header is passed to the processing item in the HEADER parameter of the PROC_ITEM procedure.

Table 5–2 describes the possible values you will find in bits [47:16] of the FIELDS word, after the direct-window program sends a message and the processing item is called. Change the values in bits [47:16] according to your processing needs. The default for carriage control is 0 (zero).

Table 5–2. Carriage Control Field Values

Bit	Value	Meaning
[32:1]	1	No line advance.
	0	A line is advanced before or after the message text is written to the output device, depending on the value of bit [38:1].
[33:1]	1	No carriage return is done.
	0	A carriage return is done before or after the message text is written to the output device, depending on the value of bit [38:1].
[34:1]	1	A new page is required for the output device.
[37:3]		Bits [37:3] select an action requiring the use of bits [47:8] as a parameter. All the actions indicated by bits [37:3] are taken before or after sending the message, according to the value of bit [38:1].
	0	No action is required that uses the value in [47:8] as a parameter.
	1	Bits [47:8] contain a line number. The cursor is set to column 1 of this line before or after the message is sent, depending on the value of bit [38:1].
	2	Bits [47:8] contain the number of lines to be advanced, in addition to the number of lines advanced by bit [32:1]. The lines are advanced before or after the message is sent, depending on the value of bit [38:1].
[38:1]	1	All interpretation of the carriage control fields should be done before the message is sent.
	0	All interpretation of the carriage control fields should be done after the message is sent.

continued

Table 5–2. Carriage Control Field Values (cont.)

Bit	Value	Meaning
[39:1]	0	This bit is currently unused, and must retain the value of 0 (zero) set by COMS. (The bit might be used by COMS in future releases.)
[47:8]	n	Contains the number of lines to be advanced. Use of this field depends on bit [37:3].

Providing for Processing-Item Results

Prior to the END statement that completes the tasks performed by your processing item, include a statement for returning the REAL result of your processing-item procedure. The processing item must provide such a result as an instruction to the Agenda Processor library regarding the disposition of items in a processing-item list.

If the processing item does not provide a result, a default value of 0 (zero) is passed to the Agenda Processor library. If the processing item calls the OUTPUT_PROC procedure, then OUTPUT_PROC could call other processing items. When the last processing item called by OUTPUT_PROC completes processing, OUTPUT_PROC returns the result of the last processing item to the Agenda Processor library. This result is one of the four REAL values described in the following table.

Table 5–3 contains definitions of the REAL values that a processing item can return in the result word.

Table 5–3. Result Word REAL Values

Location	Value	Meaning
PROC_ITEM.[07:08]	0	Continue to process other items in the processing-item list.
	1	Stop processing because there are no more items in the processing-item list. For a message sent by an application program, a value of 1 means that the processing item prematurely stopped its processing tasks. In this case, COMS assigns a value of 96 to the Status Value field in the output header of the program and returns the newest message date to the Agenda Processor library.

continued

Table 5–3. Result Word REAL Values (cont.)

Location	Value	Meaning
	2	Stop processing and return the newest message data to the station that originated it. For a message yet to be received by an application program, a value of 2 means that COMS is transmitting the newest message data to the originating station. For a message sent by an application program, a value of 2 means that the processing item prematurely stopped its processing tasks. In this case, COMS assigns a value of 96 to the Status Value field in the output header of the program. By examining the conversation area, you might be able to determine what caused the processing tasks to stop prematurely.
	3	COMS returns this value only when a processing item has explicitly called the OUTPUT_PROC procedure, and OUTPUT_PROC has called a processing-item list. If a processing item is missing from the list called by OUTPUT_PROC, then COMS returns a value of 3 to the processing item that originally called OUTPUT_PROC.
PROC_ITEM.[47:39]		This field is reserved for use by COMS.

Section 6
Interactive Recovery

Interactive recovery applies to application programs that update Data Management System II (DMSII) databases and Semantic Information Manager (SIM) databases through direct windows. Recovery through COMS is not supported in the Pascal programming language.

Components of COMS Recovery

COMS recovery includes the following components:

- Protected input queues
- Two-phase transactions
- Concurrency

Protected Input Queues

Protected input queues assure that transactions waiting to be processed by application programs are not lost in a halt/load. These input transactions are audited to disk when they are received by COMS. All transactions that are not protected are lost in a halt/load.

The protected input specification is made in the COMS Utility on an agenda-by-agenda basis. For further information on input queue protection, see the *COMS Configuration Guide.*

Two-Phase Transactions

Two-phase transactions consist of two phases. In the first phase, all resources are locked (no records are freed). In the second phase, all resources are freed by the END-TRANSACTION statement.

All transactions processed by application programs must be two-phase transactions to guarantee reproducibility. For further details on two-phase transactions, see “Writing Two-Phase Transactions” later in this section.

Concurrency

The database attribute “concurrency” implies that in the event of an abort or halt/load recovery, all transaction states that have been completed are still reflected in the database. No completed transaction states are backed out of the database. Concurrency is a feature of a database and does not require intervention by a message control system (MCS) to reapply transactions.

DMSII databases can run with or without concurrency. Concurrency is achieved in DMSII by the Data and Structure Definition Language (DASDL) options INDEPENDENTTRANS and REAPPLYCOMPLETED. The INDEPENDENTTRANS option assures that all transactions processed against the database are two-phase transactions. The REAPPLYCOMPLETED option assures that all transactions that have completed transaction state are not backed out of a database after an abort or a halt/load.

All transactions processed against DMSII databases that have the INDEPENDENTTRANS option set are by default two-phase transactions. That is, the DMSII database software automatically converts a non-two-phase transaction into a two-phase transaction with no change to the application programs.

All transactions processed against DMSII databases that do not have the INDEPENDENTTRANS option set must be two-phase transactions. If you do not use the INDEPENDENTTRANS and REAPPLYCOMPLETED options, the DMSII database does not have concurrency control.

For more information on the INDEPENDENTTRANS and REAPPLYCOMPLETED options, see the *A Series DMSII Application Program Interfaces Programming Guide.*

SIM databases always run with concurrency control. This is an integral part of the software. Because of the presence of concurrency control in SIM, all transactions processed against a SIM database are by default two-phase transactions. For further information on SIM, see the *InfoExec SIM Technical Overview.*

Preparing to Use Interactive Recovery

To prepare you for using interactive recovery, the following information is provided:

- General conventions to follow when writing programs that update a database by means of two-phase transactions
- COMS actions when a program fails
- Requirements for a transaction-processing system that updates a database
- An overview of the COMS components that facilitate COMS recovery, and an explanation of how recovery works

General Programmatic Conventions

In writing programs that run under COMS and use interactive recovery, you must use the specific programmatic conventions explained in this section. Moreover, you need to observe the following general programmatic conventions to ensure effective recovery and perfect reproducibility:

- You must group all instructions together into a transaction, so that the program enters transaction state using a BEGIN-TRANSACTION statement, performs the update activity while in transaction state, and then exits transaction state using an END-TRANSACTION statement.

  ### Caution

  Avoid using SEND and RECEIVE statements during transaction state (between a BEGIN-TRANSACTION statement and an END-TRANSACTION statement) or you might lose some of the data in your database.

  All transactions that are to be included in interactive recovery, such as the COMS header name included in DMS BEGINTRANSACTION and ENDTRANSACTION statements, must occur after a RECEIVE statement has been executed.

- For databases not using the DASDL option INDEPENDENTTRANS, each transaction must be a two-phase transaction. In the first phase, the transaction should lock records but not free any. In the second phase, the transaction should free records but not lock any. In databases with concurrency control, all transactions are by default two-phase transactions.

- For proper recovery of messages after a database abort or program abort, the program must first execute a RECEIVE statement to handle the recovery transactions and then execute a SEND statement. If a SEND statement is executed before a RECEIVE statement, the transactions are not resubmitted.

Updating by Using Transactions

All application programs that update an audited database must be restartable, that is, able to resume processing where COMS directs them to after an interruption such as an abnormal program termination or a system halt/load.

For databases without concurrency control, DMSII ABORT/RECOVERY ensures that an interruption in processing does not leave the database with partially completed transactions. After DMSII ABORT/RECOVERY completes, COMS retrieves information from the DMSII restart data set that points to the last transaction completed by DMSII. Because all transactions completed by COMS are recorded in the COMS transaction trail, COMS can resubmit all the transactions that followed the last DMSII-completed transaction. COMS resubmits the transactions to the appropriate programs in the appropriate order. When the last transaction recorded in the transaction trail is successfully reprocessed, COMS recovery is complete. Databases with concurrency control do not encounter aborts. Next, COMS recommences to submit live transactions to the database.

To perform an update to the database, an application program must place the database into the condition called "transaction state." Transaction state refers to the time during which all the updates required for a single transaction are performed. In an application program, the BEGIN-TRANSACTION and END-TRANSACTION statements must delimit the set of updates to the database that logically compose one transaction. DMSII guarantees that either all or none of these updates will appear in the database as the result of a database abort.

When you write update programs for audited databases, think of your updates in terms of transactions rather than as arbitrary changes to the database. A transaction is a series of changes to the database that constitutes an indivisible, logical change. Write each transaction as a group of one or more data-set updates that are performed in one transaction-state cycle, causing the program to take the following steps:

1. Receive a message.
2. Make all preparations for the update.
3. Enter transaction state.
4. If the program is using a DMSII-oriented application, perform the update activity, which can assign, delete, generate, insert, remove, or store information. If the program is using SIM-oriented applications, perform the update activity, which can assign, delete, exclude, include, insert, modify, select, or retrieve information.
5. End transaction state in one of the following ways:
   - Without text. Then send the result to the originator of the transaction.
   - With text that includes an implicit SEND statement.
6. Return to step 1 (receive another message).

Your program should perform this sequence once for each update transaction that is to be applied to the database. The particular steps the program takes before entering transaction state vary, depending upon the application. In applications without concurrency, these steps consist of locking or creating records and changing data-item values in the work area of the program.

Every possible action that your program can do in preparation for the update should be done prior to entering transaction state. For applications without concurrency, only the storing, deleting and other record-update functions must be done during transaction state. For all applications, once transaction state is entered, it must not be exited until all updates associated with the transaction are performed. Transaction state should be entered and exited only once per transaction.

If a program tries to use any of the six update statements when the program is not in transaction state, the program is discontinued with an "INVALID OP" error. If a program does not follow the sequence of steps listed in this subsection, COMS reports an error message to the monitor station.

Writing Two-Phase Transactions

Every database includes a set of assertions, or consistency constraints, that data within the database must satisfy. When the data preserve all the constraints, the database is said to be consistent; otherwise, the database is said to be inconsistent. To ensure that consistency can be achieved in a multiple-program environment, DMSII provides a tool known as record-level locking, which allows a process to update a record only after retrieving the record with an exclusive lock. Although other processes can concurrently retrieve the record, record-level locking protects the record from updating by other processes until after it is explicitly or implicitly freed.

In order for record-level locking to preserve consistency, a programmatic convention known as the two-phase transaction must be observed. A transaction is two-phase if it can be divided into a locking phase followed by an updating phase. During the locking phase, the transaction locks records. During the updating phase, the transaction updates records but does not free or lock any. Records are freed automatically at the end of the updating phase.

After the last record is locked and until the first record is updated, the transaction is at the mid-transaction point. If two-phase transactions are not used, reproducibility of previous results and continued consistency of the database cannot be guaranteed when recovery operations complete.

COMS Actions When a Program Fails

COMS can take a variety of different actions when a program fails during execution. The following explanations show how COMS responds as it encounters different statements. The first two lists show the order in which COMS encounters the statements for an explicit SEND and an implicit SEND. Following those lists are Tables 6–1 and 6–2 that show the points at which COMS can fail and the corresponding actions that occur.

Order of Statements for an Explicit SEND

```
RECEIVE

BEGIN-TRANSACTION with HEADER

END-TRANSACTION

SEND
```

Order of Statements for an Implicit SEND

```
RECEIVE

BEGIN-TRANSACTION with HEADER

END-TRANSACTION with Text:  SEND

END TRANSACTION
```

Table 6–1. COMS Actions When a Program with an Explicit SEND Fails

Point in Program Failure	Action
Before a RECEIVE statement starts.	No action. Start program when next transaction is to be delivered.
After a RECEIVE statement starts, but before a BEGIN-TRANSACTION with HEADER statement starts.	Start program and redeliver transaction with a status 93 message.
After a BEGIN-TRANSACTION with HEADER statement starts, but before the END-TRANSACTION statement starts.	Start program and redeliver transaction with a status 93 message.
After an END-TRANSACTION with HEADER statement starts, but before the SEND statement starts.	No action. Start program when next transaction is to be delivered. (This means that output might be lost because the SEND is never done.)
After a SEND statement starts.	No action. Start program when the next transaction is to be delivered.

Table 6–2. COMS Actions When a Program with an Implicit SEND Fails

Point in Program Failure	Action
Before a RECEIVE statement starts.	No action. Start program when next transaction is to be delivered.
After a RECEIVE statement starts, but before a BEGIN-TRANSACTION with HEADER statement starts.	Start program and redeliver transaction with a status 93 message.
After a BEGIN-TRANSACTION with HEADER statement starts, but before an END-TRANSACTION with Text: SEND statement starts.	Start program and redeliver transaction with a status 93 message.

continued

Table 6–2. COMS Actions When a Program with an Implicit SEND Fails (cont.)

Point in Program Failure	Action
After an END-TRANSACTION with Text: SEND statement starts, but before an END-TRANSACTION statement starts.	Start program and redeliver transaction with a status 93 message. (This means that output might get delivered twice. The duplicate can be handled by a processing item.)
After an END-TRANSACTION statement starts.	No action. Start program when next transaction is to be delivered.

Note: *The ABORT-TRANSACTION statement causes the current transaction state to be exited without any updates. COMS does not resubmit this transaction.*

Requirements for Using Interactive Recovery

To run under COMS and use interactive recovery, a database-processing system must include the following:

- One or more application programs that process transactions
- One DMSII database or one SIM database
- If a DMSII database is used, one restart data set within the DMSII database that identifies restart records belonging to COMS

Possible configurations for data-base-processing systems that use interactive recovery include these:

- One application program that updates one SIM database or one DMSII database that is synchronized with COMS
- Multiple application programs that update one SIM database or one DMSII database that is synchronized with COMS

To use synchronized recovery, you must create a restart data set that stores the data needed for recovery. The role played by the restart data set, and instructions for creating and using it, are given later in this section.

Note that COMS does not support modeled databases. For instance, if you have one database (DB1), and then develop a new database (DB2) modeled after DB1, COMS stores the restart records of DB2 in the restart data set of DB1. This mixing of records prevents you from using synchronized recovery.

For databases without concurrency control, a DMSII database is synchronized with COMS when DMSII and COMS do the following:

1. DMSII recovery restores the database to the last point in time when no programs were in transaction state.
2. COMS resubmits all committed transactions to their respective application program beyond the DMSII recovery point, in the order that they were originally processed by multiple programs running asynchronously.

COMS Components That Facilitate Recovery

For each database that uses recovery, a COMS internal process called COMS Control initiates a separate task called the database (DB) control program. Each DB control program initiates a separate DB library. The DB library serves as the data communications interface (DCI) library for programs that are controlled by a common DB control program. Therefore, the DB control program and the DB library work together for each database to support the programmatic interface to COMS and recovery operations.

The transaction processor (TP) library is the DCI library that handles the COMS interface to application programs that do not need synchronized recovery. The TP library is initiated by COMS Control and does for these application programs what the DB control program and the DB library do together for application programs that need synchronized recovery.

Programs associated with the TP library can interface to one or more databases. However, COMS does not participate in recovery for these programs.

How DB Control and the DB Library Work

The DB control program initiates all application programs that use recovery with a particular database. DB control can detect transaction-state aborts that occur while your application programs are attempting to update the database. If the DB control program does detect an abort, it initiates a recovery cycle and, if necessary, restarts each application program.

In updating the database, application programs receive incoming messages and send outgoing messages by calling an entry point known as DCI. This is an entry point of the DB library that is associated with a particular DB control program. If a database recovery is in progress, the program receives a recovery transaction sent by the DB library.

If a program attempts to call a DCI_ENTRYPOINT associated with a control stack other than its own, the program is discontinued by COMS and an error message is displayed.

How the Restart Data Set and Transaction Trail Work

Every DB library has its own transaction trail, which is shared by all the application programs assigned to the particular DB library. A transaction trail is a time-ordered, logical audit trail that resides on disk and provides the data for reprocessing database transactions in case of a transaction-state abort, system crash, or rollback. A transaction trail actually consists of a series of files numbered 1 through N, with 1 being the oldest file and N being the newest file.

When a transaction-trail file is full, COMS closes the file and opens another one, incrementing the file number by 1. Alternatively, an operator can use the COMS *DATABASE <database name> TRAIL CLOSE* command to close the current transaction trail for a given database and open a new one, whose number is incremented by 1. COMS automatically causes a database SYNC point when a transaction trail is closed.

For the multiple-program environment, where preserving the order in which update transactions occur is essential to full recovery, COMS provides an efficient way to store and use data for reprocessing the transactions in the correct order. COMS stores restart information in the transaction trail based on the order of occurrence of the END-TRANSACTION statement.

Each transaction-trail record contains a copy of the COMS header and message area as they appeared in a particular program at the moment that the END-TRANSACTION statement was executed for a particular transaction.

Caution

Changing the current transaction trail file number on the Database Activity menu of the COMS Utility might make recovery of records in existing files impossible. Refer to the *COMS Configuration Guide* for further information.

The remainder of this section presents the following programming information for DMSII databases with and without concurrency and for SIM databases, which always include concurrency:

- Interactive recovery with DMSII databases
- Interactive recovery with SIM databases

Interactive Recovery with DMSII Databases

Interactive recovery with DMSII databases can be used either with or without concurrency. Interactive recovery without concurrency is referred to as synchronized recovery. Synchronized recovery is a COMS function that resubmits transactions to the database application program after a transaction-state abort, system crash, or rollback. It is called synchronized recovery because it reprocesses transactions in the same order that they were originally processed by multiple programs running asynchronously. The recovery process is slightly different, depending on the kind of interruption that has occurred.

If a transaction state abort occurs, first, DMSII recovery restores the database to the last point in time when no programs were in transaction state. Next, COMS resubmits all transactions already committed to the database that occurred beyond the DMSII recovery point. Last, COMS resubmits all in-process transactions, that is, those transactions that have reached the system but have not yet reached the database. If a system crash or rollback occurs, the first two steps are the same as in the case of a transaction-state abort, but in the third step COMS resubmits all transactions protected by input queue protection.

For information on input queue protection, see the *COMS Configuration Guide.* For information on archive operations, see the *COMS Operations Guide.*

To use interactive synchronized recovery, your direct-window program must include specific synchronized recovery code, as well as your usual message processing code. This section provides general guidelines and specific instructions for using synchronized recovery.

In databases that have concurrency control set, once a transaction has been committed to the database, that is, it is past the END-TRANSACTION statement, it is not backed out of the database. For example, if three programs are currently in transaction state and one of the programs terminates in transaction state, the other two transaction states are allowed to complete. The two completed transaction states are not backed out of the database.

This does not mean that these application programs do not need COMS recovery. Three recovery situations that require COMS/database synchronization are

- Reprocessing an aborted transaction
- Processing messages held in an input queue of the program
- Archival recovery

Reprocessing an Aborted Transaction

An aborted transaction is a transaction that has not completed successfully, due to program termination. COMS resubmits this transaction to the application program with an error in the Status field of the input header. For information on appropriate error messages, refer to Appendix A.

Processing Messages Held in the Input Queue of the Program

After a halt/load and the database has recovered, COMS submits all transactions in the application queues. Only those transactions in protected input queues are submitted. For information on input queue protection, see the *COMS Configuration Guide.*

Archival Recovery

Archival recovery is the reprocessing of transactions from the transaction trail, typically done after a database has been repositioned through a DMSII rollback. COMS resubmits the transactions to the application in the order the transactions were originally processed against the database.

Refer to “Creating a Restart Data Set” in this section for information about the fields in the restart data set that you need to create. Refer to Section 3, “Communicating with COMS through Direct Windows,” for information about the COMS input and output headers.

Writing Interactive Recovery Programs Using DMSII

This subsection provides the programming code for using recovery with DMSII databases. A sample of the program flow is presented, highlighting the six subroutines necessary for synchronized recovery. There are two possible main loops in the interactive recovery program, one with an explicit SEND statement and the other with a SEND statement built into the END-TRANSACTION subroutine. Also, note that the subroutines INITIALIZE_COMS, BEGIN-TRANSACTION, END-TRANSACTION, and EXIT_COMS are different, depending on the release and features you are using. To use the features in the current release of DMSII, you need to use the current release of COMS. Programs using concurrency control will not encounter aborts. Programs without concurrency control might receive resubmitted END-TRANSACTION subroutines.

Creating a Restart Data Set

In addition to the direct-window program containing synchronized recovery coding, you must create a restart data set. When the database administrator defines a database that your programs update, he or she must create a restart data set containing three fields that are associated with three specific attributes.

Following is an example of the Data and Structure Definition Language (DASDL) description for a restart data set that must be created for each DMSII database updated by programs that run under COMS and need synchronized recovery:

```
RESTART-DS RESTART DATA SET
  (
   RDS-ID       ALPHA(6) COMS-ID;
   RDS-PROG     REAL COMS-PROGRAM;
   RDS-LOCATOR  REAL COMS-LOCATOR;
  );
```

These fields of the restart data set are used for the recovery of direct-window programs and are maintained by COMS. COMS sets the value of RDS_ID to ONLINE for all the transactions generated by direct-window programs. In programs that are not COMS programs, such as batch programs, you can use these fields as long as you do not set the value of RDS_ID to the value ONLINE.

You are free to choose any valid DASDL names for the restart data set and the three fields in it, but you must do the following:

1. Be sure that the names in DASDL are the same as the names in your initialization routine.
2. Use the data types indicated in the example to define each field.
3. Identify each field directly to the DASDL compiler and indirectly to COMS by including these attributes with exactly these spellings:
   - COMS-ID
   - COMS-PROGRAM
   - COMS-LOCATOR
4. If you have a spanning set on the restart data set, you must also allow duplicates and specify an initial value for the implicit keys in DASDL.

COMS must create and store a master recovery record in the restart data set of any database updated by a COMS transaction processor. If the restart data set contains any required items or any other special requirements that COMS cannot satisfy, the CREATE and STORE procedures of the master recovery record will fail. This failure also can produce negative results on subsequent synchronized recoveries. To prevent these potential results, define any specially required items with INITIALVALUE clauses in DASDL.

Using Exception-Condition Statements and DMTERMINATE

Using exception-condition statements in your steps for closing a database is not recommended, because your program should terminate abnormally if a database error is detected during the database close. If your program does not terminate abnormally under these circumstances, recursive aborts of the database could occur. If you must use exception-condition statements in your steps for database close, then your program should also call DMTERMINATE for those exceptions not specifically handled by your program.

DMTERMINATE is a system-level DMSII procedure that you can invoke at any time to display a standard, recognizable error message and to discontinue the application program. For information on the syntax in your programming language to call DMTERMINATE, see the appropriate language manual.

Usually, DMTERMINATE should be the last procedure your program calls after it checks all other exception conditions that it specifically handles. The DMTERMINATE procedure returns the same values and results that would occur if the program had not intercepted the error in the first place—it displays a standard system error message and terminates the program.

DMTERMINATE is not intended as a method of handling common errors such as NOTFOUND. It is provided as a way out for programs that encounter unexpected errors, such as system errors or I/O errors.

As a general rule, an application program should go to end of task (EOT) once the database that is to be synchronized with COMS is closed.

Exceptions on the close of a DMSII database with concurrency control do not affect any of the completed transaction states of the interactive application program. By default, once a transaction has been committed to the database, it will not be backed out due to an abort or a halt/load recovery.

For DMSII databases with concurrency, the ABORT-TRANSACTION statement can be used to discontinue the current transaction state. If this statement is executed by an application program, the current transaction will not be resubmitted by COMS.

For further information about the ABORT-TRANSACTION statement, refer to the *DMSII Utilities Operations Guide.*

Program Flow for Recovery Programs Using DMSII

Following is an example of the program flow for the declaration and main loop of the interactive recovery program using DMSII databases, either with or without concurrency control. The program is presented in pseudolanguage, which uses indentation to indicate the scope of each statement.

