

# iWay

iWay Stored Procedures Reference  
Version 5 Release 2.0

Cactus, EDA/SQL, FIDEL, FOCCALC, FOCUS, FOCUS Fusion, FOCUS Vision, Hospital-Trac, Information Builders, the Information Builders logo, Parlay, PC/FOCUS, SmartMart, SmartMode, SNAPPack, TableTalk, WALDO, Web390, WebFOCUS and WorldMART are registered trademarks and EDA, iWay, and iWay Software are trademarks of Information Builders, Inc.

Acrobat and Adobe are registered trademarks of Adobe Systems Incorporated.

Allaire and JRun are trademarks of Allaire Corporation.

NOMAD is a registered trademark of Aonix.

UniVerse is a registered trademark of Ardent Software, Inc.

WebLogic is a registered trademark of BEA Systems, Inc.

SUPRA and TOTAL are registered trademarks of Cincom Systems, Inc.

Impromptu is a registered trademark of Cognos.

Alpha, DEC, DECnet, and NonStop are registered trademarks and Tru64, OpenVMS, and VMS are trademarks of Compaq Computer Corporation.

CA-ACF2, CA-Datcom, CA-IDMS, CA-Top Secret, and Ingres are registered trademarks of Computer Associates International, Inc.

MODEL 204 and M204 are registered trademarks of Computer Corporation of America.

Paradox is a registered trademark of Corel Corporation.

StorHouse is a registered trademark of FileTek, Inc.

HP MPE/iX is a registered trademark of Hewlett Packard Corporation.

Informix is a registered trademark of Informix Software, Inc.

Intel is a registered trademark of Intel Corporation.

ACF/VTAM, AIX, AS/400, CICS, DB2, DRDA, Distributed Relational Database Architecture, IBM, MQSeries, MVS, OS/2, OS/390, OS/400, RACF, RS/6000, S/390, VisualAge, VM/ESA, and VTAM are registered trademarks and DB2/2, Hiperspace, IMS, MVS/ESA, QMF, SQL/DS, VM/XA and WebSphere are trademarks of International Business Machines Corporation.

INTERSOLVE and Q+E are registered trademarks of INTERSOLVE.

Orbit is a registered trademark of Iona Technologies Inc.

Approach and DataLens are registered trademarks of Lotus Development Corporation.

ObjectView is a trademark of Matesys Corporation.

ActiveX, FrontPage, Microsoft, MS-DOS, PowerPoint, Visual Basic, Visual C++, Visual FoxPro, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

Teradata is a registered trademark of NCR International, Inc.

Netscape, Netscape FastTrack Server, and Netscape Navigator are registered trademarks of Netscape Communications Corporation.

NetWare and Novell are registered trademarks of Novell, Inc.

INFOAccess is a trademark of Pioneer Systems, Inc.

Progress is a registered trademark of Progress Software Corporation.

SAP and SAP R/3 are registered trademarks and SAP Business Information Warehouse and SAP BW are trademarks of SAP AG.

Silverstream is a trademark of Silverstream Software.

ADABAS is a registered trademark of Software A.G.

CONNECT:Direct is a trademark of Sterling Commerce.

Java and all Java-based marks, NetDynamics, Solaris, SunOS, and iPlanet are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

PowerBuilder and Sybase are registered trademarks and SQL Server is a trademark of Sybase, Inc.

Unicode is a trademark of Unicode, Inc.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Due to the nature of this material, this document refers to numerous hardware and software products by their trade names. In most, if not all cases, these designations are claimed as trademarks or registered trademarks by their respective companies. It is not this publisher's intent to use any of these names generically. The reader is therefore cautioned to investigate all claimed trademark rights before using any of these names other than to refer to the product described.

Copyright © 2002, by Information Builders, Inc. All rights reserved. This manual, or parts thereof, may not be reproduced in any form without the written permission of Information Builders, Inc.

Printed in the U.S.A.

---

---

## Preface

The iWay Stored Procedures Reference manual provides information about programs or procedures, called stored procedures, that reside on a server and are called by connector applications.

Stored procedures allow you to build on existing applications to create new client/server applications for the desktop environment.

This manual is intended for the API Programmer, the Dialogue Manager Programmer, and others who develop and maintain client/server applications that call stored procedures.

---

## How This Manual Is Organized

This manual includes the following chapters:

Chapter/Appendix		Contents
1	Introduction	Describes the types of stored procedures, how they are called, and their execution order. Explains why stored procedures are used.
2	Calling a Program as a Stored Procedure	Describes two ways to call a compiled program: using the API function call EDARPC, and using the commands CALLPGM or EXEC in a Dialogue Manager procedure. Addresses the use of parameters.
3	Writing a 3GL Compiled Stored Procedure Program	Describes the requirements for writing a program to be called by EDARPC, or by CALLPGM in a Dialogue Manager procedure. Addresses the control block used for communication between the server and the program; storage of program values; error handling; and the command CREATE TABLE, which a program issues to describe the answer set it is returning.
4	Writing a Dialogue Manager Procedure	Describes the features of the Dialogue Manager language, including the syntax and use of Dialogue Manager commands and how they are processed by the server.

Chapter/Appendix		Contents
<b>5