Declaration and Main Program Loop

```
  ********************
  * DECLARATION PART *
  ********************
% database must contain restart data set (RDS)  %
% with the ID, PROG, and LOC fields.             %

database DATABASE;
COMS HEADER CDIN;
COMS HEADER CDOUT;
DATA AREA MSG;

  *************
  * MAIN LOOP *
  *************

     INITIALIZE_COMS;
 % main loop with explicit send %
     WHILE CDIN.STATUS NOT = 99 DO
          RECEIVE CDIN INTO MSG;
          IF CDIN.STATUS NOT = 99 THEN
               PREPARE_MSG;     % your message-processing code %
               BEGIN-TRANSACTION CDIN NO-AUDIT;
               UPDATE_DATABASE;
               END-TRANSACTION CDOUT AUDIT RDS;
               SEND CDOUT FROM MSG;
 % or main loop with send built into end transaction %
     WHILE CDIN.STATUS NOT = 99 DO
          RECEIVE CDIN INTO MSG;
          IF CDIN.STATUS NOT = 99 THEN
               PREPARE_MSG;     % your message-processing code %
               BEGIN-TRANSACTION CDIN NO-AUDIT;
               UPDATE_DATABASE;
               END-TRANSACTION CDOUT USING MSG AUDIT RDS;
     EXIT_COMS;
```

Subroutines Using COMS and DMSII (No RDS STORE)

If you have the current release of COMS and of DMSII and do not use the DASDL option RDS STORE, use the following subroutines for the main loop of your interactive recovery program:

```
  *******************
  * INITIALIZE_COMS *
  *******************

     OPEN UPDATE DATABASE
          ON EXCEPTION
               DMTERMINATE;
     ESTABLISH COMS LINK;
     ENABLE INPUT COMS "ONLINE"
       CREATE RDS;                        % required
       MOVE "ONLINE" TO RDS.ID;           % required
       MOVE CDIN.PROGRAMDESG TO RDS.PROG; % required
     % MOVE CDIN.RESTARTLOC TO RDS.LOC;   % no longer required

  ************************************************
  * BEGIN TRANSACTION TIME (without concurrency) *
  ************************************************

BEGIN-TRANSACTION CDIN NO-AUDIT
     ON EXCEPTION
          IF ABORT THEN           % not with INDEPENDENTTRANS option
               GO_RECEIVE_TRANSACTION
          ELSE IF AUDITERROR THEN % program logic error
               DMTERMINATE        % already in transaction state
          ELSE IF DEADLOCK THEN
               GO_RECEIVE_TRANSACTION
          ELSE DMTERMINATE;       % should not happen

  *********************************************
  * BEGIN TRANSACTION TIME (with concurrency) *
  *********************************************

BEGIN-TRANSACTION CDIN NO-AUDIT RDS
     ON EXCEPTION
          IF AUDITERROR THEN      % program logic error
               DMTERMINATE        % already in transaction state
          ELSE IF DEADLOCK THEN
               GO_RECEIVE_TRANSACTION
          ELSE DMTERMINATE;       % should not happen
  **********************************************
  * END TRANSACTION TIME (without concurrency) *
  **********************************************

END-TRANSACTION CDOUT AUDIT RDS
     ON EXCEPTION
          IF ABORT OR DEADLOCK THEN % no abort with INDEPENDENTTRANS
                                                              option
               GO_RECEIVE_TRANSACTION
          ELSE IF AUDITERROR THEN   % program logic error
               DMTERMINATE          % not in transaction state
          ELSE IF DATAERROR THEN
               DMTERMINATE
          ELSE DMTERMINATE;         % should not happen

  *******************************************
  * END TRANSACTION TIME (with concurrency) *
  *******************************************

END-TRANSACTION CDOUT AUDIT RDS
     ON EXCEPTION
          IF AUDITERROR THEN        % program logic error
               DMTERMINATE          % not in transaction state
          ELSE IF DATAERROR THEN
               DMTERMINATE
          ELSE DMTERMINATE;         % should not happen

  *************
  * EXIT_COMS *
  *************

     % RDS handling no longer required %

     CLOSE DATABASE;
     IF DB_CLOSE_ERROR THEN
        DMTERMINATE;
     EXIT PROGRAM;
```

Subroutines Using COMS and DMSII (RDS STORE)

If you have the current release of DMSII and of COMS and your database uses the DASDL option RDS STORE, use the following program flow for interactive recovery. This program includes the RECREATE RDS option after the ABORT or DEADLOCK exception in both the BEGIN-TRANSACTION and END-TRANSACTION statements.

```
  *******************
  * INITIALIZE_COMS *
  *******************

     OPEN UPDATE DATABASE
          ON EXCEPTION
               DMTERMINATE;
     ESTABLISH COMS LINK;
     ENABLE INPUT COMS "ONLINE"
       CREATE RDS;                         % required
       MOVE "ONLINE" TO RDS.ID;            % required
       MOVE CDIN.PROGRAMDESG TO RDS.PROG; % required
     % MOVE CDIN.RESTARTLOC TO RDS.LOC;    % no longer required

  ************************************************
  * BEGIN TRANSACTION TIME (without concurrency) *
  ************************************************

BEGIN-TRANSACTION CDIN NO-AUDIT
     ON EXCEPTION
          IF ABORT THEN           % not with INDEPENDENTTRANS option
               RECREATE RDS;      % unique for RDS STORE case
               GO_RECEIVE_TRANSACTION;
          ELSE IF AUDITERROR THEN % program logic error
               DMTERMINATE        % already in transaction state
          ELSE IF DEADLOCK THEN
               GO_RECEIVE_TRANSACTION
          ELSE DMTERMINATE;       % should not happen

  *********************************************
  * BEGIN TRANSACTION TIME (with concurrency) *
  *********************************************

BEGIN-TRANSACTION CDIN NO-AUDIT
     ON EXCEPTION
          IF AUDITERROR THEN      % program logic error
               DMTERMINATE        % already in transaction state
          ELSE IF DEADLOCK THEN
               GO_RECEIVE_TRANSACTION
          ELSE DMTERMINATE;       % should not happen
  **********************************************
  * END TRANSACTION TIME (without concurrency) *
  **********************************************

END-TRANSACTION CDOUT AUDIT RDS
     ON EXCEPTION
          IF ABORT OR DEADLOCK THEN % no abort with INDEPENDENTTRANS
                                                              option
               RECREATE RDS;        % unique for RDS STORE case
               GO_RECEIVE_TRANSACTION;
          ELSE IF AUDITERROR THEN   % program logic error
               DMTERMINATE          % not in transaction state
          ELSE IF DATAERROR THEN
               DMTERMINATE
          ELSE DMTERMINATE;         % should not happen

  *******************************************
  * END TRANSACTION TIME (with concurrency) *
  *******************************************

END-TRANSACTION CDOUT AUDIT RDS
     ON EXCEPTION
          IF DEADLOCK THEN
               RECREATE RDS;        % unique for RDS STORE case
               GO_RECEIVE_TRANSACTION;
          ELSE IF AUDITERROR THEN   % program logic error
               DMTERMINATE          % not in transaction state
          ELSE IF DATAERROR THEN
               DMTERMINATE
          ELSE DMTERMINATE;         % should not happen

  *************
  * EXIT_COMS *
  *************

     % RDS handling no longer required %

     CLOSE DATABASE;
     IF DB_CLOSE_ERROR THEN
        DMTERMINATE;
     EXIT PROGRAM;
```

Interactive Recovery Programs with SIM Databases

This subsection provides the program code for using COMS recovery with SIM databases.

A sample of the program flow is presented, highlighting the six subroutines necessary for COMS/SIM recovery. There are two possible main loops in the recovery program, one with an explicit SEND statement and the other with a SEND statement built into the END-TRANSACTION subroutine.

Using Exception-Condition Statements

As a general rule, an application program should go to end of task (EOT) once the database that is to be synchronized with COMS is closed. Exceptions on the close of a DMSII database with concurrency control do not affect any of the completed transaction states of the interactive application program. By default, once a transaction state has been committed to the database, it is not backed out due to an abort or a halt/load recovery.

You can use the ABORT-TRANSACTION statement to discontinue the current transaction state. If this statement is executed by an application program, the current transaction is not resubmitted by COMS.

For information about error messages on program termination, or about the ABORT-TRANSACTION statement, see the *DMSII Utilities Operations Guide.*

Declaration and Main Program Loop

Following is an example of the program flow for the declaration and main loop of a COMS/SIM recovery program. The program is presented in pseudolanguage, which uses indentation to indicate the scope of each statement.

```
  ********************
  * DECLARATION PART *
  ********************

SEMANTIC database DATABASE;
COMS HEADER CDIN;
COMS HEADER CDOUT;
DATA AREA MSG;
```

```
 *************
 * MAIN LOOP *
 *************

    INITIALIZE_COMS;
% main loop with explicit send %
    WHILE CDIN.STATUS NOT = 99 DO
         RECEIVE CDIN INTO MSG;
         IF CDIN.STATUS NOT = 99 THEN
              PREPARE_MSG;     % your message-processing code %
              BEGIN-TRANSACTION;
              UPDATE_DATABASE;
              END-TRANSACTION CDOUT;
              SEND CDOUT FROM MSG;
% or main loop with send built into end transaction %
    WHILE CDIN.STATUS NOT = 99 DO
         RECEIVE CDIN INTO MSG;
         IF CDIN.STATUS NOT = 99 THEN
              PREPARE_MSG;     % your message-processing code %
              BEGIN-TRANSACTION;
              UPDATE_DATABASE;
              END-TRANSACTION CDOUT;
    EXIT_COMS;
```

Section 7
Batch Recovery

This section describes a programming method for processing batch input in database applications. Batch recovery through COMS is not supported in the Pascal programming language.

You can use batch programs to process a set of transactions without continually interacting with the terminal. At any time, a batch program can be initiated by an interactive transaction. The method of batch programming described in this section allows batch processing and online processing to take place simultaneously. This method also ensures the following:

- The batch job will be synchronized with online transactions.
- Recovery can be performed.

This method of batch programming requires all batch transactions to be invoked by an incoming online transaction from a terminal or another program. All programs must be run through COMS direct windows.

Before you begin the procedures in this section, read the descriptions of the Conversation Area field of the input and output headers in Section 3, "Communicating with COMS through Direct Windows."

Recovery Considerations

Batch recovery procedures require each transaction to be synchronized with the COMS-generated transaction trail for the database. Your batch program must retain enough information to know where to restart processing after database recovery and COMS recovery complete.

For example, in a case in which you are using a batch file, this information might include the batch file name and the actual key of the record that causes the update to be performed.

Input to batch transactions can take one of three forms: the input transactions that initiated the program; input transactions read from a tape, disk file, or database; and transactions that are resubmitted during recovery.

To synchronize recovery between online programs and batch programs, the batch programs must identify to COMS each transaction being performed. This is accomplished by using the BEGIN-TRANSACTION WITH option, which allows COMS to audit the input transaction on the transaction trail. In the event of a recovery, COMS can then reprocess the online and batch transactions in the original order.

One way to retain the information needed for recovery is to place that information in the Conversation Area field of the input header before the program enters transaction state. This information is used during recovery to determine which file and record the last processed transaction came from and, thus, where to continue processing in the batch file. If the application is using a DMSII database, the information regarding batch recovery should be duplicated in the restart data set record. If the application is using a SIM database, the information regarding batch recovery should be duplicated in the restart class. For information on classes, see the *InfoExec SIM Technical Overview.*

Data to be saved in the input header must be placed in the Conversation Area field before executing a BEGIN-TRANSACTION statement. For batch programs, it is sufficient to save the file name and the record number. The program can also place a copy of the record in the message variable specified by the BEGIN-TRANSACTION statement. If this were done during recovery, the program would not require the input file to be available. This is an important operational consideration for sites that require archival recovery.

Recovery data must also be saved in the database. This is necessary for those cases in which COMS does not resubmit transactions to the batch application after a halt/load (that is, no transactions for this batch program have been rolled off the database). The program must be able to restart from the information in the database. Therefore, the batch program must also store in the database the restart information saved in the Conversation Area field.

For every BEGIN-TRANSACTION and END-TRANSACTION statement, COMS needs to be notified so that it can put the information into its transaction trail. This requires COMS application programs that have enabled batch mode to use the BEGIN-TRANSACTION statement that identifies the message variable. This information is used for ordering and recovery purposes.

During recovery, in each resubmitted transaction (that is, each transaction for which the Status field of the input header contains the value 92), the header and message area appear as they did at the begin-transaction point when the transaction was first processed. As a result, the batch file and record location that you placed in the Conversation Area field of the input header when processing the record the first time are resubmitted along with the message variable.

The following are the results produced by a batch transaction failure:

- Batch transaction failure during transaction state

  When a batch transaction repeatedly fails after the mid-transaction point, it is resubmitted with a value of 90 in the Status Value field of the input header.

  If a batch program receives a value of 90 for a transaction, do not allow the program to process the transaction, because that transaction caused the original series of aborts.

- Batch transaction failure after transaction state

  When a batch transaction repeatedly fails after a transaction occurs, it is resubmitted with a value of 80 in the Status Value field of the input header.

> If a batch program receives a value of 80 for a transaction, do not allow the program to restart a transaction immediately, because the completion of that transaction initially caused the program repeated failures.

After the begin-transaction point, the message area and relevant portions of the header should appear the same regardless of whether the program is in normal or recovery mode. Therefore, the programming steps that you take after this point do not need to differentiate recovery transactions from normal transactions.

For additional information on the restart data set and recovery, refer to Section 6, "Interactive Recovery."

Programming for batch recovery differs, depending on whether or not your database has concurrency control. The remainder of this section presents programming information in the following subsections:

- Writing programs using batch recovery without concurrency
- Writing programs using batch recovery with concurrency

Writing Programs Using Batch Recovery without Concurrency

This subsection provides the program flow and subroutines for programs that use batch synchronized recovery. These are programs that access DMSII databases without concurrency. The examples are presented in pseudolanguage.

A COMS batch program is initiated by an interactive transaction. This transaction can be received only while a program is in interactive mode. The batch recovery program first initializes batch mode to receive and process any transaction being recovered due to a halt/load. It then changes to interactive mode to receive the transaction that initiated the batch program. The program then switches to batch mode to process the batch data. Finally, the batch program switches back to interactive mode to receive either another interactive transaction or the COMS notification to go to end of task.

In batch mode, the program does its processing until it gets an abort on a DMSII statement. In response to the abort, the program has to receive the transactions that were backed out by the database abort and reapply them to the database. This is done while the program is still in batch mode. The transactions are identical copies of what the program gave to COMS at the begin-transaction point in the message area. They must provide enough information to simulate the original database update.

When the recovery is complete, the program restarts its batch processing, continuing where it left off. The restart data set and/or the Conversation Area field of the input header must provide enough information for the program to determine the restart location for the batch process. Examples of the necessary information are file title and record number if the program reads a disk file as input to the database, or a unique key into the data set if the program scans through the database. When all the batch work is done, the program goes back into interactive mode to receive either a new transaction or the COMS notification to go to end of task.

Application recovery data must be stored in both the Conversation Area field of the input header and the restart data set. Upon resubmission of messages, the application program uses the recovery data stored in the Conversation Area field when the transaction was originally processed. When a halt/load occurs, a transaction must have been backed out of the database to be resubmitted by COMS. The application uses the recovery data in the restart data set.

At initialization, the program can be in one of two operation modes. It might have been brought into the mix because a new transaction was given to it. It might have been restarted because it was running earlier and it aborted, the system halt/loaded, or the database was rolled back for archival recovery. If the program was restarted, it does not receive its initial transaction, but receives only the transactions supplied with the BEGIN-TRANSACTION statement.

Declaration and Main Program Loop

Your program should start by declaring its database, headers, and data message area. When it moves to the main loop, it should initialize the COMS interface and then handle any recovery of batch processing that had not been completed. When it has finished running any interrupted processing, the program should then handle any online input from COMS that submits new batch runs, or it should terminate. Following is the program flow for the declaration and main loop of the batch synchronized recovery program. The examples are presented in pseudolanguage, which uses indentation to indicate the scope of each statement.

```
 ********************
 * DECLARATION PART *
 ********************

database DATABASE;
COMS HEADER CDIN;
COMS HEADER CDOUT;
DATA AREA MSG;

 *************
 * MAIN LOOP *
 *************

   INITIALIZE_COMS;
   IF MY_BATCH_RESTARTED THEN
      DO_BATCH_PROCESSING;
   INITIALIZE_ONLINE_MODE
   WHILE CDIN.STATUS NOT = 99 DO
      RECEIVE CDIN INTO MSG;
      IF CDIN.STATUS NOT = 99 THEN
         INITIALIZE_BATCH_MODE;
         DO_BATCH_PROCESSING;
      INITIALIZE_ONLINE_MODE;
   EXIT_COMS;
```

Initializing COMS

This step of the batch synchronized recovery program should open the database to be updated, establish a link to COMS, and initialize the restart data set. It should also find the restart data set record for this program. For more information on initialization, see Section 3, "Communicating with COMS through Direct Windows." For more information on the restart data set, see Section 6, "Interactive Recovery."

Following is the program flow for the initialization step of the batch synchronized recovery program:

```
*******************
* INITIALIZE_COMS *
*******************

  OPEN UPDATE DATABASE
       ON EXCEPTION
            DMTERMINATE;
  ESTABLISH COMS LINK;
  ENABLE INPUT COMS "BATCH";
  RESET MY_BATCH_RESTARTED;
  % The program has just come up. It must check to see if any
  % of its transactions have been rolled back. If transactions have
  % been rolled back, then COMS resubmits them from the
  % transaction trail. The CONVERSATION_AREA
  % of such a resubmitted message can include the
  % data necessary to simulate the database update.


   RECEIVE CDIN INTO MSG
     NO DATA
       RECEIVE FOUND_RECOVERY_MESSAGE
     ELSE
     SET FOUND_RECOVERY_MESSAGE  % COMS is resubmitting transactions
                                 % that were rolled back.
       SET MY_BATCH_RESTARTED;   % After the resubmissions are
                                 % complete, the remaining
                                 % batch transactions
                                 % must be completed.
```

```
 % COMS might not have transactions to resubmit, but the batch update
 % might have been incomplete during the last run. The program must
 % check the RESTART DATA SET now.

 LOCK RDS AT RDS.ID = "ONLINE" AND   % Note: ONLINE not BATCH.
       RDS.PROG = CDIN.PROGRAMDESG
    ON EXCEPTION
      IF NOTFOUND THEN         % No recovery need be done.
        CREATE RDS             % RDS area is created for later use.
        % The following code is no longer required.
        % MOVE "ONLINE"     TO RDS.ID
        % MOVE CDIN.PROGRAMDESG TO RDS.PROG
        % MOVE CDIN.RESTARTLOC TO RDS.LOC
        PLACE_USERDATA_IN_RESTART_REC
      ELSE
         DMTERMINATE           % Some other DMERROR.
      ELSE                     % Last run was incomplete.
       SET MY_BATCH_RESTARTED; % The remaining batch transactions
                               % must be completed.
IF FOUND_RECOVERY_MESSAGE THEN % Finish all resubmitted transactions.
  DO_INITIAL_BATCH_RECOVERY;
```

Initializing Interactive Mode

Interactive mode allows a program to receive nonrecovery transactions, and to be signaled when to go to end of job (EOJ). It deletes an eventual batch mode restart-data-set record. It reinitializes the restart data set to contain the interactive designator of the program and informs COMS about the mode change.

Following is the program flow for initializing the interactive mode in the batch synchronized recovery program:

```
********************************
* INITIALIZE_INTERACTIVE_MODE *
********************************

   BEGIN-TRANSACTION CDIN NO-AUDIT RDS;
   LOCK RDS AT RDS.ID = RDS.ID AND RDS.PROG = RDS.PROG
        ON EXCEPTION
             IF NOT FOUND THEN
                  RECREATE RDS;
                  INITIALIZE RDS.MYRESTARTINFO;
              ELSE DMTERMINATE
        ELSE
             DELETE RDS;
   ENABLE INPUT CDIN "ONLINE";
   END-TRANSACTION AUDIT RDS;
```

Initializing Batch Mode

Initializing batch mode allows your program to receive only recovery transactions. This program subroutine also informs COMS of the switch from interactive to batch mode. Following is the program flow for initializing batch mode within your batch synchronized recovery program:

```
*************************
* INITIALIZE_BATCH_MODE *
*************************

   ENABLE INPUT CDIN "BATCH";
```

Initial Batch Recovery

You can use the batch recovery subroutine to allow your program to participate in the COMS synchronized/archival recovery when it receives a recovery transaction in INITIALIZE_COMS. This subroutine receives recovery transactions and updates a database.

Following is the program flow for doing initial batch recovery within the batch synchronized recovery program:

```
*****************************
* DO_INITIAL_BATCH_RECOVERY *
*****************************

   RESET NO_MORE_RECOVERY
   WHILE NOT NO_MORE_RECOVERY DO
        HANDLE_RECOVERY_TRANSACTION;
        RECEIVE CDIN INTO MSG
             NO DATA
                 SET NO_MORE_RECOVERY;
```

Abort Batch Recovery

Use this subroutine to allow your program to participate in synchronized or archival recovery when it discovers a database abort. The subroutine receives recovery transactions and updates the database.

Following is the program flow for doing abort batch recovery within the batch synchronized recovery program:

```
***************************
* DO_ABORT_BATCH_RECOVERY *
***************************

    RESET NO_MORE_RECOVERY
    DO
        RECEIVE CDIN INTO MSG
            NO DATA
                SET NO_MORE_RECOVERY
            ELSE
                HANDLE_RECOVERY_TRANSACTION
    UNTIL NO_MORE_RECOVERY;
```

Handling the Recovery Transaction

Following is the subroutine for handling the recovery transaction within the batch synchronized recovery program:

```
*******************************
* HANDLE_RECOVERY_TRANSACTION *
*******************************

    SIMULATE_ORIGINAL_TRANSACTION;
    MOVE MY_RESTARTINFO TO CDIN.CONVERSATION.RESTARTINFO;
    BEGIN-TRANSACTION CDIN USING MSG NO-AUDIT RDS;
    UPDATE_DATABASE;
    MOVE MY_RESTARTINFO TO RDS.RESTARTINFO;
    END-TRANSACTION CDOUT AUDIT RDS;
```

Batch Processing

You use the batch processing subroutine to update a database until an abort is detected or the processing is done. This subroutine stores restart information necessary to identify where to continue for recovery. This is done in both the transaction trail (using the Conversation Area field of the input header) and in the restart data set.

Following is the program flow for batch processing within the batch synchronized recovery program:

```
  ***********************
  * DO_BATCH_PROCESSING *
  ***********************

     DO
          MOVE MY_RESTARTINFO TO RDS.RESTARTINFO;
          MOVE MY_RESTARTINFO TO CDIN.CONVERSATION.RESTARTINFO;
          BEGIN-TRANSACTION CDIN USING MSG NO AUDIT RDS;
          UPDATE_DATABASE;
          END-TRANSACTION CDOUT AUDIT RDS;
     UNTIL ALL_WORK_DONE;
```

COMS and DMSII (No RDS STORE)

If you have COMS and DMSII and do not use the RDS STORE option, use the following program flow for batch processing:

```
  **************************
  * BEGIN TRANSACTION TIME *
  **************************

BEGIN-TRANSACTION CDIN NO-AUDIT USING MSG
     ON EXCEPTION
          IF ABORT THEN           % not with INDEPENDENTTRANS option
               GO_DO_BATCH_RECOVERY
          ELSE IF AUDITERROR THEN % program logic error
               DMTERMINATE        % already in transaction state
          ELSE IF DEADLOCK THEN
               GO_DO_BATCH_RECOVERY
          ELSE DMTERMINATE;       % should not happen

  ************************
  * END TRANSACTION TIME *
  ************************

END-TRANSACTION CDOUT AUDIT RDS
     ON EXCEPTION
          IF ABORT OR DEADLOCK THEN % no abort with INDEPENDENTTRANS
                                                              option
               GO_DO_BATCH_RECOVERY
          ELSE IF AUDITERROR THEN   % program logic error
               DMTERMINATE          % not in transaction state
          ELSE IF DATAERROR THEN
               DMTERMINATE
          ELSE DMTERMINATE;         % should not happen
```

COMS and DMSII (RDS STORE)

If you have the current release of DMSII and of COMS and your program uses the DASDL option RDS STORE, use the following program flow for batch processing (note the RECREATE RDS option after the ABORT exception in the BEGIN-TRANSACTION statement and after the ABORT or DEADLOCK exception in the END-TRANSACTION statement):

```
  **************************
  * BEGIN TRANSACTION TIME *
  **************************

BEGIN-TRANSACTION CDIN NO-AUDIT USING MSG
     ON EXCEPTION
          IF ABORT THEN           % not with INDEPENDENTTRANS option
               RECREATE RDS;      % unique for RDS STORE case
               GO_DO_BATCH_RECOVERY
          ELSE IF AUDITERROR THEN % program logic error
               DMTERMINATE        % already in transaction state
          ELSE IF DEADLOCK THEN
               GO_DO_BATCH_RECOVERY
          ELSE DMTERMINATE;       % should not happen

  ************************
  * END TRANSACTION TIME *
  ************************

END-TRANSACTION CDOUT AUDIT RDS
     ON EXCEPTION
          IF ABORT OR DEADLOCK THEN % no abort with INDEPENDENTTRANS
                                                              option
               RECREATE RDS;        % unique for RDS STORE case
               GO_DO_BATCH_RECOVERY;
          ELSE IF AUDITERROR THEN   % program logic error
               DMTERMINATE          % not in transaction state
          ELSE IF DATAERROR THEN
               DMTERMINATE
          ELSE DMTERMINATE;         % should not happen
```

Exiting COMS

Use the following subroutine to exit COMS whether or not your program uses the DASDL *RDS STORE* option:

```
*************
* EXIT_COMS * (Database with or without RDS STORE)
*************

    % RDS handling no longer required if coming out of online mode %

    CLOSE DATABASE;
    IF DB_CLOSE_ERROR THEN
              DMTERMINATE;
    EXIT PROGRAM;
```

Writing Programs Using Batch Recovery with Concurrency

This section provides the program flow and subroutines for programs that use batch recovery. These are programs that access DMSII databases with concurrency, or programs that access SIM databases. These databases do not need additional synchronized recovery programming.

A COMS batch program is initiated by an interactive transaction. This transaction can be received only while a program is in interactive mode. The batch recovery program first initializes batch mode to receive and process any transaction being recovered due to a halt/load. It then changes to interactive mode to receive the transaction that initiated the batch program.

The program then switches to batch mode to process the batch data. Finally, the batch program switches back to interactive mode to receive either another interactive transaction or the COMS notification to go to end of task (EOT).

At initialization, the program can be in one of two operation modes. It might have been brought into the mix because a new transaction was given to it. If the program was restarted, it does not receive its initial transaction, but does receive only the transactions supplied with the BEGIN-TRANSACTION statement. It might have been restarted because it was running earlier and it aborted, because of a system halt/load, or because the database was rolled back for archival recovery.

Application recovery data must be stored both in the Conversation Area field of the input header and in the restart class. Upon resubmission of messages, the application program uses the recovery data stored in the Conversation Area field when the transaction was originally processed. When a halt/load occurs, no transaction is resubmitted by COMS if it has not been backed out of the database. If no transaction is resubmitted to the batch application program, but there is recovery data in the restart class, the application program must start reprocessing from the data in the restart class.

If the program faults and gets terminated abnormally, COMS automatically restarts the program. After enabling the BATCH option, the program must receive the transaction that caused it to fault. This transaction has a status code of 90. Do not reprocess the transaction, but rather evaluate the error message that is returned. The transaction is an identical copy of what the program gave to COMS at the begin-transaction point in the message area.

Once the transaction that caused the abort is processed, the program restarts its batch processing, continuing where it left off. The restart class has to contain enough information to allow the program to find where in the batch process to restart. For information on classes, see the *InfoExec SIM Technical Overview.*

Examples of the necessary information are file title and record number if the program reads a disk file as input to the database, or a unique key into the data set if the program scans through the database. When all the batch work is done, the program goes back into interactive mode to receive either a new transaction or the COMS notification to go to end of task.

Declaration and Main Program Loop

Your program should start by declaring its database, headers, and data message area. When it moves to the main loop, it should initialize the COMS interface and then handle any recovery of batch processing that had not been completed. When it has finished running any interrupted processing, the program should then handle any online input from COMS that submits new batch runs, or it should terminate.

Following is the program flow for the declaration and main loop of the batch recovery program. The program is presented in pseudolanguage, which uses indentation to indicate the scope of each statement.

```
  ********************
  * DECLARATION PART *
  ********************

database DATABASE;
COMS HEADER CDIN;
COMS HEADER CDOUT;
DATA AREA MSG;

  *************
  * MAIN LOOP *
  *************

     INITIALIZE_COMS;
     IF MY_BATCH_RESTARTED THEN
          DO_BATCH_PROCESSING;
     INITIALIZE_ONLINE_MODE
     WHILE CDIN.STATUS NOT = 99 DO
          RECEIVE CDIN INTO MSG;
          IF CDIN.STATUS NOT = 99 THEN
               INITIALIZE_BATCH_MODE;
               DO_BATCH_PROCESSING;
          INITIALIZE_ONLINE_MODE;
     EXIT_COMS;
```

Initializing COMS

This step of the batch recovery program should open the database to be updated, establish a link to COMS, and initialize the restart class. It should also find the restart class entity for this program. For more information on initialization, see Section 3, "Communicating with COMS through Direct Windows." For more information on the restart data set, see Section 6, "Interactive Recovery."

Following is the program flow for the initialization step of the batch recovery program:

```
*******************
* INITIALIZE_COMS *
*******************

   OPEN FILE
   OPEN UPDATE DATABASE
        ON EXCEPTION
             STOP RUN;
   ESTABLISH COMS LINK;
   ENABLE INPUT COMS "BATCH";
   RECEIVE CDIN INTO MSG
        NO DATA RESET FOUND_RECOVERY_MESSAGE
   ELSE
        SET MY_BATCH_RESTARTED;
        SET FOUND_RECOVERY_MESSAGE;


   IF NOT FOUND_RECOVERY_MESSAGE
     SELECT RSTQD FROM RSTINFO
       WHERE RST-PROG-DESG = MY-PROG-DESG;
     RETRIEVE RSTQD
       ON EXCEPTION
         MOVE 1                    TO WS-RETRIEVE-FIELD;
     IF RSTINFO-NOT-RETRIEVED
         INSERT RSTINFO
           ASSIGN WS-PROG-NAME     TO RST-PROG-NAME
           ASSIGN MY-PROG-DESG     TO RST-PROG-DESG
           ASSIGN MY-RESTARTINFO   TO RST-INFO;
   REPOSITION FILE
   IF RSTINFO-RETRIEVED
     SET MY_BATCH_RESTART;
   IF FOUND_RECOVERY_MESSAGE THEN
     DO_INITIAL_BATCH_RECOVERY;
```

Initializing Interactive Mode

Interactive mode allows a program to receive nonrecovery transactions, and to be signaled when to go to end of job (EOJ). It deletes an eventual batch mode restart class entity.

Following is the program flow for initializing the interactive mode in the batch recovery program:

```
**************************
* INITIALIZE_ONLINE_MODE *
**************************

   BEGIN-TRANSACTION;
   DELETE RSTINFO
     WHERE RST-PROG-DESG = MY-PROG-DESG;
   ENABLE INPUT CDIN "ONLINE";
   MOVE CDIN.PROGRAMDESG             TO MY-PROG-DESG;
   END-TRANSACTION;
```

Initializing Batch Mode

Initializing batch mode allows your program to receive only recovery transactions. This program subroutine also informs COMS of the switch from interactive to batch mode. Following is the program flow for initializing batch mode within your batch recovery program:

```
*************************
* INITIALIZE_BATCH_MODE *
*************************

   ENABLE INPUT CDIN "BATCH";
   MOVE CDIN.PROGRAMDESG        TO MY-PROG-DESG;
```

Initial Batch Recovery

You can use the batch recovery subroutine to allow your program to participate in the COMS recovery and archival recovery when it receives a recovery transaction in INITIALIZE_COMS. This subroutine receives recovery transactions and updates a database.

Following is the program flow for doing initial batch recovery within the batch recovery program:

```
*****************************
* DO_INITIAL_BATCH_RECOVERY *
*****************************

   RESET NO_MORE_RECOVERY
   WHILE NOT NO_MORE_RECOVERY DO
        HANDLE_RECOVERY_TRANSACTION;
        RECEIVE CDIN INTO MSG
             NO DATA
                 SET NO_MORE_RECOVERY;
   REPOSITION FILE;
```

Error Recovery

Use this subroutine when an error is returned from either BEGIN-TRANSACTION or END-TRANSACTION statements. The error indicates that the transaction has not been committed to the database. COMS submits this transaction to the program by way of the RECEIVE statement. The subroutine receives the recovery transaction and updates the database.

Following is the program flow for handling begin- and end-transaction errors within the batch recovery program:

```
*****************************
* DO_BTR_ETR_ERROR_RECOVERY *
*****************************

RESET NO_MORE_RECOVERY
RECEIVE CDIN INTO MSG
  NO DATA
    SET NO_MORE_RECOVERY
  ELSE
    HANDLE_RECOVERY_TRANSACTION
```

Handling the Recovery Transaction

Following is the subroutine for handling the recovery transaction within the batch recovery program:

```
*******************************
* HANDLE_RECOVERY_TRANSACTION *
*******************************

    SIMULATE_ORIGINAL_TRANSACTION;
    MOVE MY_RESTARTINFO TO CDIN.CONVERSATION.RESTARTINFO;
    BEGIN-TRANSACTION CDIN USING MSG;
    UPDATE_DATABASE;
    MODIFY RSTINFO
      ASSIGN MY-RESTARTINFO   TO RST-INFO
      WHERE RST-PROG-DESG = MY-PROG-DESG;

    END-TRANSACTION CDOUT;
```

Batch Processing

Use the batch processing subroutine to update a database until the processing is done. This subroutine stores restart information necessary to identify where to continue for recovery. This is done both in the transaction trail (using the Conversation Area field of the input header) and in the restart class.

Following is the program flow for batch processing within the batch recovery program:

```
***********************
* DO_BATCH_PROCESSING *
***********************

   DO

        MOVE MY_RESTARTINFO TO CDIN.CONVERSATION.RESTARTINFO;
        BEGIN-TRANSACTION CDIN USING MSG;
        UPDATE_DATABASE;
        MODIFY RSTINFO
          ASSIGN MY-RESTARTINFO       TO RST-INFO
          WHERE RST-PROG-DESG = WS-PROG-DESG;
        END-TRANSACTION CDOUT;
   UNTIL ALL_WORK_DONE;
```

Transaction State

Batch application programs must use a BEGIN-TRANSACTION statement that identifies the message variable for entering transaction state. Moreover, the message variable must contain an image of the batch transaction read from the disk file or tape file.

These requirements are necessary to allow COMS to capture an image of the transaction on the transaction trail. Then, if the database is ever rolled back, COMS can resubmit the transaction to the batch application in coordination with the resubmission of the online transactions. Another benefit is that the batch file that contained all the original input transactions does not need to be present during archival recovery.

Following is the program flow for transaction state processing within the batch recovery program:

```
**************************
* BEGIN TRANSACTION TIME *
**************************

BEGIN-TRANSACTION CDIN USING MSG
   ON EXCEPTION
        DO_BTR_ETR_ERROR_RECOVERY;

************************
* END TRANSACTION TIME *
************************

END-TRANSACTION CDOUT AUDIT RDS
   ON EXCEPTION
        DO_BTR_ETR_ERROR_RECOVERY;
```

Section 8
Security

COMS automatically performs security checks, provided that a message-routing and security scheme has been defined with the COMS Utility program and stored in the configuration file. For more information on COMS security, see the *COMS Configuration Guide.*

This section provides information on the security-checking routines that you can write into application programs and processing items to augment COMS security or to function independently. This type of security checking is called programmatic security. You can also use security category designators and usercode designators in application programs and processing items to assist in performing security checks on a more refined level than COMS can do alone.

To use security categories and other security-related COMS entities for programmatic security, you must have declared an input header and an output header in your program. You also need to use COMS service functions when writing routines for programmatic security. Refer to Section 3, "Communicating with COMS through Direct Windows," for additional information about the input and output headers. Refer to Section 4, "Accessing Service Functions," for information about the service functions.

When to Use Programmatic Security

Although COMS provides highly effective and flexible tools for configuring a COMS security scheme, some degree of programmatic security is an alternative to consider under the following circumstances:

- You want to perform more refined security checking than COMS security alone can provide.
- You are not using trancodes for message routing or security checking.
- You are using trancodes for message routing, but you have not assigned security categories to the trancodes you are using.
- You are using trancodes to which security categories have been assigned, but you want to perform an additional security check after the program retrieves a record from the database.
- You want to provide data-dependent security.

Using the Input Header in Security Checking

The input header is a message header that provides routing and other descriptive information about received messages. When you use an input header in a program that runs under COMS, COMS places values into the fields of the input header each time the program receives a message.

The following list, which provides the types of information that COMS places in the input header fields, can be used in security checking:

- The usercode associated with the message
- The security categories associated with the session that originated the message
- The module function index (MFI) representing the trancode associated with the message
- The time and date when the message was first encountered by COMS
- The station or program that originated the message

The following paragraphs describe specific techniques you can use for security checking in programs that contain an input header. Since most values that COMS places into the input header fields are designators, you have to call service functions of the COMS library to exchange the designators for names. Refer to Section 4, "Accessing Service Functions," for more information about the service functions. Refer to Section 3, "Communicating with COMS through Direct Windows," for more information about the input header.

Using the Usercode Designator

When a program receives a message, the Usercode field of the input header contains a designator representing the usercode associated with the incoming message. You can call the GET_NAME_USING_DESIGNATOR service function to exchange the usercode designator for a usercode name.

A program can perform a security test by checking the usercode associated with a particular message against a list of usercodes kept in the program. This type of security test is not the most efficient kind of programmatic security you can use, but it could be useful if you are not using trancodes or security categories.

Using the Security Designator

When a program receives a message, the Security Designator field of the input header contains the security designator for the session, which represents the intersection of the security-category lists assigned to the usercode and the station that originated the incoming message.

If a security-category list is assigned to your program, COMS provides a way to find out whether any security category in the list of the program intersects with the security-category list of your session. You simply have the program call the TEST_DESIGNATORS service function to test the validity of the security category.

Using the Module Function Index

If a process-security error occurs, the Function Status field of the input header contains a value to indicate the error condition. For the meaning of this and other values and mnemonics, see Appendix A.

If you have associated positive MFIs with your trancodes or group of trancodes and have not associated security categories with those trancodes, you can use the MFIs to point to the appropriate code that determines whether the user who entered the message is allowed to submit this kind of trancode. Refer to "Using Module Function Indexes with Input" in Section 3, "Communicating with COMS through Direct Windows," for information about MFIs. Using the usercode designator and/or the session-security designator are other possible methods for checking user validity.

Checking Database Records After Retrieval

This technique can be combined with other programmatic security techniques to produce a more refined security check than is possible with COMS security alone. The technique involves the program actually retrieving a database record before making the final decision as to whether a certain security-category list entitles the user to see or update the particular database record. Following are two examples of how to use this technique.

Example 1

At the ABC Company, a particular trancode has been defined to identify a transaction as an inquiry into the personnel database. Employees who work in the Shipping and Receiving Department obviously should not be permitted to make inquiries about personnel records. So the trancode representing the personnel inquiry transaction is not assigned to the window that Shipping and Receiving employees use to communicate with COMS.

However, a clerk who works in the Personnel Department is permitted to inquire about personnel records, except for his own personnel record and the personnel records of upper management. The inquiry transaction always works the same way, except that each personnel record belongs to a different person. Because the program does not know until the inquiry has been made who the record belongs to, the program must perform a security test of record names to determine whether the record is one of the few records that the personnel clerk is not allowed to see.

Example 2

Following is an example of making a more refined decision by using programmatic security checking.

At the XYZ Company, suppose that security categories have been defined for the following employee levels:

- Senior manager
- Junior manager
- Clerk

Suppose that the Payroll Department has a rule that permits senior managers to access only their own payroll records and those belonging to junior managers, while permitting junior managers to access their own payroll records and the payroll records of everyone else except for senior managers. Payroll clerks are permitted to see only their own payroll records and those belonging to other payroll clerks.

The security requirements of this situation are too complicated for COMS to handle alone. Whenever a payroll transaction is entered, authority of the user to see a payroll record of a senior manager, junior manager, or clerk depends on the identity of a particular record in the data base. Therefore, the program needs to do a more refined security check.

First, the program could find the record on the database and determine what kind of employee it belongs to. Next, the program could find out, through a programmatic convention, whether the security category associated with the record is in the security-category list for the session that originated the transaction. To do this, the program would call the TEST_DESIGNATORS service function. At this point, the program is ready to decide whether the transaction is valid for this record.

If the company hires a new manager, he or she just needs to be given his or her own usercode with the appropriate security categories assigned to it. The security-checking routines in the program need not be adjusted, nor the program recompiled.

When the destination is a station, the message is a program-to-station message. COMS security permits a program to send a processed message back to the originating station. Alternatively, a program can send a message to any station or window dialogue within the window of the program. In addition, a program can send a message to a printer, which is known to COMS as a single-output window and defined with the Network Definition Language II (NDLII) as *MYUSE = OUTPUT*.

When the destination for a processed message is another transaction-processing program, the message is a program-to-program message. COMS security permits a program to send a message to another program only when the security category belonging to the trancode in the message is in the security-category list assigned to the destination program. Refer to “Routing Messages by Specifying a Destination” in Section 3, “Communicating with COMS through Direct Windows,” for additional information.

Thus, the primary purpose of COMS process security is to prevent programs from processing transactions submitted by users or programs who are unable to obtain the security clearance required by COMS.

How COMS Handles Security Errors

When a user fails access security and cannot log on to a particular station or window, the Menu-Assisted Resource Control (MARC) program sends an error message to the user. Refer to the *MARC Operations Guide* for a list of error messages that can be received in regard to access security.

When a message fails process security, the failure occurs because the security-category list of the user or program submitting the message does not include the security category associated with the trancode in the message. Under these circumstances, COMS assigns the failed message to the default agenda for the window.

Like all agendas, the default agenda must specify a program as a destination for messages, and can specify a list of processing items. The existence of a default agenda must be defined with the COMS Utility program and stored in the configuration file. You must write a destination program containing an input header and an output header if you want COMS to identify a security error and report it in the Function Status field of the input header of your program.

The program you write to handle security errors does not have to be written solely for error-handling. It can be an ordinary application program that is capable of processing various transactions associated with valid MFIs as well as handling security errors.

Section 9
Communicating with COMS through Remote-File Windows

As an alternative to using direct windows to access a station or program in COMS, you might want to use remote-file windows for the following reasons:

- You have existing programs that use the remote-file interface, either from the Generalized Message Control System (GEMCOS), the Command and Edit (CANDE) MCS, another Unisys software product, or user-written applications.
- You want to create a time-sharing type of application. You want to ensure that if multiple copies of a program are running, only one copy of that program will receive input from any particular user.

With remote files, each program is associated with a set of users. Messages entered in a current window are sent to the program associated with that window. This differs from use of direct windows, in which multiple copies of a program read from a common queue. In remote files, every message from a station is processed by the same copy of a program. In direct windows, the current message from a station might be processed by one copy of a program, while another message from that same station might be processed by another copy of the program.

For further information on remote files and file attributes, see the *A Series I/O Subsystem Programming Guide.* For further information on task attributes, see the *A Series Work Flow Administration and Programming Guide.* For additional information about remote files in COMS, connected with a comparison between COMS and CANDE, see the *COMS Migration Guide.*

There are two kinds of remote-file windows: dynamic and declared. The following two parts of this section tell you how to use each kind of remote-file window.

Dynamic Remote-File Windows

You can use dynamic remote-file windows to open to program environments that are not defined to COMS.

For example, when you are in a Menu-Assisted Resource Control (MARC) window session, you might want to run a remote-file program. MARC initiates the program and sets the STATION task attribute to the name of the station you are currently using. Since this task attribute is set, COMS is called to handle the opening of the remote file. When your program executes a statement to open a remote file, COMS approves the opening of the file and associates a dynamic window with it. MARC requests COMS to change the current window from a MARC window to the just-opened dynamic remote-file window, which might be identified as REM0001.

When you close the remote-file window, your current window is changed to the MARC window dialogue you were in before you opened the remote-file window. It is important to be aware of what your current program environment is, particularly if you move among several windows without closing them.

Dynamic remote-file windows can also be created when a remote-file program initiates a connection to a station on its own. The remote-file program initiates this connection in its remote file by adding the station to the remote-file station list. When the remote-file program requests a station that is connected to COMS, COMS opens a dynamic window for the station. You are not notified directly that the window has been opened, but you can use the COMS *?WINDOWS* command to verify that it has. Then, you can use the COMS *?ON* command (as in *?ON REM002*) to change the current window to the remote-file window.

One dynamic remote-file window is established per user. If the user at station one submits a *?WINDOWS* command, he or she might receive the message that the dynamic window that has been opened is REM0001. If another user submits the window command, he or she receives a message that a different dynamic window, such as REM0005, has been opened, and yet these users might both be accessing the same remote-file program through the same remote file.

Declared Remote-File Windows

You can use declared remote-file windows to create either single-user or multiuser environments within COMS. To declare a remote-file window, you need to run the COMS Utility. In the COMS Utility, you name your remote-file program and can set various options, such as number of users per program copy or number of copies of the program allowed. For more information on using the COMS Utility, see the *COMS Configuration Guide.*

The relative station numbers (RSNs) associated with remote files become particularly important with declared remote-file windows. An RSN is associated with each station in a remote file, depending on the number of program copies and users per copy you have set in the COMS Utility. When you close a window, COMS causes the remote-file program to receive an end-of-file (EOF) indication. Your program should check the LASTSTATION file attribute to determine on which RSN the EOF was given. If LASTSTATION is RSN 1, you should go to end of job (EOJ). If you receive any other RSN number, this means that the station associated with that RSN has closed its connection to the remote-file program; you might want to clean up local storage.

Single-User Declared Windows

Within COMS, when you move to a window using the COMS *?ON* command for single-user remote-file declared windows, COMS starts the remote-file program and sets the task attribute USERCODE. On the Program Activity menu of the COMS Utility, you can specify a usercode under which you would like your programs to run. If you do not do this, COMS assigns the usercode with which you logged on to COMS to the remote-file program it starts when you submit the *?ON RMT* command.

In a single-user declared remote-file window, each copy of a program is dedicated to one station. If another station tries to access the remote-file window, COMS starts a separate copy of it for that user. When no usercode is set in the Program Activity menu of the COMS Utility, COMS assigns to that copy the usercode currently logged on at the station.

Each remote-file window in a single-user environment has only one RSN assigned to it. If you decide to close this type of window, COMS causes an EOF indication to send the program to EOJ. Also, if COMS wants your program to go away for any reason, the program receives an EOF.

Multiuser Declared Windows

If you want to have multiuser declared windows, set the number of users to more than one in the Program Activity menu of the COMS Utility. Within COMS, when you submit the command *?ON RMT*, COMS starts the remote-file program and sets the STATION task attribute to a special COMS station.

If in the COMS Utility you have set the number of users to two, COMS is able to assign two users to each copy of the remote-file program. The following example shows how COMS would route three users who want to access the same remote-file program.

Remote files have RSNs associated with them. RSN 1 for a remote file is reserved by COMS for its own record-keeping purposes. The second and third RSNs for the remote file designate the stations that can be attached to this remote file, since the number of users has been set to two. If a third user tries to access the remote-file program, COMS initiates another copy of the program and uses RSN 1 for its own record keeping. Then, RSN 2 designates the station of the third user.

If the first user decides to close his or her window, COMS causes the remote-file program to receive an EOF indication. This indicates that the program of the user needs to check the LASTSTATION file attribute to determine on which RSN the EOF was given. In this case, the second user is still attached to the remote file through RSN 3. Therefore, the program might clean up the local storage for RSN 2, but the program should not terminate because there is still a user attached to it. When the second user closes, the program receives an EOF indication for RSN 3, and the program might clean up the storage for RSN 3. Because this is the last station associated with this copy of the program, COMS sends an EOF indication on RSN 1 to indicate that the program should go to EOJ. This EOF indication is sent even though there is a user attached to another copy of the remote-file program (the third user, represented by RSN 2 in the second copy of the remote-file program).

Additional Programming Notes for Remote-File Windows

When communicating with COMS through remote-file windows, be sure to observe the precautions and limitations described in this section.

Designation of Input or Output Files

When you use remote-file windows, the usual kind of file you will be opening is an input/output file. However, you might want your program to open files that are input only or output only. In COMS, as in CANDE, you can open several output files. In declared remote-file programs, all output files are associated with the declared remote-file window established in the COMS Utility. You should declare only one input-capable file, and that should be associated with the established declared remote-file window.

In dynamic remote-file programs, all output files are associated with the same dynamic window and the first input-capable remote file is also associated with this window. All subsequent input-capable files are assigned different dynamic windows.

Tanking and Multiuser Remote-File Windows

If you are writing a program for a multiuser remote-file window, you should set the TANKING file attribute to SYNC or ASYNCH. When you use either of these values, the output for the remote file is tanked, and the remote file program continues after the remote file closes. The default value for TANKING is NONE. Be aware that if this attribute is set to NONE, a user whose terminal is on local or on another window can prevent all other users of this window from receiving output. If the attribute is set to NONE and the TIMELIMIT file attribute is used, COMS gives a timelimit exception when the number of outstanding characters for a station exceeds 2400 characters. This range of characters includes data comm headers.

Remote-file programs written to run under COMS must not write to the file until the processing of the OPEN statement is completed. To make sure that this happens, always wait until an input is received for a read operation. If necessary, use notification text to provide this input on a declared remote-file window.

Exception Handling

When you are writing remote-file programs for use in the COMS environment, use exception handling for read and write operations. Your program should always check both read and write operations for the presence of EOF exceptions. When an EOF exception is found, the program should check its LASTSTATION file attribute to find out what to do.

COMS gives a time limit exception if TANKING is set to NONE and a TIMELIMIT attribute is used when the number of outstanding characters for a station exceeds the maximum limit for the specified time limit. If a broadcast operation is done, COMS gives a time limit exception if output could not be sent to any of the stations in the remote file.

If the attribute is set to NONE and the file attribute TIMELIMIT is used, COMS suspends output to the file when the number of outstanding characters for a station exceeds the limit of 2,400 characters, including data comm headers. When the time limit expires, COMS gives a time limit exception to the program attempting the write operation.

Appendix A
Tables of Values and Mnemonics

The following tables contain the values and mnemonics your program looks for in response to the execution of various program statements. The tables give quick-reference information for the Function Status and Status Value fields of the input header, the Status Value field of the output header, result messages for service function calls, and service function security category values and mnemonics.

Input Header Function Status Field Values and Mnemonics

Different syntax is used to access mnemonics in different languages. RPG uses the mnemonic as a figurative constant, which is the name preceded by an asterisk (*). In Pascal, mnemonics are context-sensitive identifiers.

Function Status fields passed to MARC also contain additional information in bits [38:19]. Therefore, any processing items to be placed in the MARCINPUT agenda should handle messages with nonzero values in this field. The values within this field are to be used solely for communication between COMS and MARC, and they might change on any release without notification.

Table A–1 shows the values and mnemonics of the input header Function Status field.

Table A–1. Input Header Function Status Field Values and Mnemonics

COBOL/ALGOL Value	RPG Mnemonic	Pascal Mnemonic	Description
0	NULFNCTN	NULLFNINDX	The field value is zero.
–01	CTLMSG	CTLMSG	MARC message.
–02	NOWNDW	NOWNDW	No window is available.
–03	LOST	LOST	No destination is available to receive messages.
–04	BADTCODE	BADTCODE	The message has been routed to the default agenda because one of the following two conditions has occurred: • COMS has detected an alphanumeric trancode that has not been defined to COMS. • COMS has detected a trancode beginning with an alphanumeric character followed by a special character or characters.
–05	NOTCODE	NOTCODE	A trancode beginning with a nonalphanumeric character has been found. The message will be routed according to the default agenda.

continued

Table A–1. Input Header Function Status Field Values and Mnemonics (cont.)

COBOL/ALGOL Value	RPG Mnemonic	Pascal Mnemonic	Description
–06	CTLNOWDW	CTLNOWDW	A controlled message is pending without an available window.
–07	CTLTOMCS	CTLTOMCS	A controlled message has been sent to the MCS.
–08	ONMARC	ONMARC	The MARC window is now available.
–09	SECTYERR	SECERR	A process-security error has occurred because the trancode associated with the incoming message is not allowed for this station or session.
–10	NOITEM	NOITEM	An attempt was made to apply a processing-item list to a message, but an item of the list was not found. As a result, the message is sent to the default agenda.
–11	TRDIVRT	TRDIVRT	A message has been diverted from a dialogue that is in transaction mode.
–12	GOODDEL	GOODDELVY	Delivery of the message is confirmed.
–13	BRK	BRK	A break condition has caused output from a direct window to be discarded.
–14	BADDELVY	BADDELVY	For a CP 2000 station, rejection of a message for which delivery confirmation was requested.
–15	BADDATA	BADDATA	For a CP 2000 station, rejection of a message for which delivery confirmation was not requested.

continued

Table A–1. **Input Header Function Status Field Values and Mnemonics** (cont.)

COBOL/ALGOL Value	RPG Mnemonic	Pascal Mnemonic	Description
–16	OPNOTEXT	OPENNOTEXT	When you define a direct window in the COMS Utility and do not add open-notification text, COMS returns this value upon each request for an open.
–17	ONNOTEXT	ONNOTEXT	When you define a direct window in the COMS Utility and do not add on-notification text, COMS returns this value upon each request for an on.
–18	BADTPDEL	BADTPDELVY	Delivery confirmation has been requested for a TP-to-TP message. This value notifies the sending program that the message has not been delivered.
–21	CTLRMSG	CTLRMSG	Control message.
–22	OUTMSG	OUTMSG	Output message.
–23	RTNMSG	RTNMSG	Return message.
–24	RMTFILE	RMTFILE	Remote file activity.
–25	CLOSMARC	CLOSMARC	Close MARC window.
–26	STSECMSG	STARTSECMSG	Start security message.
–27	NOSECMSG	STOPSECMSG	Stop security message.
–30	PREVATT	ALREADYATT	Successful attachment; station was already attached.
–31	TERMATT	ATTACHED	Successful attachment; station was attached through an enable statement.

continued

Table A–1. Input Header Function Status Field Values and Mnemonics (cont.)

COBOL/ALGOL Value	RPG Mnemonic	Pascal Mnemonic	Description
–32	WAITBUSY	WAITNOTBUSY	Attachment pending; waiting for station to become not busy.
–33	WAITATT	WAITATT	Attachment pending; waiting for physical attachment.
–34	DIALOK	DIALOUTOK	The requested dial-out over a modem was completed.
–35	ATTPEND	PENDINGATT	An attachment request was offered; waiting for a match from the other host.
–39	NOHOST	NOHOST	COMS made attachment offer; other host not available. COMS cancels pending attach.
–40	PROTERR	PROTOCOLERR	Failure during request to attach; protocol error.
–41	NORSRCS	NORESOURCES	Failure during request to attach; no resources available.
–42	NOTDEF	NOTDEFINED	Failure during request to attach; station not defined.
–43	INUSE	INUSE	Failure during request to attach; station in use.
–44	NOTATT	NOTATT	Failure during request to attach; station not physically attached.
–45	BADCALL	CALLFAILED	Failure during request to attach; the BNA *ESTABLISH CALL* command failed.
–46	NOTAVAIL	NOTAVAIL	Failure during request to attach; station not available.

continued

Table A–1. Input Header Function Status Field Values and Mnemonics (cont.)

COBOL/ALGOL Value	RPG Mnemonic	Pascal Mnemonic	Description
–47	BADDIALI	BADDIALINIT	COMS was unable to initiate the requested dial-out.
–48	DIALINC	DIALINCOMPLETE	COMS was unable to complete the requested dial-out.
–50	NOREQ	NOREQUEST	Successful detachment; the window was closed.
–51	RETAINED	RETAINED	Successful logical detachment; the CP 2000 terminal gateway retained the physical attachment.
–52	NORTAIND	NOTRETAINED	Successful logical and physical detachment from a CP 2000 station.
–53	DIALDISC	DIALDISCONNECT	Successful detachment from an NSP station that had been attached through a modem.
–60	SHUTDOWN	SHUTDOWN	Program is asked to terminate because COMS is shutting down.
–61	DISABLED	DISABLED	Program is asked to terminate because a DISABLE entity command was entered. The entity is the COMS entity (for example, *DISABLE <entity type> <entity name>*) that caused program termination. This entity could also be a program, a window, or a database.
–62	TOOMANY	TOOMANY	Program is asked to terminate because activity was reduced and current copies exceeded minimum copies.

continued

Table A–1. Input Header Function Status Field Values and Mnemonics (cont.)

COBOL/ALGOL Value	RPG Mnemonic	Pascal Mnemonic	Description
–100	(not applicable)	(not applicable)	An invalid input message key was detected by the SDF formlibrary. An input error has occurred, and the application program should perform error handling. Typically, this error indicator is received on the first input when SDF is used. SDF takes no action on the input message.

Input Header Status Value Field Values and Mnemonics

Different syntax is used to access mnemonics in different languages. RPG uses the mnemonic as a figurative constant, which is the name preceded by an asterisk (*). In Pascal, the mnemonics are context-sensitive identifiers.

Table A–2 shows the input header Status field values and mnemonics.

Table A–2. Input Header Status Field Values and Mnemonics

COBOL/ALGOL Value	Pascal/RPG Mnemonic	Description
0	COMSOK	Successful incoming message.
20	NORQST	The station designated for a dynamic attachment or detachment is invalid or unknown, the specified hostname does not match, or the host system has denied access to the station.
80	SKIPNEXT	A batch synchronized recovery transaction has aborted outside of the transaction state. The program should skip the returned successful message, skip the next failed message, and then continue.
89	BADMSGSZ	The input message has been truncated because it was larger than the input message area provided in the RECEIVE statement.
90	BADTR	A batch synchronized recovery transaction has aborted after mid-transaction. The transaction caused the program to abort and must not be reprocessed.
91	NODATA	There is no message to be received by your program.
92	OKRECOV	Successful incoming message is being resubmitted for synchronized recovery.

continued

Table A–2. **Input Header Status Field Values and Mnemonics** (cont.)

COBOL/ALGOL Value	Pascal/RPG Mnemonic	Description
93	BADPREV	Successful incoming message was submitted once and caused the program to abort, so is being resubmitted again. If the message causes another abort, the originating station will be notified and the message discarded.
94	BADDEST	A processing item has attempted to reroute an input message and one of the following conditions has occurred. • An invalid program designator was detected while a processing item attempted to route a message from one application program to another. • An invalid station designator was detected while a processing item attempted to send a message.
95	BADAGND	An invalid agenda designator was detected while a processing item was attempting to reroute an input message.
97	NODEST	The destination program of an application program message was not enabled while a processing item was attempting to reroute an input message.
99	GOEOT	COMS has directed the application program to terminate. The Function Status field of the input header will show one of three values (–60, –61, –62) or corresponding mnemonics, depending on the reason for the termination.
100	DIALING	The destination station for an attempted dynamic attachment is already attached to another program.
101	BADDIAL	An attempted dial-out over a modem failed.

continued

Table A–2. Input Header Status Field Values and Mnemonics (cont.)

COBOL/ALGOL Value	Pascal/RPG Mnemonic	Description
102	BADDCNCT	An attempted dynamic attachment from a dial-out station failed.
103	BADENBLE	An attempt to add a window to a window list of a station list failed.

Output Header Status Value Field Values and Mnemonics

Different syntax is used to access mnemonics in different languages. RPG uses the mnemonic as a figurative constant, which is the name preceded by an asterisk (*). In Pascal, the mnemonics are context-sensitive identifiers.

Table A–3 shows the output header Status Value field values and mnemonics.

Table A–3. Output Header Status Value Field Values and Mnemonics

COBOL/ALGOL Value	Pascal/RPG Mnemonic	Description
0	COMSOK	The outgoing message has been accepted for transmission.
86	POINVDST	The specified TP destination for the protected output message is invalid; this TP is associated with a window that is either not protected or is protected by a different protected output file. Used only by applications generated by LINC Release 14.
87	POETR	A successful transaction commit has been reached; therefore, no further protected output is allowed for this transaction. Used only by applications generated by LINC Release 14.
88	POTOTP	A protected TP-to-TP message has already been generated; only one such message is allowed for each protected transaction. Used only by applications generated by LINC Release 14.
89	BADMSGSZ	The length of the output message specified in the Text Length field of the output header was larger than the output message area.
92	OKRECOV	The outgoing message has been accepted, but will be discarded because COMS is processing a recovery transaction.
94	BADDEST	One of the following two conditions has occurred:

continued

Table A–3. Output Header Status Value Field Values and Mnemonics (cont.)

COBOL/ALGOL Value	Pascal/RPG Mnemonic	Description
		• The outgoing message was rejected because the station designator for the destination was invalid. • An invalid program designator was detected while a processing item attempted to route a message from one application program to another.
95	BADAGND	The outgoing message was rejected because the agenda designator in the Agenda Designator field of the output header was invalid.
96	AGNDQUIT	The outgoing message was prematurely stopped by a processing item. This error number is intended to be used as an error signal from the processing item to the calling program.
97	NODEST	One of the following conditions has occurred: • The outgoing message was rejected because the station is not open to this window. • The destination program of an application program message is not enabled. • The destination program and its associated database are disabled. • The outgoing message was rejected because it contained an invalid trancode and no default output agenda was specified.

continued

Table A–3. Output Header Status Value Field Values and Mnemonics (cont.)

COBOL/ALGOL Value	Pascal/RPG Mnemonic	Description
98	STOPPROC	One of the following conditions has occurred: • A security error was detected when a program routed a request from one application program to another program specified by the transaction code in the message. • A processing item in the processing-item list associated with the requested agenda was not available. COMS stopped processing the processing-item list when the unavailable processing item was encountered.
104	MSGTOOBG	The outgoing message was rejected because it was too large to be processed.
105	DBDSBLED	The database associated with the destination program is disabled.
109		The window and database of the destination program are disabled.
110		The connection establishment process is still in progress because of a prior ENABLE TERMINAL request. The outgoing message has been rejected.

Service Function Result Values and Mnemonics

Table A–4 shows the service function result value integers for ALGOL, COBOL, and RPG, the service function result value mnemonics for Pascal, and a description of what occurs when these integers or mnemonics are executed.

Table A–4. Service Function Result Values and Mnemonics

COBOL/ALGOL Value	Pascal/RPG Mnemonic	Description
0	OK	Function completed successfully.
1	FUNCFAIL	Error. Function failed because of invalid name or designator.
2	INVDESG	Designator error. Invalid designator.
3	SHORTARRAY	Array size error. Supplied array was too short to house the information.
4	NOINSTDATA	No-data error. The requested installation data was not present in the configuration file.

Service Function Security Category Values and Mnemonics

Different syntax is used to access service function security category values and mnemonics in different languages. RPG encloses the mnemonic in single quotation marks (alpha literal). In Pascal, the mnemonics are context-sensitive identifiers.

Table A–5 shows the service function security category values or mnemonics for ALGOL, Pascal and RPG, and COBOL74 in columns 1, 2, and 3, respectively.

Table A–5. Service Function Security Category Values and Mnemonics

ALGOL Value	Pascal/RPG Mnemonic	COBOL74 Name
1	STATION	STATION
2	USERCODE	USERCODE
3	AGENDA	AGENDA
4	PROGRAM	PROGRAM
5	SECURITY	SECURITY
8	SECTYCAT	SECURITY-CATEGORY
9	DEVICE	DEVICE
10	STNLIST	STATION-LIST
11	DVCLIST	DEVICE-LIST
12	WINDOW	WINDOW
13	DATABASE	DATABASE
14	PROCITEM	PROCESSING-ITEM
15	PRITMLST	PROCESSING-ITEM-LIST
16	TRANCODE	TRANCODE
17	WNDWLST	WINDOW-LIST
18	LIBRARY	LIBRARY
19	CATLIST	CATEGORY-LIST
20	INSTDATA	INSTALLATION-DATA
41	INSTINT1	INSTALLATION-INTEGER-1
42	INSTINT2	INSTALLATION-INTEGER-2
43	INSTINT3	INSTALLATION-INTEGER-3
44	INSTINT4	INSTALLATION-INTEGER-4
45	INSTINTS	INSTALLATION-INTEGER-ALL
46	INSTSTR1	INSTALLATION-STRING-1

continued

Table A–5. Service Function Security Category Values and Mnemonics (cont.)

ALGOL Value	Pascal/RPG Mnemonic	COBOL74 Name
47	INSTSTR2	INSTALLATION-STRING-2
48	INSTSTR3	INSTALLATION-STRING-3
49	INSTSTR4	INSTALLATION-STRING-4
50	INSTHEX1	INSTALLATION-HEX-1
51	INSTHEX2	INSTALLATION-HEX-2
52	INSTLINK	INSTALLATION-DATA-LINK
61	QDEPTH	QUEUE-DEPTH
62	MSGCOUNT	MESSAGE-COUNT
63	LASTRESP	LAST-RESPONSE
64	AGGRRESP	AGGREGATE-RESPONSE
65	STATS	STATISTICS
71	DATE	DATE
72	TIME	TIME
81	MAXUSRCT	MAXIMUM-USER-COUNT
82	CURUSRCT	CURRENT-USER-COUNT
83	LSN	LSN
84	MIXNMBRS	MIXNUMBERS
85	VIRTTERM	VIRTUAL TERMINAL
86	SCREENSZ	SCREENSZ
95	LANGUAGE	LANGUAGE
120	CONVEN	CONVENTION

Appendix B
COMS Header Layout

Defining Input Header Information

Table B–1 lists the words, the COMS field names, and the Language field names associated with standard COMS input header layout.

Table B–1. Input Header Information

Word	COMS Field Name	Language Field Name
0	Program Designator	PROGRAMDESG 3.6 COBOL74 CD: SYMBOLIC QUEUE
1	Function Index	FUNCTIONINDEX 3.6 COBOL74 CD: SYMBOLIC SUB-QUEUE-1
	Function Status	FUNCTIONSTATUS
2	Usercode Designator	USERCODE 3.6 COBOL74 CD: SYMBOLIC SUB-QUEUE-2
3	Security Designator	SECURITYDESG 3.6 COBOL74 CD: SYMBOLIC SUB-QUEUE-3
4	(FIELDS)	FIELDS
	43:16 (not used)	
	31:02 (reserved)	
	29:01 Transparent	TRANSPARENT
	28:01 VT Flag	VTFLAG
	27:04 (reserved)	

continued

Table B–1. Input Header Information (cont.)

Word	COMS Field Name	Language Field Name
	23:23 (not used)	
	00:01 (reserved)	
5	Timestamp	TIMESTAMP 3.6 COBOL74 CD: MESSAGE TIME
6	Station Designator	STATION 3.6 COBOL74 CD: SYMBOLIC SOURCE
7	Text Length	TEXTLENGTH 3.6 COBOL74 CD: TEXT LENGTH
8	(not used)	
9	Status Value	STATUSVALUE 3.6 COBOL74 CD: STATUS KEY
10	Message Count	MESSAGECOUNT 3.6 COBOL74 CD: MESSAGE COUNT
	Restart	RESTART
11	Agenda Designator	AGENDA
12	SDF Plus Information	SDFINFO
13	SDF Plus Form Record	SDFFORMRECNUM
14	SDF Plus Transaction	SDFTRANSNUM
15	Retries to Go	RETRIESLEFT
16–31	(not used)	

continued

Table B–1. Input Header Information (cont.)

Word	COMS Field Name	Language Field Name
32–N	Conversation Area	ALGOL: (user-defined) COBOL74: CONVERSATION AREA Pascal: CONVERSATIONAREA RPG: CONVERSATION

Defining Output Header Information

Table B–2 lists the words, the COMS field names, and the Language field names associated with standard COMS output header layout.

Table B–2. Output Header Information

Word	COMS Field Name	Language Field Name
0	Destination Count	DESTCOUNT 3.6 COBOL74 CD: DESTINATION COUNT
1	Text Length	TEXTLENGTH 3.6 COBOL74 CD: TEXT LENGTH
2	Status Value	STATUSVALUE 3.6 COBOL74 CD: STATUS KEY
3	(FIELDS)	FIELDS
	47:16 Carriage Control	
	47:08 (Advancing Amount)	
	39:01 (not used)	
	38:01 (Before or After)	
	37:03 (Action)	
	34:01 (New Page)	
	33:01 (No Carriage Return)	
	32:01 (No Line Feed)	
	31:05 (reserved)	
	26:01 Transparent	TRANSPARENT
	25:01 VT Flag	VTFLAG

continued

Table B–2. Output Header Information (cont.)

Word	COMS Field Name	Language Field Name
	24:01 Delivery Confirmation Flag	CONFIRMFLAG
	23:24 Delivery Confirmation Key	CONFIRMKEY
4	Destination Designator	DESTINATIONDESG 3.6 COBOL74.CD: DESTINATION TABLE
5	Next Input Agenda Designator	NEXTINPUTAGENDA
6	(TOGGLES)	TOGGLES
	47:45 (not used)	
	02:01 Casual Output	CASUALOUTPUT
	01:01 Set Next Input Agenda	SETNEXTINPUTAGENDA
	00:01 Retain Transaction Mode	RETAINTRANSACTIONMODE
7–10	(not used)	
11	Agenda Designator	AGENDA
12	SDF Information	SDFINFO
13	SDF Plus Form Record	SDFFORMRECNUM

continued

Table B–2. Output Header Information (cont.)

Word	COMS Field Name	Language Field Name
14	SDF Plus Transaction	SDFTRANSNUM
15	Retries to Go	RETRIESLEFT
16–31	(not used)	
32–N	Conversation Area	ALGOL (user-defined)
	COBOL74:	CONVERSATION AREA
	Pascal:	CONVERSATIONAREA
	RPG:	CONVERSATION

Appendix C
Sample COBOL74 Programs

This appendix provides three sample COBOL74 programs that demonstrate

- Using a default agenda with minimal features of the COMS direct-window interface
- Applying a Screen Design Facility (SDF) form as a processing item to messages
- Routing by trancode with different SDF forms for sending and receiving messages

(A COMS interface to SDF is not currently available through the ALGOL, Pascal, or RPG programming languages.)

To write programs with messages routed by COMS, you must be familiar with COMS as a whole. You should understand the information presented in this guide and be familiar with the information explained in the following documents:

- The *COMS Configuration Guide*
- The *COMS Operations Guide*
- The *SDF Operations and Programming Guide*
- The *COBOL ANSI-74 Reference Manual,* Volumes 1 and 2

Also, if you plan to use SDF extensively, you should know how to create SDF forms and formlibraries before you try to write application programs that use them.

The first sample program in this appendix, Program 1, shows a simple way to use the COMS direct-window interface with a default agenda. The second sample program modifies the first by adding an SDF form as a processing item. Program 3 modifies the second example by implementing trancode routing and a second SDF form.

The actual program listings for each sample program are proceeded by the following:

- A brief introductory description of the program
- A discussion of the COMS and SDF entities you must define for the program
- Guidelines for the declarations and routines used in the program
- Procedures for running and terminating the program

Note: *The procedures for running and terminating the sample programs assume that your station has been defined to COMS.*

Program 1: Using a Default Agenda

This is an echo program that returns a duplicate of each message a station sends. To use this program, you need to create a default agenda that specifies the echo program as the destination for messages received and sent.

Defining COMS Entities for Program 1

Because this is a direct-window program, it must be initiated by COMS. You cannot start this program with the CANDE *RUN* or *EXECUTE* commands. To cause COMS to initiate the program and route messages for it, use the COMS Utility menus or commands, as described in the *COMS Configuration Guide,* to define the following COMS entities in the order presented:

1. Define a window that this program will access. Supplying notify-on and notify-open text for the window might help you understand how the program operates within COMS.
2. Define the program name to COMS.
3. Define a default agenda whose destination is this program, and associate this agenda with the window you defined in step 1.

   (You do not need to define any processing items for use with this agenda, because Program 1 does not apply processing items to the messages that it sends or receives.)

Declarations and Routines for Program 1

Program 1 shows the following basic, minimal declarations and routines needed for an application program that uses COMS direct windows:

- A message area
- An input header
- An output header
- A routine that initializes the program to run under COMS
- A main processing loop that executes the SEND and RECEIVE statements

The first three components are explained in Section 3, "Communicating with COMS through Direct Windows." A main processing loop that goes to end of job (EOJ) if the Status Value field of the input header contains a value of 99 is considered a standard termination routine for a program using the direct-window interface. Minimally, you should use the COBOL74 *MOVE* statement to specify values for the following fields of the output header before executing the SEND statement:

1. Move the value 1 to the Destination Count field of the output header to specify a single destination for outgoing messages.
2. Move a value (a station or program designator) into the Destination field of the output header to indicate a particular destination for outgoing messages.

3. Move a value representing the length of the output message text into the Text Length field of the output header.

Using Program 1

After you have defined the required entities with the COMS Utility, you can start Program 1 by issuing an *?ON <window name>* command from your station, followed by the first input message.

You can start the program automatically, without entering an input message, if you specify a notify-open action for the window. You can specify a notify-open action either with text or without text. If you do not specify text, COMS sends a message with a length of 0 (zero) to the program when you open the window.

Use the *?DISABLE PROGRAM <program name>* command to terminate Program 1.

If you want the program to terminate automatically, specify a termination time limit other than 0 (zero) when you use the COMS Utility to define the program. You can also use the COMS *?WINDOWS, ?CLOSE, ?SUSPEND,* and *?RESUME* commands to manipulate this program. Refer to the *COMS Operations Guide* for more information about these COMS commands.

Program 1 Listing

Following is the listing of Program 1:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
Ø1 COMS-NAME                  PIC X(Ø72).
Ø1 COMS-MESSAGE-AREA.
    Ø5 COMS-MESSAGE           PIC X(192Ø).
COMMUNICATION SECTION.
   INPUT HEADER COMS-IN.
   OUTPUT HEADER COMS-OUT.
PROCEDURE DIVISION.
CONTROLLER-SECTION             SECTION  1.
CONTROLLER.
    PERFORM START-UP-SECTION.
    PERFORM PROCESS-IT-SECTION
       UNTIL STATUSVALUE OF COMS-IN = 99.
    STOP RUN.

START-UP-SECTION               SECTION  5Ø.
START-UP.
    MOVE ATTRIBUTE NAME OF ATTRIBUTE EXCEPTIONTASK OF
        ATTRIBUTE EXCEPTIONTASK OF MYSELF TO COMS-NAME.
    CHANGE ATTRIBUTE TITLE OF "DCILIBRARY"
       TO COMS-NAME.
PROCESS-IT-SECTION             SECTION 1.
PROCESS-IT.
    RECEIVE COMS-IN MESSAGE INTO COMS-MESSAGE.
    IF STATUSVALUE OF COMS-IN NOT = 99
    IF NOT FUNCTIONSTATUS OF COMS-IN < Ø
       MOVE 1                      TO DESTCOUNT OF COMS-OUT
       MOVE STATION OF COMS-IN     TO DESTINATIONDESG OF COMS-OUT
       MOVE TEXTLENGTH OF COMS-IN  TO TEXTLENGTH OF COMS-OUT
       SEND COMS-OUT FROM COMS-MESSAGE
       GO TO PROCESS-IT.
END-OF-JOB.
       STOP RUN.
```

Program 2: Using an SDF Form Processing Item

This program is a modification of Program 1. It shows how to use a COMS agenda to apply an SDF form as a processing item to messages received and sent. The program shows the most basic COMS and SDF concepts and COBOL74 language extensions needed to write a COBOL74 program that uses a COMS direct window and an SDF form.

Defining COMS and SDF Entities for Program 2

Before writing a program like this, you must have created a form in a formlibrary using screens in the SDF system. For this particular program, you can use default values in the fields of the formlibrary Definition screen and the Form Definition screen, except for library sharing. Specifically, you must choose SHAREDBYRUNUNIT for the Library Sharing option on the formlibrary Definition screen. (Refer to the *A Series Screen Design Facility (SDF) Operations and Programming Guide* for detailed instructions on creating SDF entities.)

Next, you must define the COMS and SDF entities that are needed to perform direct-window message processing with an SDF form. Use the COMS Utility menus or commands to define the following entities in the order presented:

1. Define a COMS library and assign the name of the formlibrary generated by SDF as the library title. Include the appropriate usercode and pack as part of the title.
2. Define a COMS processing item that is associated with the formlibrary already defined. The Actual Name attribute should be PROC_ITEM. The library name should be the name of the library that you defined in step 1.
3. Define a COMS processing-item list that includes the processing item already defined.
4. Define a COMS direct window.
5. Define the COMS program name.
6. Define a default output agenda that names the direct window already defined and the destination program. Place the processing-item list you defined in the agenda.

Declarations and Routines for Program 2

Like Program 1, Program 2 requires the following minimal declarations and routines:

- A message area
- An input header
- An output header
- A routine that initializes the program to run under COMS

Additionally, Program 2 can use the same standard COMS termination routine as Program 1.

Program 2, however, invokes the SDF formlibrary, and the SDF form is applied to messages as a processing item. To incorporate an SDF form into this scheme, use the following declarations and routines, in the order presented:

1. Use the SPECIAL-NAMES paragraph in the Configuration Section to specify the program name and the dictionary that stores the SDF formlibrary.
2. Declare the SDF formlibrary as a level 01 record, using the following syntax:

   ```
   Ø1 <generated  formlibrary name>   FROM DICTIONARY.
   ```

   This declaration invokes the formlibrary. The compiler copies in all the record descriptions for the formlibrary, causing a break in the sequence range of the program.
3. Declare a level 01 record for the Agenda Name using the following data type:

Agenda Name: PIC X(17) VALUE "<agenda name>".

4. Declare a level 01 record for the Agenda Designator using the following data type:

   Agenda Designator: PIC S9(11) USAGE BINARY.

5. In the initialization routine, obtain an agenda designator by calling the GET_DESIGNATOR_USING_NAME service function. Be sure to include the service function call after the ENABLE INPUT statement.

6. In the main processing loop, let the RECEIVE statement use the SDF form as a message area with the following syntax:

```
RECEIVE <input header name> MESSAGE INTO <SDF form name>.
```

7. Move the agenda designator into the Agenda Designator field of the output header to specify the message destination for the SEND statement.

8. Move the FORM-KEY, which identifies the desired SDF form to the SDF system, into the Conversation Area field of the output header.

9. Let the SEND statement use the SDF form as the message area with the following syntax:

```
SEND <output header name> FROM <SDF form name>.
```

Using Program 2

When COMS responds to the *?ON <window name>* command by placing the station in the direct window specified in the command, you must transmit an initial message from the station in order to receive the first SDF form. Once you receive the SDF form, the program echoes each message you transmit, because the destination defined for the agenda is the echo program itself.

Use the *?DISABLE PROGRAM <program name>* command or a termination time limit other than 0 (zero) to terminate Program 2.

When you use a program that invokes SDF forms, you might need to recompile the program whenever you regenerate the SDF formlibrary, if you have made changes to the formlibrary.

Program 2 Listing

Following is the listing of Program 2:

```
$SET LIST WARNSUPR
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
     SPECIAL-NAMES.
     DICTIONARY IS "SCREENDESIGN",
     PROGRAM-NAME IS "ECHOWITHSDF".
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 Ø1 COMS-NAME                  PIC X(Ø72).
 Ø1 SDF formlibrary               FROM DICTIONARY.
******************************************************************
*  THE FOLLOWING LINES OF CODE ARE GENERATED BY THE SYSTEM:      *
******************************************************************
*--DICTIONARY                                                    *
*--DICTIONARY FORMLIST< SDF formlibrary >.                       *
*   SDFFORM.                                                     *
*     ACTION PIC  X(1Ø).                                         *
*     CHOICE PIC  X(2Ø).                                         *
*     MESSAGEAREA PIC X(3Ø).                                     *
******************************************************************
 Ø1 SDF-AGENDA-NAME            PIC X(17) VALUE "SDFAGENDA".
 77 SDF-AGENDA-DESIGNATOR      USAGE REAL.
 77 SDF-CALL-ERROR             PIC S9(11) USAGE BINARY.
 COMMUNICATION SECTION.
    INPUT HEADER COMS-IN.
    OUTPUT HEADER COMS-OUT
      CONVERSATION AREA.
      Ø2 COMS-OUT-CONVERSATION   REAL.
 PROCEDURE DIVISION.
 CONTROLLER-SECTION             SECTION  1.
 CONTROLLER.
     PERFORM START-UP-SECTION.
     PERFORM PROCESS-IT-SECTION
        UNTIL STATUSVALUE OF COMS-IN = 99.
     STOP RUN.
```

```
START-UP-SECTION               SECTION  5Ø.
START-UP.
    MOVE ATTRIBUTE NAME OF ATTRIBUTE EXCEPTIONTASK OF
       ATTRIBUTE EXCEPTIONTASK OF MYSELF TO COMS-NAME.
    CHANGE ATTRIBUTE TITLE OF "DCILIBRARY"
       TO COMS-NAME.
    ENABLE INPUT COMS-IN KEY "ONLINE".
    CALL "GET_DESIGNATOR_USING_NAME IN DCILIBRARY"
       USING  SDF-AGENDA-NAME
              ,VALUE AGENDA
              ,SDF-AGENDA-DESIGNATOR.
       GIVING SDF-CALL-ERROR.
PROCESS-IT-SECTION             SECTION 1.
PROCESS-IT.
    RECEIVE COMS-IN MESSAGE INTO SDFFORM.
    IF STATUSVALUE OF COMS-IN NOT = 99
    IF NOT FUNCTIONSTATUS OF COMS-IN < Ø
      MOVE 1                        TO  DESTCOUNT OF COMS-OUT
      MOVE STATION OF COMS-IN       TO  DESTINATIONDESG OF COMS-OUT
      MOVE TEXTLENGTH OF COMS-IN    TO  TEXTLENGTH OF COMS-OUT
      MOVE SDF-AGENDA-DESIGNATOR    TO  STATUSVALUE OF COMS-OUT
      MOVE FORM-KEY(SDFFORM)        TO  COMS-OUT-CONVERSATION
       SEND COMS-OUT FROM SDFFORM.
END-OF-JOB.
```

Program 3: Routing by Trancode

This echo program is a modification of Program 2. It shows routing by trancode and the use of two SDF forms as processing items.

Defining COMS and SDF Entities for Program 3

Before writing a program like this, you must have created two forms in a formlibrary using screens in the SDF system. To use routing by trancode with an SDF form, you must specify the following values when completing these SDF screens:

1. On the formlibrary Definition screen, make the following choices:
   - Choose SHAREDBYRUNUNIT for the Library Sharing option.
   - Answer Y, for yes, to the question, "Will this library process forms based on message keys?"
   - Answer Y, for yes, to the question, "Will the Module Index be used for routing with COMS interface?"
2. After creating the forms with the Form Definition screen, use the Additional Field Definition screen to define a message key for each form. For each form you are defining, you need to define a message key at the same offset and with the same length. (Use the first field on each form for the message keys.) You also need to define default values for the message keys.

3. The message key in SDF corresponds to the trancode in COMS. The entity referred to as the COMS index or module index in SDF corresponds to the module function index (MFI) in COMS. The FORM-KEY used in a program like Program 3 identifies to SDF which form the program is invoking.

Next, all of the COMS entities defined for Program 2 need to be defined for Program 3. These include the COMS library, a processing item and processing-item list, a direct window, the program name, and an agenda.

Program 3 can use the same window as Program 2, but it requires a separate agenda, processing item, and processing-item list. Also, you must define two trancodes for routing the two SDF forms to the program.

Use the COMS Utility to assign the window and agenda to each trancode. You also need to assign an MFI, which is an integer value associated with the trancode used in the application program to reference the trancode.

Declarations and Routines for Program 3

All the guidelines for program declarations and routines presented for Programs 1 and 2 apply to Program 3 as well. However, the clause SAME RECORD AREA has been added to the level 01 record declaration for the name of the generated formlibrary. In a program that uses more than one SDF form, the SAME RECORD AREA clause makes every form a redefinition of the other forms.

In Program 3, the second form description for an SDF form redefines the form description for the first SDF form. Since the program does not know what kind of message it will receive, it must be able to redefine the form that serves as the message area.

In the main processing loop, an IF-THEN-ELSE statement is used to determine which SDF form to invoke based on the MFI value. After the program receives a message, COMS places a value for the MFI (if defined) into the COMS-in-Function-Index field of the input header. In the case of Program 3, entering one of the two defined trancodes from the terminal causes COMS to place the appropriate MFI value in the Function Index field of the input header. The program can then execute the corresponding branch of the IF-THEN-ELSE statement.

Before executing the SEND statement, the program moves the FORM-KEY that references the required SDF form into the Conversation Area field of the output header. The following syntax should be used:

```
MOVE FORM-KEY(SDFFORM) TO COMS-OUT-CONVERSATION
```

Using Program 3

When COMS responds to the *?ON <window name>* command by placing the station in that direct window, enter from your terminal one of the trancodes you defined to invoke one of the two SDF forms used in this example.

Use the *?DISABLE PROGRAM <program name>* command or specify a termination time limit other than 0 (zero) for the program to terminate Program 3.

Note: *When using a program that invokes SDF forms, you must recompile the program whenever you regenerate the SDF formlibrary.*

Program 3 Listing

Following is a listing of Program 3:

```
$SET LIST WARNSUPR
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
     SPECIAL-NAMES.
     DICTIONARY IS "SCREENDESIGN",
     PROGRAM-NAME IS "ECHOWITHSDFTBR".
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 COMS-NAME                  PIC X(072).
 01 SDF form library        FROM DICTIONARY; SAME RECORD AREA.

*******************************************************************
*  THE FOLLOWING LINES OF CODE ARE GENERATED BY THE SYSTEM:       *
*******************************************************************
*--DICTIONARY                                                     *
*--DICTIONARY FORMLIST < SDFFORM LIBRARY >.                       *
*   SDFFORM.                                                      *
*     ACTION PIC X(10).                                           *
*     CHOICE PIC X(20).                                           *
*     MESSAGEAREA PIC X(30).                                      *
*   SDFFORM2 REDEFINES SDFFORM.                                   *
*     ACTION2 PIC X(10).                                          *
*     NAME2 PIC X(30).                                            *
*     TITLE2 PIC X(20).                                           *
*******************************************************************
 01 SDF-AGENDA-NAME            PIC X(17) VALUE "SDFTBRAGENDA".
 77 SDF-AGENDA-DESIGNATOR      USAGE REAL.
 77 SDF-CALL-ERROR             PIC S9(11) USAGE BINARY.
 COMMUNICATION SECTION.
    INPUT HEADER COMS-IN.
    OUTPUT HEADER COMS-OUT
      CONVERSATION AREA.
      02 COMS-OUT-CONVERSATION    REAL.
 PROCEDURE DIVISION.
 CONTROLLER-SECTION             SECTION  1.
 CONTROLLER.
     PERFORM START-UP-SECTION.
     PERFORM PROCESS-IT-SECTION
        UNTIL STATUSVALUE OF COMS-IN = 99.
     STOP RUN.
```

```
START-UP-SECTION               SECTION  5Ø.
START-UP.
    MOVE ATTRIBUTE NAME OF ATTRIBUTE EXCEPTIONTASK OF
       ATTRIBUTE EXCEPTIONTASK OF MYSELF TO COMS-NAME.
    CHANGE ATTRIBUTE TITLE OF "DCILIBRARY"
       TO COMS-NAME.
    ENABLE INPUT COMS-IN KEY "ONLINE".
    CALL "GET_DESIGNATOR_USING_NAME" IN DCILIBRARY"
       USING  SDF-AGENDA-NAME
              ,VALUE(AGENDA)
              ,SDF-AGENDA-DESIGNATOR
       GIVING SDF-CALL-ERROR.

PROCESS-IT-SECTION             SECTION 1.
PROCESS-IT.
    RECEIVE COMS-IN MESSAGE INTO SDFFORM.
    IF STATUSVALUE OF COMS-IN NOT = 99
    IF NOT FUNCTIONSTATUS OF COMS-IN < Ø
       MOVE 1                     TO  DESTCOUNT OF COMS-OUT
       MOVE STATION OF COMS-IN    TO  DESTINATIONDESG OF COMS-OUT
       MOVE TEXTLENGTH OF COMS-IN TO  TEXTLENGTH OF COMS-OUT
       MOVE SDF-AGENDA-DESIGNATOR TO  STATUSVALUE OF COMS-OUT
       IF FUNCTIONINDEX OF COMS-IN = 1 THEN
          MOVE FORM-KEY(SDFFORM)   TO  COMS-OUT-CONVERSATION
          SEND COMS-OUT FROM SDFFORM
    ELSE
       IF FUNCTIONINDEX OF COMS-IN = 2 THEN
          MOVE FORM-KEY(SDFFORM2)  TO  COMS-OUT-CONVERSATION
          SEND COMS-OUT FROM SDFFORM2.
END-OF-JOB.
```

Appendix D
Sample Processing Items

This appendix provides two sample processing items and the set of global declarations that these processing items require. All these declarations should precede the code for the processing items, and therefore are listed first. The two sample processing items, TPTOMARC and STATUS_LINE, are described later in this appendix.

Commentary within the code provides additional explanation.

Global Declarations

There are six sets of global declarations that do the following:

- Set the TITLE parameter.
- Define the input header parameters.
- Define the output header parameters.
- Define the STATE parameters.
- Define the result action values.
- Define the function values.

The code in the following pages of this section corresponds to the global declarations.

Setting the TITLE Parameter

The following code sets the TITLE parameter, which is used by the TPTOMARC processing item. Note the sequence of steps described in the commentary.

```
 $ SET SHARING = SHAREDBYALL
BEGIN

%
%  First, the code declares a library that is used  to call COMS
%  service functions.
%

LIBRARY COMS_LIB(LIBACCESS = BYTITLE);

%
%  The code below declares the GET_NAME_USING_DESIGNATOR
%  service function entry point.
%

INTEGER PROCEDURE GET_NAME_USING_DESIGNATOR(STA_DESG, STA_NAME);
    VALUE STA_DESG;
    REAL STA_DESG:
EBCDIC ARRAY STA_NAME[Ø];
LIBRARY COMS_LIB;

%
%  The following code declares the GET_DESIGNATOR_USING_NAME service
%  function entry point.
%

INTEGER PROCEDURE GET_DESIGNATOR_USING_NAME(STA_NAME, MNEMONIC,
                                            STA_DESG);
  VALUE MNEMONIC;
  REAL STA_DESG
EBCDIC ARRAY STA_NAME[Ø];
LIBRARY COMS_LIB;
DEFINE STATION_MNEMONIC = 1#;
%
%  Next, the code declares the global data structures and the locks and
%  lock routines to manage them.
%

BOOLEAN
  TITLE_SET;            % This value is TRUE if the title of COMS_LIB
                        % has been set.

EBCDIC ARRAY            % Holds title of COMS_LIB.
  TTL[Ø:299];
```

```
%
%  The following example has only one lock, the TITLE lock (TTL).
%  When the title of COMS_LIB is set, the TITLE lock must be locked.
%

REAL
  TTL_OWNER,       % The PROCESSID of the owner of the TITLE lock.
  TTL_NUM;         % The PROCESSID of the last requestor of the TITLE
                   % lock.

EVENT
  TTL_EVENT;       % The event to PROCURE/LIBERATE if there is lock
                   % contention for the TITLE lock.

     %
     %  The following define causes the TITLE lock to be acquired.  The
     %  define uses a modified form of FIBLOCK, an operating
     %  system lock type used by the logical I/O to lock files.
     %  The modifications of FIBLOCK reduce overhead with high lock
     %  contention.
     %
     %  To ensure that locking TTL and identifying the owner of TTL
     %  either both occur or neither occur, the code disables external
     %  interrupts.
     %
     %
     %                          NOTE
     %
     %         The following  defines  use  the  DMALGOL
     %         constructs  DISALLOW  and  ALLOW to enter
     %         and exit control state.  If  the  defines
     %         are  used,  modify them only with extreme
     %         care.

DEFINE
     ACQUIRE_TTLLOCK =
       BEGIN
       DISALLOW;
       IF READLOCK(PROCESSID, TTL_NUM) NEQ Ø THEN
         DO
           PROCURE(TTL_EVENT)
         UNTIL
           READLOCK(-1, TTL_NUM) = Ø;
       TTL_OWNER:=PROCESSID;
       ALLOW;
       END#,
%
% The following define relinquishes the TITLE lock.  The define
% also disables external interrupts so that either of the
% following occurs:
%
% 1.  The TITLE lock is relinquished  and the owner of the TITLE
%     lock is reset.
%
% 2.  The TTL lock is not relinquished and the owner of the TITLE
%     lock is not reset.
%

RELINQUISH_TTLLOCK =
  BEGIN
  DISALLOW;
  TTL_OWNER:=Ø;
  IF READLOCK(Ø, TTL_NUM) NEQ PROCESSID THEN
    LIBERATE(TTL_EVENT);
  ALLOW;
  END#,

%
% The PROTECTION_HEADING define should be invoked once in every
% block that needs to acquire the TITLE lock.  This define
% declares a dummy label and an EPILOG procedure, then invokes
% the ACQUIRE_TTLLOCK define.  Therefore, this define must come
% after all declarations and before all statements in a block.
%
% The label is used to prevent programming errors by requiring
% the scope of the lock protection to be identified.  The EPILOG
% procedure is used to relinquish the TITLE lock if it has been
% acquired.
%
% The purpose of locking the TITLE lock in this way is to allow
% the COMS programs that invoke processing items to be
% discontinued while they are in processing item code.  As
% a result, the lock does not remain locked.  This procedure is
% used to keep global data structures from being left in
% half-changed states.
%
% If a lock is acquired, the owner can always be identified.  If
% a program that is discontinued (DSed) is the owner of the lock,
% its EPILOG procedure will relinquish the lock.  If a program
% left TTL locked, the next program to attempt to acquire the
% TITLE lock would have its processing suspended until the
% program is discontinued.
%
```

```
     PROTECTION_HEADING =
       LABEL   PROTECTION_LABEL;
       EPILOG PROCEDURE PROTECT_MARC;
         BEGIN
         DISALLOW;
         IF TTL_OWNER = PROCESSID THEN
           RELINQUISH_TTLLOCK;
         ALLOW;
         END PROTECT MARC;
       ACQUIRE_TTLLOCK #,
     %
     %  The following is the dummy label declared in the
     %  PROTECTION_HEADING define.  It is used to minimize programming
     %  errors by identifying the scope of the lock protection.
     %

     PROTECTION_TRAILER =
       PROTECTION_LABEL: #;

     %
     %  Since TITLE_SET is never reset once it is TRUE, SET_TITLE need
     %  never be called if TITLE_SET is TRUE.
     %

 DEFINE
     CHECK_TITLE     = IF NOT TITLE_SET THEN
                         SET_TITLE#;
PROCEDURE SET_TITLE;
     BEGIN

     %
     %  The SET_TITLE procedure is called only if TITLE_SET is FALSE.
     %  This procedure uses the PROTECTION_HEADING and
     %  PROTECTION_TRAILER defines to ensure proper access to the
     %  global TITLE lock and the TITLE_SET global variables.  The
     %  function of this procedure is to set the title of COMS_LIB to
     %  the correct name.
     %

     PROTECTION_HEADING;

     %
     %  The TITLE_SET Boolean must be checked under the lock.  This
     %  checking closes the timing hole caused when multiple stacks
     %  call SET_TITLE simultaneously.
     %

     IF NOT TITLE_SET THEN
       BEGIN
       REPLACE TTL BY MYSELF.EXCEPTIONTASK.EXCEPTIONTASK.NAME;
       COMS_LIB.TITLE:=HEAD(STRING(TTL, 72), NOT 48"ØØ");
       TITLE_SET:=TRUE;
       END;
     PROTECTION_TRAILER;
     END OF SET TITLE;
```

Defining Input Header Parameters

The input header parameters are used for the ACCEPT, ENABLE INPUT, ENABLE INPUT TERMINAL, DISABLE INPUT TERMINAL, and RECEIVE statements. These parameters are used by the TPTOMARC processing item. The following code defines input header parameters:

```
DEFINE
     COMS_IN_PROGRAM         = CD_IN[Ø]#,
     COMS_IN_FUNCTION_INDEX  = CD_IN[1]#,
     COMS_IN_USERCODE        = CD_IN[2]#,
     COMS_IN_SECURITY_DESG   = CD_IN[3]#,
     COMS_IN_DATE            = CD_IN[4]#,
     COMS_IN_TIMESTAMP       = CD_IN[5]#,
     COMS_IN_STATION         = CD_IN[6]#,
     COMS_IN_TEXT_LENGTH     = CD_IN[7]#,
     COMS_IN_END_KEY         = CD_IN[8]#,
     COMS_IN_STATUS_KEY      = CD_IN[9]#,
     COMS_IN_RST_LOC         = CD_IN[1Ø]#,
     COMS_IN_CD_SIZE(S)      = (STATE_CONVINX(S)) #;
```

Defining Output Header Parameters

The output header parameters are used for sending data. These parameters are used by the STATUS_LINE processing item. The following code defines output header parameters:

```
DEFINE

     COMS_OUT_COUNT          = CD_OUT[Ø]#,
     COMS_OUT_TEXT_LENGTH    = CD_OUT[1]#,
     COMS_OUT_STATUS_KEY     = CD_OUT[2]#,
     COMS_OUT_DESTIN_ERROR   = CD_OUT[3]#,
     COMS_OUT_DESTINATION    = CD_OUT[4]#,
     COMS_OUT_CONV_TSTAMP    =              % SDF FORM-KEY timestamp.
                               CD_OUT[STATE_CONVINX(STATE)]#,
     COMS_OUT_CONV_MSGN      =              % MARC MultiLingual System
                                            % (MLS) message number.
                               CD_OUT[STATE_CONVINX(STATE)+1]#;
```

Defining the STATE Parameters

The following code defines the STATE parameters, which is used by both processing items:

```
DEFINE

                        % [47:24]     % Available to the user.

                        % [23:24]     % Reserved for COMS use.

     STATE_IN_OUTF      = [ØØ:Ø1]#,   % 1 = Output header
                                      % Ø = Input header

     STATE_OUT     (S)  = BOOLEAN(S).STATE_IN_OUTF#,
     STATE_IN      (S)  = NOT STATE_OUT(S)#,

     STATE_CONVINXF     = [13:Ø6]#,   % Index to conversation area.

     STATE_CONVINX (S)  = (S).STATE_CONVINXF#,

     STATE_MSG_LOCF     = [15:Ø2]#,   % Valid data locations:
                                      %    Ø = USER_TEXT
                                      %    1 = TEXT_1
                                      %    2 = TEXT_2
                                      %    3 = Invalid value

     STATE_MSG_LOC (S)  = S.STATE_MSG_LOCF#;
```

Defining the Result Action Values

The following code defines the result action values of the STATUS_LINE processing item:

```
DEFINE

     CONTINUEV          = Ø#,
     STOPV              = 1#,
     STOP_RETURNV       = 2#;
```

Defining the Function Values

The following code defines the function values used by Menu-Assisted Resource Control (MARC). These values are required by the TPTOMARC processing item.

```
DEFINE

     FUNC_NORMALMSGV    = Ø#, % A normal message (for example, No ?).
     FUNC_CONTROLMSGV   =-1#; % A control message (for example, A ?).
```

TPTOMARC Processing Item

The TPTOMARC processing item translates the station designator associated with a message from a program in a direct window into a designator that MARC recognizes. As a result, programs in direct windows can send input messages to MARC, and MARC will accept them. (MARC discards messages with designators that it does not recognize.) The following code defines the TPTOMARC processing item:

```
REAL PROCEDURE TPTOMARC  (STATE, CD_IN, USER_TEXT, TEXT_1,
                          TEXT_2, OUTPUT_PROC);

     REAL   STATE;
     ARRAY  CD_IN[Ø];
     EBCDIC ARRAY USER_TEXT, TEXT_1, TEXT_2[Ø];
     REAL PROCEDURE OUTPUT_PROC(STATE, CD, TEXT_1, TEXT_2);
       REAL   STATE;
       ARRAY  CD[Ø];
       EBCDIC ARRAY TEXT_1, TEXT_2[Ø];
       FORMAL;
     BEGIN
     EBCDIC ARRAY REFERENCE
       MSGIN[Ø],        % Points to valid data.
       STA_NAME[Ø];     % Points to scratch data area.

     DEFINE
       CAND    (A, B)  = (IF (A) THEN (B)  ELSE FALSE)#;

     IF CAND(STATE_IN(STATE),
             COMS_IN_FUNCTION_INDEX = FUNC_NORMALMSGV OR
             COMS_IN_FUNCTION_INDEX = FUNC_CONTROLMSGV) THEN

       %
       %  The code above accepts input messages that are either normal
       %  input or control input.  Output messages and messages with
       %  other function values are ignored.
       %

       IF COMS_IN_PROGRAM ISNT Ø THEN      % Checks whether input
                                           % is not from a station.
         BEGIN

         %
         %  In the code above, the message has been rerouted by some
         %  other direct-window program.  If more security is required,
         %  check the value of COMS_IN_PROGRAM here, and write a
         %  procedure to provide the desired control.
         %
```

```
CASE STATE_MSG_LOC(STATE) OF
  BEGIN
  Ø:
    MSGIN:=USER_TEXT;
    STA_NAME:=TEXT_1;
  1:
    MSGIN:=TEXT_1;
    STA_NAME:=TEXT_2;
  2:
    MSGIN:=TEXT_2;
    STA_NAME:=TEXT_1;
  END;

%
%  MSGIN now references the text of the valid data and
%  STA_NAME references an unused, scratch data area; for
%  example, TEXT_1 or TEXT_2.
%


%  STA_NAME is resized to contain the largest possible
%  station name.
%

IF SIZE(STA_NAME) LSS 3ØØ THEN
  RESIZE(STA_NAME, 3ØØ);

%
%  Ordinarily, control messages are only identified by the
%  NDLII.  However, a processing item can change a message into
%  a control message (and vice versa) by changing the function
%  value in the Input CD.  TPTOMARC looks at the first
%  character of input.  If the input begins with a question
%  mark (?), the message is changed to a control message.  This
%  procedure allows programs to send control messages as well
%  as noncontrol messages to MARC.
%

IF MSGIN[Ø] = "?" THEN
  COMS_IN_FUNCTION_INDEX:=FUNC_CONTROLMSGV
ELSE
  COMS_IN_FUNCTION_INDEX:=FUNC_NORMALMSGV;
```

```
    %
    % Before the program calls COMS service functions, the
    % following code insures that the title of COMS_LIB has
    % been set.

    CHECK_TITLE;

    %
    % The next two steps are essential.
    %
    % First, the code calls GET_NAME_USING_DESIGNATOR to
    % translate the station designator to the station name
    % it represents.
    %

    REPLACE STA_NAME BY " " FOR 5Ø WORDS;
    GET_NAME_USING_DESIGNATOR(COMS_IN_STATION,
                              STA_NAME);

    %
    % Second, the code calls GET_DESIGNATOR_USING_NAME to change
    % the station name back to a station designator.  Because a
    % call is made on top of a MARC stack, the station designator
    % returned represents dialogue 1 of the MARC window for the
    % station name.
    %

    GET_DESIGNATOR_USING_NAME(STA_NAME,STATION_MNEMONIC,
                              COMS-IN-STATION);
    END;
END OF TPTOMARC;
```

STATUS LINE Processing Item

The STATUS_LINE processing item handles MARC error messages sent in response to direct-window input from ET, MT, or TD terminals. The processing item causes the error message to be displayed on the status line of the terminal. As a result, the screen of the user's screen is not disturbed by error messages received by the terminal.

Note that MARC uses the second word of the conversation area (COMS_OUT_CONV_MSGN) as follows:

- If the value COMS_OUT_CONV_MSGN ranges from 0 through 65535, then the value is the MultiLingual System (MLS) message number of the last MLS message MARC formatted into the output.
- If the value of COMS_OUT_CONV_MSGN ranges from –65535 through –1, then the value is a valid MLS message number of the last MLS message MARC formatted. However, the MLS message either was not found in the OUTPUTMESSAGE array or was not in the original language requested. This code is provided to show how to use an output processing item to monitor proper message translations.
- Any other value for COMS_OUT_CONV_MSGN means that MARC did not format an MLS message number into the output. MARC places the value 4“000000100000”(representing –65536) into COMS_OUT_CONV_MSGN to represent an invalid value.

The following code defines the STATUS_LINE processing item:

```
REAL PROCEDURE STATUSLINE(STATE, CD_OUT, USER_TEXT, TEXT_1,
                          TEXT_2, OUTPUT_PROC);

     REAL   STATE;
     ARRAY  CD_OUT[0];
     EBCDIC ARRAY USER_TEXT, TEXT_1, TEXT_2[0];
     REAL PROCEDURE OUTPUT_PROC(STATE, CD, TEXT_1, TEXT_2);
       REAL   STATE;
       ARRAY  CD[0];
       EBCDIC ARRAY TEXT_1, TEXT_2[0];
       FORMAL;
     BEGIN
     DEFINE
       DC1V           = 48"11"#, % Blind transmission for
                                 % ET, MT, and TD terminals.
       ESCV           = 48"27"#, % Escape character for
                                 % ET, MT, and TD terminals.
       HOMEV          = 48"3C"#, % Home character for ET, MT, and TD
                                 % terminals.
       STATUS_LINE_SZ = 60#,     % Maximum number of characters allowed
                                 % on the status line.
       LAST_WORD      = (((STATUS_LINE_SZ+30)DIV 6)-1)#;
     REAL
       MSGN,            % MLS message number.
       OUTPUT_MSG_LN;   % Size of text placed on the status line.
     EBCDIC ARRAY REFERENCE
       INP_DATA[0],
       OUT_DATA[0];
     REAL ARRAY REFERENCE
       ROUT_DATA [0];
     IF STATE_OUT(STATE) THEN
       BEGIN

       %
       %  The following code is an output message.  Input messages are
       %  ignored.
       %
```

```
MSGN:=COMS_OUT_CONV_MSGN(STATE);
IF MSGN GEQ 26ØØ AND MSGN LSS 29ØØ THEN
  BEGIN

  %
  %  MSGN contains the MLS message number placed in the output
  %  conversation area by MARC.
  %
  %  MSGN values between 26ØØ and 29ØØ are used by MARC to
  %  denote error responses caused by an input to a direct
  %  window.  Following is an example of such a response:
  %
  %  MESSAGE REJECTED, THE PROGRAM IS DISABLED.  TEXT = ....
  %
  CASE STATE_MSG_LOC(STATE) OF
    BEGIN
    Ø:

      %
      %  In the following code, the data is in USER_TEXT.  This
      %  data will be reformatted into TEXT_1.
      %

      INP_DATA:=USER_TEXT;
      OUT_DATA:=TEXT_1;
      STATE_MSG_LOC(STATE):=1;


    1:

      %
      %  In the following code, the data is in TEXT_1.  This
      %  data will be reformatted into TEXT_2.
      %

      INP_DATA:=TEXT_1;
      OUT_DATA:=TEXT_2;
      STATE_MSG_LOC(STATE):=2;
  2:

    %
    %  In the following code, the data is in TEXT_2.  This
    %  data will be reformatted into TEXT_1.
    %

    INP_DATA:=TEXT_2;
    OUT_DATA:=TEXT_1;
    STATE_MSG_LOC(STATE):=1;
  END; % CASE

%
%  INP_DATA now references the data area containing the valid
%  data.  OUT_DATA now references an unused scratch area.  The
%  purpose of this code is to reformat the input message from
%  INP_DATA into OUT_DATA, and add the appropriate control
%  characters.
%
%  If OUT_DATA is too small, it is resized.
%

IF SIZE(OUT_DATA) LSS (STATUS_LINE_SZ+30) THEN
  RESIZE(OUT_DATA, STATUS_LINE_SZ+30);

%
%  If the error message sent to the terminal is longer than
%  the smallest status line of an ET, MT, or TD terminal, the
%  message will be truncated.
%

OUTPUT_MSG_LN:=MIN(COMS_OUT_TEXT_LENGTH, STATUS_LINE_SZ);

%
%  The following code reformats the message.
%

REPLACE OUT_DATA[0] BY
  DC1V,                 % Suppresses NDLII editing.
  ESCV, "RS",           % Writes to the status line.
  "00",                 % Leaves room for the hexadecimal length
                        % of the text to be placed on the
                        % status line.
  INP_DATA[0] FOR       % Moves actual data.
    OUTPUT_MSG_LN,
  HOMEV,                % Leaves cursor in the home position.
  ESCV, "W";            % Puts terminal into forms mode.

%
```

```
         % The following code uses the HEXTOEBCDIC translation table
         % to translate the binary value of OUTPUT_MSG_LN into its
         % 2-character EBCDIC representation expressed in hexadecimal
         % notation.
         %
         % ROUT_DATA (a type-real-array reference variable to
         % OUT_DATA) is used to generate the HEX pointer required by
         % the HEXTOEBCDIC translation table.
         %

         ROUT_DATA:=OUT_DATA;
         ROUT_DATA[LAST_WORD].[47:8]:=OUTPUT_MSG_LN;
         REPLACE OUT_DATA[4] BY POINTER(ROUT_DATA[LAST_WORD],4)
           FOR 2 WITH HEXTOEBCDIC;

         %
         % The following code updates the length field in the Output
         % CD to reflect the edited message.
         %

         COMS_OUT_TEXT_LENGTH:=OUTPUT_MSG_LN+9;
         END;
       END;
     END OF STATUSLINE;
EXPORT
     STATUSLINE,
     TPTOMARC;

     FREEZE(TEMPORARY);
END.
```

Appendix E
Sample COBOL74 Processing-Item Interface

Processing items can be written in any programming language that ALGOL can call. This appendix provides an example of a COBOL74 program that can be bound to an ALGOL shell. Included in the example is a sample Binder program. (For further information on Binder programs, see the *A Series Binder Programming Reference Manual.*)

```
%
%  The assumption in Binder is that the object for this program unit
%  is titled "OBJECT/COMS/PROCESSINGITEM/ALGOLSHELL".
%
BEGIN
REAL PROCEDURE PROCESSING_ITEM
        (STATE,
         CD,
         USER_DATA,
         TEXT_1,
         TEXT_2,
         OUTPUT_PROC);
REAL STATE;
ARRAY CD [Ø];
EBCDIC ARRAY USER_DATA, TEXT_1, TEXT_2 [Ø];
REAL PROCEDURE OUTPUT_PROC
               (STATE,
                CD,
                TEXT_1,
                TEXT_2);
        REAL STATE;
        ARRAY CD [Ø];
        EBCDIC ARRAY TEXT_1, TEXT_2 [Ø];
        FORMAL;
```

```
BEGIN % PROCESSING_ITEM
%
% Since we cannot declare a typed procedure in COBOL74, we must
% simulate it by passing the result back from the subprogram
% as a parameter local to the shell, and then setting the value of the
% shell (a typed procedure) to the value returned in the parameter.
%
% This applies as well to calls to OUTPUT_PROC from the COBOL74
% program--we cannot call a typed procedure, so we declare an untyped
% procedure in the shell, and pass the result back to the COBOL74
% program as a parameter.
%
% Note that external to the COBOL74 program, the parameters are REALs.
% Internally they are integers; hence, the double definition.
%
% There are two ALGOL procedures declared global to the COBOL74 program
% to resize the text arrays (which can have zero length on entry).
% NEWTEXTSIZE is a parameter used by the subprogram and these two
% procedures.

REAL
    RETURNRESULTREAL,    % from the subprogram
    NEWTEXTSIZEREAL,     % for resizing of arrays if needed
    OUTPUTPROCRESULTREAL; % result from OUTPUT_PROC
INTEGER
    RETURNRESULT = RETURNRESULTREAL,
    NEWTEXTSIZE  = NEWTEXTSIZEREAL,
    OUTPUTPROCRESULT = OUTPUTPROCRESULTREAL;
%
PROCEDURE RESIZE_TEXT_1_IF_NEEDED;
BEGIN   % to avoid SEG ARRAY errors:
IF NEWTEXTSIZE GEQ SIZE (TEXT_1) THEN
    RESIZE (TEXT_1, NEWTEXTSIZE * 2, RETAIN);
END;
%
PROCEDURE RESIZE_TEXT_2_IF_NEEDED;
BEGIN   % likewise:
IF NEWTEXTSIZE GEQ SIZE (TEXT_2) THEN
    RESIZE (TEXT_2, NEWTEXTSIZE * 2, RETAIN);
END;
```

```
%
PROCEDURE CALL_OUTPUT_PROC;
BEGIN
OUTPUTPROCRESULTREAL := OUTPUT_PROC (STATE, CD, TEXT_1, TEXT_2);
END;
%
PROCEDURE SUBP; EXTERNAL; % This is the COBOL74 subprogram
%
% PROCESSING_ITEM EXECUTABLE
%
%   Call the COBOL74 Subprogram:
%
    SUBP;
%
%   Set PROCESSING_ITEM.VALUE to the result passed from the
%   COBOL74 subprogram:
%
    PROCESSING_ITEM := RETURNRESULTREAL;
%
%
    END OF PROCESSING_ITEM;

EXPORT PROCESSING_ITEM;

FREEZE (PERMANENT);

END.
%%%%%%%%%%%  End of ALGOL host for COBOL74 processing-item subprogram.

        SAMPLE COBOL74 SUBPROGRAM FOR PROCESSING ITEM



 *  The assumption in BINDER is that the object for this program unit
 *  is titled "OBJECT/COMS/PROCESSINGITEM/SUBPROGRAM".
 *
 *  Note:  The COBOL74 subprogram should be declared at LEX LEVEL 4.
 *
  $LEVEL = 4
 *
   IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   DATA DIVISION.
   WORKING-STORAGE SECTION.
 *
 * COMS-STATE, COMS-NEW-TEXT-SIZE, COMS-RETURN-RESULT,
 * COMS-OUTPUTPROC-RESULT, COMS-IN-CD-ARRAY,
 * COMS-IN-TEXT-1, COMS-IN-TEXT-2, and COMS-USER-DATA are all
 * global to the subprogram and MUST be so declared.
```

```
*
* Everything else is local to this subprogram.
* Try to minimize local data to avoid significant
* overhead due to stack building and teardown, since
* the local stack for this program will be built every time
* it is called.
*
  77  COMS-STATE             PICTURE S9(11) USAGE IS BINARY GLOBAL.
  77  COMS-NEW-TEXT-SIZE     PICTURE S9(11) USAGE IS BINARY GLOBAL.
  77  COMS-STATE-I-OR-O      PICTURE S9(11) USAGE IS BINARY.
  77  COMS-STATE-CONV-INX    PICTURE S9(11) USAGE IS BINARY.
  77  COMS-STATE-MSG-LOC     PICTURE S9(11) USAGE IS BINARY.
  77  COMS-STATE-USER-FIELD  PICTURE S9(11) USAGE IS BINARY.
  77  COMS-RETURN-RESULT     PICTURE S9(11) USAGE IS BINARY GLOBAL.
  77  COMS-OUTPUTPROC-RESULT PICTURE S9(11) USAGE IS BINARY GLOBAL.
  Ø1  COMS-IN-CD-ARRAY                      USAGE IS BINARY GLOBAL.
      Ø3  COMS-IN-PROGRAM           PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-FUNCTION-INDEX    PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-USERCODE          PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-SECURITY-DESG     PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-DATE              PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-TIMESTAMP         PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-STATION           PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-TEXT-LENGTH       PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-END-KEY           PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-STATUS-KEY        PICTURE S9(11) USAGE IS BINARY.
      Ø3  COMS-IN-RST-LOCATOR       PICTURE S9(11) USAGE IS BINARY.
  66 MARC-MESSAGE-NUMBER RENAMES COMS-IN-STATION.

  Ø1  COMS-IN-TEXT-1                        PICTURE X(192Ø) GLOBAL.
  Ø1  COMS-IN-TEXT-2                        PICTURE X(192Ø) GLOBAL.
  Ø1  COMS-IN-USER-DATA                     PICTURE X(192Ø) GLOBAL.
  PROCEDURE DIVISION.
*
  DECLARATIVES.
  CHECK-TEXT-1-SIZE SECTION.
     USE AS GLOBAL PROCEDURE.
  CHECK-TEXT-2-SIZE SECTION.
     USE AS GLOBAL PROCEDURE.
  COMS-OUTPUT-PROC SECTION.
     USE AS GLOBAL PROCEDURE.
  END DECLARATIVES.
*
  MAIN-SECTION.
  ENTRY-PARAGRAPH.
```

```
*
* Split the fields in STATE into separate words so they can be
* more easily examined/manipulated in the subprogram:
      MOVE Ø TO COMS-STATE-I-OR-O,
                COMS-STATE-CONV-INX,
                COMS-STATE-MSG-LOC,
                COMS-STATE-USER-FIELD.
      MOVE COMS-STATE TO COMS-STATE-I-OR-O [Ø:Ø:1].
      MOVE COMS-STATE TO COMS-STATE-CONV-INX [13:5:6].
      MOVE COMS-STATE TO COMS-STATE-MSG-LOC [15:1:2].
      MOVE COMS-STATE TO COMS-STATE-USER-FIELD [47:23:24].
*
*
*       Body of this particular processing item:
*

      IF COMS-STATE-I-OR-O > Ø
          IF MARC-MESSAGE-NUMBER > 29ØØ
          OR MARC-MESSAGE-NUMBER < 26ØØ
              MOVE 1 TO COMS-RETURN-RESULT
              GO TO RETURN-TO-COMS.
      IF COMS-IN-FUNCTION-INDEX = -1
          MOVE 1 TO COMS-RETURN-RESULT
          GO TO RETURN-TO-COMS.
      IF COMS-IN-FUNCTION-INDEX = Ø
         IF COMS-STATE-MSG-LOC = Ø
             ADD 1Ø, COMS-IN-TEXT-LENGTH GIVING COMS-NEW-TEXT-SIZE
             CALL CHECK-TEXT-1-SIZE
             STRING "?ON CANDE:" DELIMITED BY SIZE,
                    COMS-IN-USER-DATA FOR COMS-IN-TEXT-LENGTH
                INTO COMS-IN-TEXT-1
             MOVE -1 TO COMS-IN-FUNCTION-INDEX
             MOVE COMS-NEW-TEXT-SIZE TO COMS-IN-TEXT-LENGTH
             MOVE 1 TO COMS-STATE-MSG-LOC
         ELSE IF COMS-STATE-MSG-LOC = 1
             ADD 1Ø, COMS-IN-TEXT-LENGTH GIVING COMS-NEW-TEXT-SIZE
             CALL CHECK-TEXT-2-SIZE
             STRING "?ON CANDE:" DELIMITED BY SIZE,
                    COMS-IN-TEXT-1 FOR COMS-IN-TEXT-LENGTH
                INTO COMS-IN-TEXT-2
             MOVE -1 TO COMS-IN-FUNCTION-INDEX
             MOVE COMS-NEW-TEXT-SIZE TO COMS-IN-TEXT-LENGTH
             MOVE 2 TO COMS-STATE-MSG-LOC
         ELSE IF COMS-STATE-MSG-LOC = 2
             ADD 1Ø, COMS-IN-TEXT-LENGTH GIVING COMS-NEW-TEXT-SIZE
             CALL CHECK-TEXT-1-SIZE
             STRING "?ON CANDE:" DELIMITED BY SIZE,
                    COMS-IN-TEXT-2 FOR COMS-IN-TEXT-LENGTH
                INTO COMS-IN-TEXT-1
             MOVE -1 TO COMS-IN-FUNCTION-INDEX
             ADD 1Ø TO COMS-IN-TEXT-LENGTH
             MOVE 1 TO COMS-STATE-MSG-LOC.
```

```
*
*
*  Should the user desire to call COMS' OUTPUT_PROC:
*       1)  Make sure the STATE word is restored before calling;
*       2)  Make sure it is "redistributed" after calling;
*       3)  Take appropriate action based on output result.
*  Example:
*
*    PERFORM COMS-OUTPUT-PARAGRAPH THROUGH
*            COMS-OUTPUT-PARAGRAPH-EXIT.
*    IF COMS-OUTPUTPROC-RESULT .........
*
*
 COMS-OUTPUT-PARAGRAPH.
* Rebuild COMS-STATE:
      MOVE COMS-STATE-I-OR-O TO COMS-STATE [Ø:Ø:1].
      MOVE COMS-STATE-CONV-INX TO COMS-STATE [5:13:6].
      MOVE COMS-STATE-MSG-LOC TO COMS-STATE [1:15:2].
      MOVE COMS-STATE-USER-FIELD TO COMS-STATE [23:47:24].
* Call ALGOL shell procedure that calls OUTPUT_PROC:
      CALL COMS-OUTPUT-PROC.
* Redistribute COMS-STATE into local words:
      MOVE Ø TO COMS-STATE-I-OR-O,
                COMS-STATE-MSG-LOC,
                COMS-STATE-USER-FIELD.
      MOVE COMS-STATE TO COMS-STATE-I-OR-O [Ø:Ø:1].
      MOVE COMS-STATE TO COMS-STATE-CONV-INX [13:5:6].
      MOVE COMS-STATE TO COMS-STATE-MSG-LOC [15:1:2].
      MOVE COMS-STATE TO COMS-STATE-USER-FIELD [47:23:24].
 COMS-OUTPUT-PARAGRAPH-EXIT.
   EXIT.

*
*
*       RETURNING TO ALGOL SHELL:
*
* Particularly in the event you have manipulated the data items
* extracted from COMS-IN-STATE, be sure it is updated before
* returning to the calling ALGOL shell:
*
  RETURN-TO-COMS.
      MOVE COMS-STATE-I-OR-O TO COMS-STATE [Ø:Ø:1].
      MOVE COMS-STATE-CONV-INX TO COMS-STATE [5:13:6].
      MOVE COMS-STATE-MSG-LOC TO COMS-STATE [1:15:2].
      MOVE COMS-STATE-USER-FIELD TO COMS-STATE [23:47:24].
*
* Exit to shell:
*
  EXIT-PARAGRAPH.
      EXIT PROCEDURE.
****************** END OF COBOL74 SUBPROGRAM
```

```
%
%
%       SAMPLE WFL SOURCE:
%

BEGIN JOB PROCESSITEM;
CLASS = 16;
JOBSUMMARY = UNCONDITIONAL;
COMPILE OBJECT/COMS/PROCESSINGITEM/ALGOLSHELL WITH ALGOL LIBRARY;
        COMPILER FILE CARD (KIND=DISK,
                            DEPENDENTSPECS = TRUE,
                            TITLE = COMS/PROCESSINGITEM/ALGOLSHELL);
COMPILE OBJECT/COMS/PROCESSINGITEM/SUBPROGRAM WITH COBOL74 LIBRARY;
        COMPILER FILE CARD (KIND=DISK,
                            DEPENDENTSPECS = TRUE,
                            TITLE = COMS/PROCESSINGITEM/SUBPROGRAM);
COMPILE OBJECT/COMS/PROCESSINGITEM/BOUND WITH BINDER LIBRARY;
        COMPILER FILE CARD (KIND = DISK,
                            DEPENDENTSPECS = TRUE,
                            TITLE = COMS/PROCESSINGITEM/BOUND);
?END JOB
%%%%%%%%%%END OF SAMPLE WFL SOURCE

%
%       SAMPLE BINDER SOURCE:
%
%
HOST IS OBJECT/COMS/PROCESSINGITEM/ALGOLSHELL;
BIND SUBP FROM OBJECT/COMS/PROCESSINGITEM/SUBPROGRAM;
%
%       Correlate the names in the ALGOL shell with those in the
%       COBOL74 subprogram:
%
%   Name in ALGOL shell:                  Name in COBOL74 Subprogram:
%
USE STATE                   FOR       COMS-STATE;
USE RETURNRESULT            FOR       COMS-RETURN-RESULT;
USE OUTPUTPROCRESULT        FOR       COMS-OUTPUTPROC-RESULT;
USE CD                      FOR       COMS-IN-CD-ARRAY;
USE TEXT_1                  FOR       COMS-IN-TEXT-1;
USE TEXT_2                  FOR       COMS-IN-TEXT-2;
USE USER_DATA               FOR       COMS-IN-USER-DATA;
USE RESIZE_TEXT_1_IF_NEEDED FOR       CHECK-TEXT-1-SIZE;
USE RESIZE_TEXT_2_IF_NEEDED FOR       CHECK-TEXT-2-SIZE;
USE CALL_OUTPUT_PROC        FOR       COMS-OUTPUT-PROC;
USE NEWTEXTSIZE             FOR       COMS-NEW-TEXT-SIZE;
STOP;
%%%%%%%%%%%%%%%% END OF BINDER SOURCE
```

Appendix F
Service Functions for Previous Releases

If you have developed your programs using Mark 3.6 or Mark 3.5 COMS, the service functions described in this appendix are valid for those programs. These service functions cannot be used by either the Pascal or the Report Program Generator (RPG) programming languages.

Programs that use the null character to initialize arrays for entity names, however, must be modified to use the space character. When COMS returns an entity name in response to a service function call, it uses space characters to blank fill the remainder of the array. Thus, programs that initialize and scan for null characters will fail.

Agenda Designators and Names

You can use COMS service functions to obtain information on

- Agenda designators (with GET_AGENDA_DESIGNATOR)
- Agenda names (with GET_AGENDA_NAME)

The following pages describe these service functions.

Agenda Designators

The COMS library returns an agenda designator when you pass to the GET_AGENDA_DESIGNATOR service function either of the following:

- An agenda name with a length of 1 to 17 alphanumeric characters. It is assumed that the agenda is defined for the window in which the program is running.
- An agenda name, with a length of 1 to 17 alphanumeric characters, and a window name, with a length of 1 to 17 alphanumeric characters, separated by the word "OF". This is how a program can reference agendas defined for windows other than the window in which the program is running. For example, the following syntax allows a program to reference agendas that are related to windows other than the window in which the program is running.

```
<agenda name> OF <window name>
```

Origin of Input Parameter

Each agenda name is defined with the COMS Utility. To obtain reports about the agenda names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–1 shows the input parameters you pass to the GET_AGENDA_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–1. GET_AGENDA_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Agenda name	COBOL74: Display	EBCDIC array
Output from COMS	Agenda designator	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_AGENDA_DESIGNATOR call if there is an invalid agenda name for the window in which the program is running.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–1 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_AGENDA_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
 CALL "GET_AGENDA_DESIGNATOR OF DCILIBRARY"
     USING <agenda name>,
     GIVING <agenda designator>.
```

ALGOL

```
<agenda designator> :=  GET_AGENDA_DESIGNATOR
     (<agenda name>);
```

Agenda Names

When you pass an agenda designator to the GET_AGENDA_NAME service function, the COMS library returns an agenda name with a length of 1 to 17 alphanumeric characters.

Origin of Input Parameter

You can obtain an agenda designator from the GET_AGENDA_DESIGNATOR service function.

Programming Requirements

Table F–2 shows the input parameters you pass to the GET_AGENDA_NAME service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–2. GET_AGENDA_NAME Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Agenda designator	COBOL74: Display	Integer
Output from COMS	Agenda name	COBOL74: Display	EBCDIC array
Output from COMS	Function value	The following function values can be returned by the GET_AGENDA_NAME call: ● 1 = No errors ● 0 = Invalid designator	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–2 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_AGENDA_NAME service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_AGENDA_NAME  IN DCILIBRARY"
     USING <agenda designator>,
           <agenda name>,
     GIVING <result of GET_AGENDA_NAME service function>.
```

ALGOL

```
<result of  GET_AGENDA_NAME service function> :=
      GET_AGENDA_NAME  (<agenda designator>,
                       <agenda name>);
```

Window Designators and Maximum Users

You can use COMS service functions to obtain information on the following:

- Window designators (with GET_WINDOW_DESIGNATOR)
- The maximum number of users who can access a window at one time (with GET_WINDOW_INFO)

The following pages describe these service functions.

Window Designators

When a program passes a window name having a length of 1 to 17 alphanumeric characters to the GET_WINDOW_DESIGNATOR service function, the COMS library returns a window designator.

If a program does not know the name of the window it is associated with, you can place an asterisk (*) in the first column of the parameter you pass for the window name. COMS will return the window designator for the window associated with the calling program.

Calling the GET_WINDOW_DESIGNATOR service function is necessary if you want to obtain a window designator in order to call the GET_WINDOW_INFO service function.

Origin of Input Parameter

A window name is defined with COMS Utility at system-definition time. To obtain reports about the window names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–3 shows the input parameters you pass to the GET_WINDOW_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–3. GET_WINDOW_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Window name	COBOL74: Display	EBCDIC array
Output from COMS	Window designator	COBOL74: Binary	Integer

continued

Table F–3. **GET_WINDOW_DESIGNATOR** Parameters (cont.)

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Output from COMS	Function value	The following function values can be returned by the GET_WINDOW_DESIGNATOR call: • 0 = Invalid window name • Other = Requested window designator	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–3 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_WINDOW_DESIGNATOR service function from an application program or a processing item:

COBOL74

```
CALL "GET_WINDOW_DESIGNATOR  IN DCILIBRARY"
     USING <window name>,
     GIVING <window designator>.
```

ALGOL

```
<window designator> :=  GET_WINDOW_DESIGNATOR
     (<window name>);
```

Maximum Users of a Window

When you pass a window designator to the GET_WINDOW_INFO service function, the COMS library returns the maximum number of users that can access the window at a particular time. COMS returns this value in the first word of the zero-relative array (that is, INFO[0]), which you pass for receiving output.

Origin of Input Parameter

You can obtain a window designator by using the window name to call the GET_WINDOW_DESIGNATOR service function.

Programming Requirements

Table F–4 shows the input parameters you pass to the GET_WINDOW_INFO service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–4. GET_WINDOW_INFO Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Window designator	COBOL74: Binary	Integer
Output from COMS	Window information	COBOL74: Binary	Integer
Output from COMS	Function value	The following function values can be returned by the GET_WINDOW_INFO call: • 0 = Invalid window name • 1 = No errors	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–4 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_WINDOW_INFO service function from an application program or a processing item:

COBOL74

```
 CALL "GET_WINDOW_INFO  IN DCILIBRARY"
      USING <window designator>,
            <window info>,
      GIVING <result of GET_WINDOW_INFO service function>.
```

ALGOL

```
<result of  GET_WINDOW_INFO  service function> :=
      GET_WINDOW_INFO (<window designator>,
     <window info>);
```

Data Comm Device-Type Designators and Names

You can use COMS service functions to obtain information on

- Data comm device-type designators (with GET_DEVICE_DESIGNATOR)
- Data comm device-type names (with GET_DEVICE_NAME)

The following pages describe these service functions.

Device-Type Designators

When you pass a device-type name with a length of 1 to 17 alphanumeric characters to the GET_DEVICE_DESIGNATOR service function, the COMS library returns a device-type designator.

Origin of Input Parameter

Each device-type name is defined with the COMS Utility at system-definition time. To obtain reports about the device-type names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–5 shows the input parameter you pass to the GET_DEVICE_DESIGNATOR service function and the output parameter that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–5. GET_DEVICE_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Device name	COBOL74: Binary	EBCDIC array
Output from COMS	Window information	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_DEVICE_DESIGNATOR call if there is an invalid device-type name.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–5 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_DEVICE_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
 CALL "GET_DEVICE_DESIGNATOR  IN DCILIBRARY"
      USING <device-type name>,
      GIVING <device-type designator>.
```

ALGOL

```
<device-type designator> :=  GET_DEVICE_DESIGNATOR
     (<device-type name>);
```

Device-Type Names

When you pass a device-type designator to the GET_DEVICE_NAME service function, the COMS library returns a device-type name with a length of 1 to 17 alphanumeric characters.

Origin of Input Parameter

You can obtain a device-type designator from one of three sources:

- The GET_DEVICE_DESIGNATOR service function
- The GET_STATION_NAME service function
- The GET_STATION_ATTRIBUTES service function

Programming Requirements

Table F–6 shows the input parameters you pass to the GET_DEVICE_NAME service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–6. GET_DEVICE_NAME Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Device-type designator	COBOL74: Binary	Integer
Output from COMS	Device-type name	COBOL74: Display	EBCDIC array
Output from COMS	Function value	The following function values can be returned by the GET_DEVICE_NAME call: • 1 = No errors • 0 = Invalid designator	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–6 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_DEVICE_NAME service function from an application program or an agenda processing item:

COBOL74

```
 CALL "GET_DEVICE_NAME  IN DCILIBRARY"
     USING <device-type designator>,
           <device-type name>,
     GIVING <result of  GET_DEVICE_NAME service function>.
```

ALGOL

```
<result of  GET_DEVICE_NAME   service function> :=
      GET_DEVICE_NAME  (<device-type designator>,
                      <device-type name>);
```

Program Designators and Names

You can use COMS service functions to obtain information on

- Program designators (with GET_PROGRAM_DESIGNATOR)
- Program names (with GET_PROGRAM_NAME)

The following pages describe these service functions.

Program Designators

When you pass a program name with a length of 1 to 17 alphanumeric characters to the GET_PROGRAM_DESIGNATOR service function, the COMS library returns a program designator.

If you place an asterisk (*) in the first column of the parameter you pass for the program name, then COMS returns a program designator for the calling program. A processing item can call the GET_PROGRAM_DESIGNATOR service function to determine which program is calling it.

Origin of Input Parameter

A program name is defined with the COMS Utility at system-definition time. To obtain reports about the program names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–7 shows the input parameters you pass to the GET_PROGRAM_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–7. GET_PROGRAM_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Program name	COBOL74: Display	EBCDIC array
Output from COMS	Program designator	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_PROGRAM_DESIGNATOR call if there is an invalid program name.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–7 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_PROGRAM_DESIGNATOR service function from an application program or a processing item:

COBOL74

```
CALL "GET_PROGRAM_DESIGNATOR  IN DCILIBRARY"
     USING <program name>,
     GIVING <program designator>.
```

ALGOL

```
<program designator> :=  GET_PROGRAM_DESIGNATOR
     (<program name>);
```

Program Names

When you pass a program designator to the GET_PROGRAM_NAME service function, the COMS library returns a program name, with a length of 1 to 17 alphanumeric characters, and the program-security designator.

Origin of Input Parameter

You can obtain a program designator from either of the following sources:

- The COMS-in-Program field of the input CD in an application program
- The GET_PROGRAM_DESIGNATOR service function

Programming Requirements

Table F–8 shows the input parameters you pass to the GET_PROGRAM_NAME service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–8. GET_PROGRAM_NAME Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Program designator	COBOL74: Binary	Integer
Output from COMS	Program-security designator	COBOL74: Binary	Integer
Output from COMS	Program name	COBOL74: Display	EBCDIC array

continued

Table F–8. GET_PROGRAM_NAME Parameters (cont.)

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Output from COMS	Function value	The following function values can be returned by the GET_PROGRAM_NAME call: • 1 = No errors 0 = Invalid designator	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–8 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_PROGRAM_NAME service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_PROGRAM_NAME  IN DCILIBRARY"
     USING <program designator>,
           <program-security designator>,
           <program name>,
     GIVING <result of GET_PROGRAM_NAME service function>.
```

ALGOL

```
<result of  GET_PROGRAM_NAME  service function> :=
      GET_PROGRAM_NAME  (<program designator>,
                         <program-security designator>,
                         <program name>);
```

Security and Usercodes

The most basic unit of security for COMS is the security category or security level. Up to 32 security categories can be defined, although the maximum number of categories for a list is limited to the number of defined categories. The categories can be combined in a security-category list. Each station and each usercode is assigned a security-category list. Therefore, the terms *station security* and *usercode security* tell you which security categories are valid for a station or usercode, respectively.

However, each trancode can be assigned only one security category, so that station security or usercode security tells you which trancodes are valid for the station or usercode.

One additional COMS security structure is called session security. This is the intersection of the valid security categories for a particular usercode and station.

You can use COMS service functions to obtain information on

- Program-security designators (with the service function GET_PROGRAM_SECURITY_DESIGNATOR)
- Security-category designators (with the service function GET_SECURITY_CATEGORY_DESIGNATOR)
- Station-security designators (with the service function GET_STATION_SECURITY_DESIGNATOR)
- Usercode designators (with the service function GET_USER_DESIGNATOR)
- Usercode names and usercode-security designators (with the service function GET_USER)
- Usercode security-category-list designators (with the service function GET_USER_SECURITY_DESIGNATOR)

In addition, you can use the TEST_SECURITY_CATEGORY service function to test the security category of various designators.

The following pages describe these service functions.

Program Security Designators

When you pass a program designator to the service function called GET_PROGRAM_SECURITY_DESIGNATOR, the COMS library returns a designator that represents the valid security-category list associated with that program.

Origin of Input Parameter

You can obtain a program designator from either of the following sources:

- The COMS-in-Program field of the input CD in an application program
- The GET_PROGRAM_DESIGNATOR service function

Programming Requirements

Table F–9 shows the input parameters you pass to the GET_PROGRAM_SECURITY_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–9. GET_PROGRAM_SECURITY_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Program designator	COBOL74: Display	Integer
Output from COMS	Program-security designator	COBOL74:Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_PROGRAM_SECURITY_DESIGNATOR call if there is an invalid security category.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–9 at level 77.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_PROGRAM_SECURITY_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
 CALL "GET_PROGRAM_SECURITY_DESIGNATOR IN DCILIBRARY"
      USING <program designator>,
      GIVING <program-security designator>.
```

ALGOL

```
<program-security designator> :=
      GET_PROGRAM_SECURITY_DESIGNATOR
          (<program designator>);
```

Security-Category Designators

When you pass a security-category name with a length of 1 to 17 alphanumeric characters to the service function called GET_SECURITY_CATEGORY_DESIGNATOR, the COMS library returns a security-category designator.

Origin of Input Parameter

Each security-category name is defined with the COMS Utility at system-definition time. To obtain reports about the security-category names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–10 shows the input parameters you pass to the GET_SECURITY_CATEGORY_

DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–10. GET_SECURITY_CATEGORY_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Security category	COBOL74: Binary	Integer
Output from COMS	Program-security designator	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_SECURITY_CATEGORY_DESIGNATOR call if there is an invalid security category.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–10 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_SECURITY_CATEGORY_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
Ø1 <security-category name> PIC X(8Ø).
77 <security-category designator> PIC S9(11) BINARY.
    •
    •
MOVE <payroll manager> TO <security-category name>.
CALL "GET_SECURITY_CATEGORY_DESIGNATOR
    IN DCILIBRARY"
    USING <security-category name>,
    GIVING <security-category designator>.
IF <security-category designator> = Ø
    DISPLAY "Invalid security-category name".
```

ALGOL

```
EBCDIC ARRAY <security-category name> [Ø:79];
INTEGER <security-category designator>;
INTEGER PROCEDURE  GET_SECURITY_CATEGORY_DESIGNATOR
     (<security-category name>);
     LIBRARY DCILIBRARY;
   •
   •
REPLACE <security-category name> [Ø] BY
     <payroll manager>;
<security-category designator> :=
      GET_SECURITY_CATEGORY_DESIGNATOR
     (<security-category name>);
IF <security-category designator>= Ø THEN
     DISPLAY ("Invalid security-category name");
```

Station-Security Designators

When you pass a station designator to the service function called GET_STATION_SECURITY_DESIGNATOR, the COMS library returns a designator that represents the security-category list associated with the station.

Origin of Input Parameter

You can obtain a station designator from either of the following sources:

- The COMS-in-Station field of the input CD in an application program
- The GET_STATION_DESIGNATOR service function

Programming Requirements

Table F–11 shows the input parameters you pass to the GET_STATION_SECURITY_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–11. GET_STATION_SECURITY_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Station designator	COBOL74: Binary	Integer
Output from COMS	Station-security designator	COBOL74: Binary	Integer

continued

Table F–11. GET_STATION_SECURITY_DESIGNATOR Parameters (cont.)

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Output from COMS	Function value	The function value 0 is returned by the GET_STATION_SECURITY_DESIGNATOR call if there is an invalid designator.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–11 at level 77.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_STATION_SECURITY_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_STATION_SECURITY_DESIGNATOR
     IN DCILIBRARY"
     USING <station designator>,
     GIVING <station-security designator>.
```

ALGOL

```
 <station-security designator> :=
       GET_STATION_SECURITY_DESIGNATOR
           (<station designator>);
```

Usercode Designators

When you pass a usercode name with a length of 1 to 17 alphanumeric characters to the GET_USER_DESIGNATOR service function, the COMS library returns a usercode designator.

Origin of Input Parameter

Each usercode name is defined with the COMS Utility at system-definition time. To obtain reports about the usercode names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–12 shows the input parameters you pass to the GET_USER_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–12. GET_USER_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Usercode name	COBOL74: Display	EBCDIC array
Output from COMS	Usercode designator	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_USER_DESIGNATOR call if there is an invalid usercode.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–12 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_USER_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_USER_DESIGNATOR  IN DCILIBRARY"
     USING <usercode name>,
     GIVING <usercode designator>.
```

ALGOL

```
<usercode designator> := GET_USER_DESIGNATOR
     (<usercode name>);
```

Usercode Names and Security Designators

When you pass a usercode designator to the GET_USER service function, the COMS library returns a usercode name, with a length of 1 to 17 alphanumeric characters, and a usercode-security designator.

If the GET_USER_DESIGNATOR service function receives a designator corresponding to the superuser usercode, the GET_USER_DESIGNATOR service function returns a name of 17 blanks.

If the GET_USER_DESIGNATOR service function receives a name consisting of 17 blanks, then the GET_USER_DESIGNATOR service function returns the superuser designator. For more detailed information on the superuser usercode and superuser designator, refer to the *Security Administration Guide.*

Origin of Input Parameter

You can obtain a usercode designator from either of the following sources:

- The COMS-in-Usercode field of the input CD in an application program
- The GET_USER_DESIGNATOR service function

Programming Requirements

Table F–13 shows the input parameters you pass to the GET_USER service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–13. GET_USER Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Usercode designator	COBOL74: Binary	Integer
Output from COMS	Usercode name	COBOL74: Display	EBCDIC array
Output from COMS	Usercode-security designator	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_USER call if there is an invalid designator.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–13 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_USER service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_USER  IN DCILIBRARY"
     USING <usercode'designator>,
           <usercode-security designator>,
           <usercode name>,
     GIVING <result of  GET_USER  service function>.
```

ALGOL

```
<result of  GET_USER > service function> :=
      GET_USER  (<usercode designator>,
               <usercode-security designator>,
               <usercode name>);
```

Usercode Security-Category-List Designators

When you pass a usercode designator to the service function called GET_USER_SECURITY_DESIGNATOR, the COMS library returns a designator that represents the security-category list associated with the usercode.

Origin of Input Parameter

You can obtain a usercode designator from either of the following sources:

- The COMS-in-Usercode field of the input CD in an application program
- The GET_USER_DESIGNATOR service function

Programming Requirements

Table F–14 shows the input parameters you pass to the GET_USER_SECURITY_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–14. GET_USER_SECURITY_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Usercode designator	COBOL74: Binary	Integer
Output from COMS	Usercode-security designator	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_USER_SECURITY_DESIGNATOR call if there is an invalid designator.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–14 at level 77.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_USER_SECURITY_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_USER_SECURITY_DESIGNATOR
     IN DCILIBRARY"
     USING <usercode designator>,
     GIVING <usercode-security designator>.
```

ALGOL

```
<usercode-security designator> :=
      GET_USER_SECURITY_DESIGNATOR
         (<usercode designator>);
```

Security-Category Testing

You can use the TEST_SECURITY_CATEGORY service function to test the security category of designators for stations, usercodes, or sessions.

When you pass a security-category designator and a security designator (a designator that represents the valid security categories for a station, a usercode, or a session) to this service function, the COMS library returns a function value that tells you whether the security categories represented by the designator are valid for the station, usercode, or session.

Origin of Input Parameters

A security-category designator can be obtained from the GET_SECURITY_CATEGORY_DESIGNATOR service function.

A station-security designator can be obtained from the GET_STATION_SECURITY_DESIGNATOR service function.

A usercode-security designator can be obtained from the GET_USER_SECURITY_DESIGNATOR service function.

A session-security designator can be obtained from the COMS-in-Security-Desg field of the input CD in an application program.

Programming Requirements

Table F–15 shows the input parameters you pass to the TEST_SECURITY_CATEGORY service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–15. TEST_SECURITY_CATEGORY Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Security-category designator	COBOL74: Binary	Integer
Output from COMS	Security-category test	COBOL74: Binary	Integer
Output from COMS	Function value	The following function values can be returned by the TEST_SECURITY_CATEGORY call: • 1 = Valid category • 0 = Invalid category	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–15 at level 77.

Examples

Following are examples of COBOL and ALGOL code for calling the TEST_SECURITY_CATEGORY service function from an application program or an agenda processing item:

COBOL74

```
CALL "TEST_SECURITY_CATEGORY  IN DCILIBRARY"
     USING <security designator>,
           <security-category designator>,
     GIVING <result of TEST_SECURITY_CATEGORY
            service function>.
```

ALGOL

```
<result of TEST_SECURITY_CATEGORY  service function> :=
      TEST_SECURITY_CATEGORY  (<security designator>,
          <security-category designator>);
```

Station Information

You can use the COMS service functions to obtain information on

- Station attributes (with GET_STATION_ATTRIBUTES)
- Station designators (with GET_STATION_DESIGNATOR)
- Station lists (with GET_STATION_LIST)

- Station-list designators (with GET_STATION_LIST_DESIGNATOR)
- Station names (with GET_STATION_NAME)

The following pages describe these service functions.

Station Attributes

When you pass a station designator to the GET_STATION_ATTRIBUTES service function, the COMS library returns the following information:

- A logical station number (LSN). If this number is 0, then the station is disconnected.
- A device designator.
- A station-security designator.
- A virtual terminal.
- A screen size.

Origin of Input Parameter

You can obtain a station designator from either of the following sources:

- The COMS-in-Station field of the input CD in an application program
- The GET_STATION_DESIGNATOR service function

Programming Requirements

Table F–16 shows the input parameters you pass to the GET_STATION_ATTRIBUTES service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–16. GET_STATION_ATTRIBUTES Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Station designator	COBOL74: Binary	Integer
Output from COMS	LSN	COBOL74: Binary	Integer
Output from COMS	Device designator	COBOL74: Binary	Integer
Output from COMS	Station-security designator	COBOL74: Binary	Integer

continued

Table F–16. GET_STATION_ATTRIBUTES Parameters (cont.)

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Output from COMS	Virtual terminal	COBOL74: Binary	Integer
Output from COMS	Screen size	COBOL74: Binary	Integer
Output from COMS	Function value	The following function values can be returned by the GET_STATION_ATTRIBUTES call: • 1 = No errors • 0 = One of the following has occurred: – An invalid designator was used – The station was no longer logged on to COMS	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–16 at level 77.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_STATION_ATTRIBUTES service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_STATION_ATTRIBUTES  IN DCILIBRARY"
     USING <station designator>
           <device designator>
           <LSN>
           <station-security designator>
           <virtual terminal>
           <screen size>
     GIVING <result of  GET_STATION_ATTRIBUTES
            service function>
```

ALGOL

```
<result of  GET_STATION_ATTRIBUTES  service function> :=
      GET_STATION_ATTRIBUTES  (<station designator>,
          <device designator>, <LSN>, <virtual terminal>,
          <screen size>, <station-security designator>);
```

Station Designators

When you pass a valid station name (with a length of 1 to 255 alphanumeric characters) to the GET_STATION_DESIGNATOR service function, the COMS library returns a station designator.

Origin of Input Parameter

Each station name is defined with the COMS Utility at system-definition time. To obtain reports about the station names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–17 shows the input parameters you pass to the GET_STATION_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–17. GET_STATION_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Station name	COBOL74: Display	EBCDIC array
Output from COMS	Station designator	COBOL74: Binary	Integer
Output from COMS	Function value	The function value 0 is returned by the GET_STATION_DESIGNATOR call if there is an invalid station name.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–17 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_STATION_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_STATION_DESIGNATOR  IN DCILIBRARY"
     USING <station name>,
     GIVING <station designator>.
```

ALGOL

```
<station designator> :=
     GET_STATION_DESIGNATOR  (<station name>);
```

Station Lists

When you pass a station-list designator to the GET_STATION_LIST service function, the COMS library returns an array of designators that represent the stations included in the station list. Each element in the array is 1 word (6 bytes).

Origin of Input Parameter

You can obtain a station-list designator by calling the GET_STATION_LIST_DESIGNATOR service function.

Programming Requirements

Table F–18 shows the input parameters you pass to the GET_STATION_LIST service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–18. GET_STATION_LIST Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Station-list designator	COBOL74: Binary	Integer
Output from COMS	Station list	COBOL74: Binary	Integer array
Output from COMS	Function value	The following function values can be returned by the GET_STATION_LIST call: • 1 through n = The number of designators returned in list • 0 = Invalid station-list designator	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the station-list designator in Table F–18 at level 77. Declare the station list at level 01 using binary (for COBOL74). The size of the station list should be large enough to hold the largest station list requested.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_STATION_LIST service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_STATION_LIST  IN DCILIBRARY"
     USING <station-list designator>,
           <station list>
     GIVING <result of  GET_STATION_LIST
            service function>
```

ALGOL

```
<result of  GET_STATION_LIST  service function> :=
      GET_STATION_LIST  (<station-list designator>,
          <station list>);
```

Station-List Designators

When you pass a station-list name with a length of 1 to 17 alphanumeric characters to the service function called GET_STATION_LIST_DESIGNATOR, the COMS library returns a designator representing the station list.

Origin of Input Parameter

A station-list name is defined with the COMS Utility at system-definition time. To obtain reports about the station-list names and other entities defined for your system, refer to the *COMS Configuration Guide.*

Programming Requirements

Table F–19 shows the input parameters you pass to the GET_STATION_LIST_DESIGNATOR service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–19. GET_STATION_LIST_DESIGNATOR Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Station-list name	COBOL74: Display	EBCDIC array
Output from COMS	Station-list designator	COBOL74: Binary	Integer

continued

Table F–19. GET_STATION_LIST_DESIGNATOR Parameters (cont.)

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Output from COMS	Function value	The function value 0 is returned by the GET_STATION_LIST_DESIGNATOR call if there is an invalid station-list name.	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–19 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_STATION_LIST_DESIGNATOR service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_STATION_LIST_DESIGNATOR  IN DCILIBRARY"
     USING <station-list name>,
     GIVING <station-list designator>.
```

ALGOL

```
<station-list designator> :=
      GET_STATION_LIST_DESIGNATOR
          (<station-list name>);
```

Station Names

When you pass a station designator to the GET_STATION_NAME service function, the COMS library returns the following information:

- A valid station name with a length of 1 to 255 alphanumeric characters.
- A device designator for the station.
- A station-security designator.
- A logical station number (LSN). If this number is 0, then the station is disconnected.
- A virtual terminal.
- A screen size.

When you pass a message to a COMS direct-window program from the ODT using the format *<mix # of COMS> SM PASS <window> <text>*, COMS puts a station designator in the input CD. The input CD passes the station designator to the GET_STATION_NAME service function, and then the COMS library returns ODT as a valid station name.

Origin of Input Parameter

You can obtain a station designator from either of the following sources:

- The COMS-in-Station field of the input CD in an application program
- The GET_STATION_DESIGNATOR service function

Programming Requirements

Table F–20 shows the input parameters you pass to the GET_STATION_NAME service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–20. GET_STATION_NAME Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Station designator	COBOL74: Binary	Integer
Output from COMS	Station name	COBOL74: Display	EBCDIC array
Output from COMS	Device designator	COBOL74: Binary	Integer
Output from COMS	Station-security designator	COBOL74: Binary	Integer
Output from COMS	LSN	COBOL74: Binary	Integer
Output from COMS	Virtual terminal	COBOL74: Binary	Integer
Output from COMS	Screen size	COBOL74: Binary	Integer

continued

Table F–20. GET_STATION_NAME Parameters (cont.)

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Output from COMS	Function value	The following function values can be returned by the GET_STATION_NAME call: • 1 = No errors • 0 = One of the following has occurred: – An invalid designator was used – The station was no longer logged on to COMS	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–20 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_STATION_NAME service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_STATION_NAME  IN DCILIBRARY"
     USING <station designator>,
           <station name>,
           <device designator>,
           <LSN>, <virtual terminal>, <screen size>
           <station-security designator>
     GIVING <result of  GET_STATION_NAME
             service function>.
```

ALGOL

```
<result of  GET_STATION_NAME  service function> :=
      GET_STATION_NAME  (<station designator>,
          <station name>, <device designator>, <virtual terminal>,
          <screen size>, <LSN>, <station-security designator>);
```

Message Date and Time

You can use the GET_DATE service function to get the message date and time.

When you pass a 1-word timestamp in TIME(6) format to this service function, the COMS library returns a six-character display field representing the time portion of the timestamp. The six-character field displays the date in the month/day/year (MMDDYY) format, and the time in the hours/minutes/seconds (HHMMSS) format.

Origin of Input Parameter

You can obtain a timestamp from the COMS-in-Timestamp field in the input CD of an application program.

Programming Requirements

Table F–21 shows the input parameters you pass to the GET_DATE service function and the output parameters that the COMS library returns. The table also provides data types for the parameters in ALGOL and COBOL, and the values that can be returned by the service function call.

Table F–21. GET_DATE Parameters

Direction	Parameter	COBOL Data Type	ALGOL Data Type
Input to COMS	Timestamp	COBOL74: Binary	Integer
Output from COMS	Date	COBOL74: Display	EBCDIC array
Output from COMS	Time	COBOL74: Display	EBCDIC array
Output from COMS	Function value	The following function values can be returned by the GET_DATE call: • 1 = No errors • 0 = Invalid timestamp	

For COBOL74, declare each variable as a PIC S9(11) (1-word) field. Declare the binary data type shown in Table F–21 at level 77, and the display data types at level 01.

Examples

Following are examples of COBOL and ALGOL code for calling the GET_DATE service function from an application program or an agenda processing item:

COBOL74

```
CALL "GET_DATE  IN DCILIBRARY"
     USING <timestamp>,
           <date>,
           <time>,
     GIVING <result of  GET_DATE  service function>.
```

ALGOL

```
<result of  GET_DATE  service function> :=
     GET_DATE  (<timestamp>, <date>, <time>);
```

Glossary

This glossary provides the definitions for selected terms used in this programming guide. The terms are presented in alphabetical order.

A

abort

To terminate an active program or session abnormally and, sometimes, to attempt to restart it.

ADDS

See Advanced Data Dictionary System.

Advanced Data Dictionary System (ADDS)

A software product that allows for the centralized definition, storage, and retrieval of data descriptions.

agenda

An entity used for message routing that consists of a processing-item list and a destination. An agenda can be applied to messages that are received or sent by application programs.

Agenda Processor library

An internal library that applies processing items to messages by executing the processing-item lists that agendas specify.

ALGOL

Algorithmic language. A structured, high-level programming language that provides the basis for the stack architecture of the Unisys A Series systems. ALGOL was the first block-structured language developed in the 1960s and served as a basis for such languages as Pascal and Ada. It is still used extensively on A Series systems, primarily for systems programming.

audit trail

In Data Management System II (DMSII), a file produced by the Access routines that contains various control records and a sequence of before-update and after-update record images resulting from changes to the database. The audit trail is used to recover the database and supply restart information to programs after a hardware or software failure has occurred.

audited database

In Data Management System II (DMSII) and in the InfoExec environment, a database that stores a record of changes (called the audit trail), which can be used for database recovery if a hardware or software failure occurs.

B

batch mode

An execution mode in which a program running under COMS can do batch-type updates to a database shared by other transaction processors.

BNA

The network architecture used on A Series, B 1000, and V Series systems as well as CP 9500 and CP 2000 communications processors to connect multiple, independent, compatible computer systems into a network for distributed processing and resource sharing.

C

CANDE

See Command and Edit.

casual output

Any output that is not protected from loss in the event of a system failure.

CD

See communication description.

COBOL

Common Business-Oriented Language. A widely used, procedure-oriented language intended for use in solving problems in business data processing. The main characteristics of COBOL are the easy readability of programs and a considerable degree of machine independence. COBOL is the most widely used procedure-oriented language.

Command and Edit (CANDE)

A time-sharing message control system (MCS) that enables a user to create and edit files, and develop, test, and execute programs, interactively.

communication description (CD)

A message header passed with the message data received and sent by application programs that interface with a message control system (MCS). The CD provides routing information about the message data and allows use of other functions, depending on the MCS.

Communications Management System (COMS)

A general message control system (MCS) that controls online environments on A Series systems. COMS can support the processing of multiprogram transactions, single-station remote files, and multistation remote files. *See also* COMS (Full-Featured) and COMS (Kernel).

Communications Processor 2000 (CP 2000)

A data communications processor (DCP) that provides communications interfaces to local area networks (LANs) and wide area networks (WANs), including BNA Version 2 and Transmission Control Protocol/Internet Protocol (TCPIP) networks. The CP 2000 also provides connections to terminals controlled by BNA Version 2 software.

COMS

See Communications Management System.

COMS Control library

A Communications Management System (COMS) internal library that initiates a database (DB) library for each database that uses synchronized recovery, and initiates a transaction processor (TP) library for nondatabase, transaction-processing programs that do not use synchronized recovery.

COMS (Full-Featured)

A version of the Communications Management System (COMS) message control system (MCS) that provides full configuration capabilities through the COMS Utility. The COMS Utility enables the user to define and customize a COMS transaction processing environment, which provides additional features like transaction-based routing and database recovery. In addition, the user can track COMS statistics and use GEMCOS migration aids.

COMS (Kernel)

The transitional, temporary version of the Communications Management System (COMS) message control system (MCS). COMS creates a predefined configuration file that enables the user to use the window feature with the following three windows: a Menu-Assisted Resource Control (MARC) window with eight dialogues, a Command and Edit (CANDE) window with two dialogues, and a Generalized Message Control System (GEMCOS) window with one dialogue. Additionally, the user can communicate with remote-file programs.

COMS library

The library that is created upon the execution of a FREEZE statement after the SYSTEM/COMS object code initiates as internal processes the Router library, the Agenda Processor library, and the COMS Control library. The COMS library contains service functions for designator conversion, dynamic selection procedures for linking callers to other libraries within the COMS system, and support for dynamic table changes.

COMS network

A system of interconnected elements consisting of at least one computer system and one or more stations for which the Communications Management System (COMS) provides communication and processing control.

COMS tables

The forms and data that comprise the information defined and maintained by means of the COMS Utility. These forms are included in the Communications Management System (COMS) configuration file.

COMS Utility

The Communications Management System (COMS) program that defines and maintains the specifications stored in the COMS configuration file.

configuration file

A file that contains descriptions of the tables defined through the COMS Utility program. These tables contain information on message routing, security, dynamic program control, and synchronized recovery. This file is also referred to as the COMS CFILE.

conversation area
The user data space in the header of a message. The conversation area is user defined and can contain information passed by a program or processing item. When used with a direct-window interface, this area contains the telephone number to be dialed.

CP 2000
See Communications Processor 2000.

current transaction
In Data Management System II (DMSII), the transaction that is attempting to update the database at the moment that a transaction-state abort or system failure occurs.

D

DASDL
See Data and Structure Definition Language.

Data and Structure Definition Language (DASDL)
In Data Management System II (DMSII), the language used to describe a database logically and physically, and to specify criteria to ensure the integrity of data stored in the database. DASDL is the source language that is input to the DASDL compiler, which creates or updates the database description file from the input.

data comm
See data communications.

data communications (data comm)
The transfer of data between a data source and a data sink (two computers, or a computer and a terminal) by way of one or more data links, according to appropriate protocols.

data communications interface (DCI) library
A library that serves as the direct programmatic interface to the Communications Management System (COMS). Application programs must communicate with COMS through the DCI library to use agendas, processing items, routing by trancode, and synchronized recovery.

Data Management System II (DMSII)
A specialized system software package used to describe a database and maintain the relationships among the data elements in the database.

database (DB)
An integrated, centralized system of data files and program utilities designed to support an application. The data sets and associated index structures are defined by a single description. Ideally, all the permanent data pertinent to a particular application resides in a single database. The database is considered a global entity that several applications can access and update concurrently.

DB

See database.

DB control program

A process that initiates all application programs using synchronized recovery with a database and detects transaction-state aborts that occur during the processing of the database. Each database being synchronized has its own DB control program.

DB library

The data communications interface (DCI) library for programs that are controlled by a common database (DB) control program.

DCI library

See data communications interface (DCI) library.

DCIENTRYPOINT

An entry point of the data communications interface (DCI) library. A compiler automatically generates code calling this entry point whenever an application program executes an ENABLE, RECEIVE, or SEND statement.

DCIWAITENTRYPOINT

An entry point of the data communications interface (DCI) library. An ALGOL application can call DCIWAITENTRYPOINT when the application is waiting for the DCI input to arrive and other independent, application-visible events to happen.

default

Pertaining to a value automatically assigned by a program or system when another value has not been specified by the user.

default agenda

The agenda that the Communications Management System (COMS) applies to a message if the message does not contain a trancode, or if the message contains a trancode value that has not been defined for the system. A default agenda is assigned to each direct window with the COMS Utility program. There can be one default input agenda and one default output agenda for each window.

designator

A binary number that is part of an internal code used in the table structure. By using designators in programs that run under COMS, the programmer can control messages symbolically rather than by communicating directly with entities in the data communications environment.

direct window

A type of window that enables the user to route messages directly to COMS, while using all the COMS capabilities for preprocessing and postprocessing of messages.

DMSII

See Data Management System II.

DMSII recovery

In Data Management System II (DMSII), a database routine that is initiated after a hardware, software, or operations failure while a database is in the update mode. DMSII recovery backs out any partially completed transactions by applying audit-trail images to the database to restore it to its proper state. It also passes restart information to the programs accessing the database.

DMTERMINATE procedure

A system-level Data Management System II (DMSII) procedure that a database processing program can invoke at any time to display a standard, recognizable error message and to discontinue the program.

E

EBCDIC

Extended Binary Coded Decimal Interchange Code. An 8-bit code representing 256 graphic and control characters that are the native character set of most mainframe systems.

EBCDIC array

In ALGOL, an array whose elements are EBCDIC characters.

echo program

A simple program that echoes or returns input messages as output. The input source and output destination can be any devices defined in the program.

element

A specifically defined item within an entity category of the configuration file.

enabled

Referring to a station that is being polled (invited to transmit in a certain order) and that can communicate with the system.

end of file (EOF)

A code at the end of a data file that signals that the last record in the file has been processed.

end of job (EOJ)

The control code that signals the receiver that a job has completed.

end of task (EOT)

The termination of processing of a task.

entity

A category of items within the configuration file.

entry point

A procedure or function that is in a library object.

EOF

See end of file.

EOJ

See end of job.

EOT

See end of task.

EXCEPTIONTASK

A task-valued task attribute that the operating system sets by default to the parent task. The EXCEPTIONTASK attribute is used to link an application program to the data communications interface (DCI) library of the Communications Management System (COMS) during program initialization.

F

file switch

The act of stopping the processing on one file and starting the processing on another file. This switch can be done automatically by the system or manually by the user.

G

GEMCOS

See Generalized Message Control System.

Generalized Message Control System (GEMCOS)

A message control system (MCS) developed for online systems. GEMCOS is transaction oriented.

H

halt/load

A system-initialization procedure that temporarily halts the system and loads the operating system from a disk to main memory.

header

A sequence of characters preceding the text of a message, containing routing or other communications-related information.

I

installation

A single computer configuration, facility, center, or system consisting of one or more mainframes and any possible combination of peripheral, communications, I/O, and other types of support devices.

INVALID OP

An error that occurs when a character or sequence of characters is not in accordance with the expected character or sequence.

L

library

A collection of one or more named routines or library objects that are stored in a file and can be accessed by other programs.

logical station number (LSN)

In Network Definition Language II (NDLII), a unique number assigned to each station in a network. Each station has an LSN assigned according to the order in which the stations are defined in NDLII. The first defined station is 0000.

LSN

See logical station number.

M

MARC

See Menu-Assisted Resource Control.

Master Control Program (MCP)

An operating system on A Series systems. The MCP controls the operational environment of the system by performing job selection, memory management, peripheral management, virtual memory management, dynamic subroutine linkage, and logging of errors and system utilization.

MCP

See Master Control Program.

MCS

See message control system.

Menu-Assisted Resource Control (MARC)

A menu-driven interface to A Series systems that also enables direct entry of commands.

message

Any information-containing data unit, in an ordered format, sent by means of a communications process to a named network entity or interface. A message contains the information (text portion) and controls for routing and handling (header portion). In Data Communications ALGOL (DCALGOL), a special form of array. Two types of messages are recognized by a message control system (MCS): those used with DCALGOL *DCWRITE* statements and those generated elsewhere in the data communications subsystem that appear in an MCS queue.

message control system (MCS)

A program that controls the flow of messages between terminals, application programs, and the operating system. MCS functions can include message routing, access control, audit and recovery, system management, and message formatting.

message header

A sequence of characters, preceding the text of a message, that contains routing or descriptive information for the message.

MFI

See module function index.

MLS

See multilingual system.

module function index (MFI)

An integer value that represents a transaction code or group of transaction codes that are used to route forms or messages.

MultiLingual System (MLS)

An environment that can process information using the standards and functional requirements of different localities, cultures, or lines of business. The processing of information depends on the ccsversion, languages, and conventions that are defined for the system. For example, output messages, online help text, menus, and screens can be developed and accessed in different natural languages, such as English, French, or Japanese.

multiprogram environment

An environment in which a software system handles multiple routines or programs simultaneously by overlapping or interleaving their execution, permitting more than one program to timeshare machine components.

N

NDLII

See Network Definition Language II.

Network Definition Language II (NDLII)

The Unisys language used to describe the physical, logical, and functional characteristics of the data communications subsystem to network support processors (NSPs), line support processors (LSPs), and data communications data link processors (DCDLPs).

network support processor (NSP)

A data communications subsystem processor that controls the interface between a host system and the data communications peripherals. The NSP executes the code generated by the Network Definition Language II (NDLII) compiler for line control and editor procedures. An NSP can also control line support processors (LSPs).

NSP

See network support processor.

P

parameter

An identifier associated in a special way with a procedure. A parameter is declared in the procedure heading and is automatically assigned a value when the procedure is invoked.

Pascal

A high-level programming language developed by Niklaus Wirth, based on the block structuring nature of ALGOL 60 and the data structuring innovations of C.A.R. Hoare. Pascal is a general-purpose language.

password

A character string associated with a usercode or accesscode in the USERDATAFILE, and used to identify legitimate users of the system. When logging on to a message control system (MCS), a user must supply a usercode and a password.

peripheral

A device used for input, output, or file storage. Examples are magnetic tape drives, disk drives, printers, or operator display terminals (ODTs). *Synonym for* peripheral device.

POF

See protected output file.

postprocessing

The processing done to a message by processing items after an application program sends out the message.

preprocessing

The processing that the Agenda Processor performs on a message before an application program receives the message.

process switching

An event that occurs when the operating system tells the central processing unit (CPU) to execute a different program.

processing item

A procedure, contained in a processing-item library, used for processing a message.

processing-item library

A user-written ALGOL library containing a set of procedures called processing items. A processing-item library can be called only by the COMS Agenda Processor library to preprocess and postprocess messages as they are received and sent by programs.

protected database

A database that is associated with a protected window.

protected dialogue

A dialogue on a protected window, which has the protected output feature enabled.

protected output feature

A feature that enables output messages to be written to a disk file and then sent to their destinations only after successful completion of the transaction.

protected output file (POF)

A disk file that contains protected output messages. These messages are sent to their destinations after a transaction has been completed.

protected window

A window that has the protected output feature enabled.

Q

queue

A data structure used for storing objects; the objects are removed in the same order they are stored. In Data Communications ALGOL (DCALGOL), a linked list of messages.

R

remote file

A file with the KIND attribute specified as REMOTE. A remote file enables object programs to communicate interactively with a terminal.

Remote Print System (ReprintS)

A Unisys software system that controls the routing and printing of backup files at remote (data comm) destinations and on BNA networks.

Report Program Generator (RPG)

A high-level commercially oriented programming language used most frequently to produce reports based on information derived from data files.

ReprintS

See Remote Print System.

reproducibility

The ability of a sequence of transactions to be reproduced under the same conditions and to achieve the same results as the original transactions.

restart

To return to a particular point in a program and resume operation from that point.

restart data set (RDS)

In Data Management System II (DMSII), a data set containing restart records that application programs can access to recover database information after a system failure.

restart record

A record containing information stored by update programs that enables the programs to restart in response to particular conditions. For each update program, COMS saves restart records in the transaction trail along with the corresponding images of the input header and the message data.

restartable application program

An application program that can resume processing automatically after an interruption such as a halt/load or an abort.

rollback

The recovery of a database or transaction base to a consistent state at an earlier point in time.

Router library

An internal library containing input-router and output-router entry points called by the Master Control Program (MCP). The input-router entry point is primarily called by the data communications controller (DCC) for all input-message and output-message results associated with any station controlled by COMS. The output-router entry point is called from logical I/O for output from remote files.

RPG

See Report Program Generator.

S

Screen Design Facility (SDF)

The InterPro product used for creating forms for online, transaction-based application systems.

SDF

See Screen Design Facility.

security category

A designation that provides access security for programs, stations, transaction codes, and usercodes. Up to 32 security categories can be defined for an installation.

security-category list

A group of several security categories that can be assigned to certain entities to accomplish forms of security.

Semantic Information Manager (SIM)

The basis of the InfoExec environment. SIM is a database management system used to describe and maintain associations among data by means of subclass-superclass relationships and linking attributes.

service function

An integer procedure of the Communications Management System (COMS) library that enables the user to access subroutines that can do the following: translate a designator to a name that represents a COMS entity; translate a name that represents a COMS entity to a designator; or obtain additional information about the name or designator passed to the service function.

session security

The intersection of the security categories assigned to a station and the security categories assigned to the usercode of the person who is logged on to that station.

SIM

See Semantic Information Manager.

stack

A region of memory used to store data items in a particular order, usually on a last-in, first-out basis. *Synonym for* process stack.

state

The condition of one or all the units or elements of a computer system.

station

A data structure that relates a logical connection to either a terminal device or a pseudostation.

string

A connected sequence or group of characters.

subroutine

A self-contained section of a program to which program control is transferred when the subroutine is invoked and that transfers control back to the point of invocation when it is exited.

synchronized recovery

A function that resubmits incomplete transactions to the database after a transaction-state abort, system crash, or rollback occurs. This COMS function is called synchronized recovery because it reprocesses transactions in the same order that they were originally processed by multiple programs running asynchronously, even if the transactions were conflicting.

syncpoint

In Data Management System II (DMSII), a point in time when no program is in a transaction state.

syntax

The rules or grammar of a language.

T

tanked messages

Incoming messages that are being deferred from display at a station because the associated window is suspended.

task

A single, complete unit of work performed by the system, such as compiling or executing a program, or copying a file from one disk to another. Tasks are initiated by a job, by another task, or directly by a user.

TBR

See transaction-based routing.

terminal

An I/O device designed to receive or send source data in a network.

text

The part of a message containing information that has an ultimate purpose and destination beyond the data communications subsystem.

throughput

The total useful information processed during a specified time period.

TIME(6) format

In ALGOL, a system format that returns a unique number representing the time and date (a timestamp) in the following form: 0 & (JULIANDATE – 70000) [47:16] & (TIME (11) DIV 16) [31:32]

timestamp

An encoded, 48-bit numerical value for the time and date. Various timestamps are maintained by the system for each disk file. Timestamps note the time and date a file was created, last altered, and last accessed.

TP library

See transaction processor (TP) library.

TP-to-TP message

Any output message directed to a program.

TPS

See transaction processing system.

trancode

See transaction code.

transaction

The transfer of one message from a terminal or host program to a receiving host program, the processing carried out by the receiving host program, and the return of an answer to the sender.

transaction code (trancode)

A code that can appear in a transaction-initiating message header, indicating the processing that is to be carried out. This code is used to route the message to the appropriate host program.

transaction processing system (TPS)

A Unisys system that provides methods for processing a high volume of transactions, keeps track of all input transactions that access the database, enables the user to batch data for later processing, and enables transactions to be processed on a database that resides on a remote system.

transaction processor (TP) library

The data communications interface (DCI) library for application programs that use the Communications Management System (COMS).

transaction state

In Data Management System II (DMSII), the period in a user-language program between a begin transaction operation and an end transaction operation.

transaction trail

A file maintained by a Communications Management System (COMS) database (DB) library that contains a series of time-ordered transactions that can be reapplied to the database to provide synchronized recovery in the event of a transaction-state abort, system crash, or rollback. The file can also be used to provide a journal of both query and update transactions for security auditing, accounting, and statistical reporting. Each DB library has its own transaction trail.

transaction-based routing (TBR)

A Communications Management System (COMS) capability that routes messages according to the transaction codes they contain.

two-phase transaction

A transaction in which the first execution phase locks records without freeing any, the second and final execution phase of the transaction frees records without locking any, and no records are retrieved without locking them.

U

update

To delete, insert, or modify information in a database or transaction base.

usercode

An identification code used to establish user identity and control security, and to provide for segregation of files. Usercodes can be applied to every task, job, session, and file on the system. A valid usercode is identified by an entry in the USERDATAFILE.

USERDATAFILE

A system database that defines valid usercodes and contains various data about each user (such as accesscodes, passwords, and chargecodes) and the population of users for a particular installation.

V

virtual terminal (VT)

A terminal attribute that identifies the text postprocessing algorithm to be applied to data messages sent to the terminal.

VT

See virtual terminal.

W

WFL

See Work Flow Language.

window

The concept that enables a number of program environments to be operated independently and simultaneously at one station. One of the program environments can be viewed while the others continue to operate.

word

A unit of computer memory. On A Series systems, a word consists of 48 bits used for storage plus tag bits used to indicate how the word is interpreted.

Work Flow Language (WFL)

A Unisys language used for constructing jobs that compile or run programs on A Series systems. WFL includes variables, expressions, and flow-of-control statements that offer the programmer a wide range of capabilities with regard to task control.

Bibliography

A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation (form 8600 0098). Unisys Corporation.

A Series ALGOL Programming Reference Manual, Volume 2: Product Interfaces (form 8600 0734). Unisys Corporation.

A Series Binder Programming Reference Manual (form 8600 0304). Unisys Corporation.

A Series BNA Version 2 Capabilities Overview (form 1182318). Unisys Corporation.

A Series BNA Version 2 Operations Guide (form 1222720). Unisys Corporation.

A Series CANDE Configuration Reference Manual (form 8600 1344). Unisys Corporation.

A Series COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation (form 8600 0296). Unisys Corporation.

A Series COBOL ANSI-74 Programming Reference Manual, Volume 2: Product Interfaces (form 8600 0130). Unisys Corporation.

A Series COBOL ANSI-74 Test and Debug System (TADS) Programming Guide (form 1169901). Unisys Corporation.

A Series Communications Management System (COMS) Capabilities Manual (form 8600 0627). Unisys Corporation.

A Series Communications Management System (COMS) Configuration Guide (form 8600 0312). Unisys Corporation.

A Series Communications Management System (COMS) Migration Guide (form 8600 1567). Unisys Corporation.

A Series Communications Management System (COMS) Operations Guide (form 8600 0833). Unisys Corporation.

A Series DMSII Application Program Interfaces Programming Guide (form 5044225). Unisys Corporation. Formerly *A Series DMSII User Language Interface Programming Guide.*

A Series DMSII Utilities Operations Guide (form 8600 0759). Unisys Corporation.

A Series I/O Subsystem Programming Guide (form 8600 0056). Unisys Corporation. Formerly *A Series I/O Subsystem Programming Reference Manual.*

A Series InfoExec Semantic Information Manager (SIM) Object Manipulation Language (OML) Programming Guide (form 8600 0163). Unisys Corporation.

A Series InfoExec Semantic Information Manager (SIM) Technical Overview (form 8600 1674). Unisys Corporation.

A Series Mark 3.9 Software Release Capabilities Overview (form 8600 0015). Unisys Corporation.

A Series Menu-Assisted Resource Control (MARC) Operations Guide (form 8600 0403). Unisys Corporation.

A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation (form 8600 0080). Unisys Corporation.

A Series Pascal Programming Reference Manual, Volume 2: Product Interfaces (form 8600 1294). Unisys Corporation.

A Series Report Program Generator (RPG) Programming Reference Manual, Volume 1: Basic Implementation (form 8600 0544). Unisys Corporation.

A Series Report Program Generator (RPG) Programming Reference Manual, Volume 2: Product Interfaces (form 8600 0742). Unisys Corporation.

A Series Screen Design Facility (SDF) Operations and Programming Guide (form 1185295). Unisys Corporation.

A Series Security Administration Guide (form 8600 0973). Unisys Corporation.

A Series Work Flow Administration and Programming Guide (form 1170149). Unisys Corporation.

B 1000 Series to A Series Progression Guide (form 8600 0619). Unisys Corporation.

Index

UNISYS

Help Us To Help You

Publication Title

Form Number

Unisys Corporation is interested in your comments and suggestions reguarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

□ Addition □ Deletion □ Revision □ Error

Comments:

Name Telephone number ( )

Title Company

Address

City State Zip code

Cut along dotted line ✂

Tape

Please Do Not Staple

Tape

Fold Here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 817 DETROIT, MI

POSTAGE WILL BE PAID BY ADDRESSEE

UNISYS CORPORATION
ATTN: PUBLICATIONS
25725 JERONIMO ROAD
MISSION VIEJO, CA 92691-9826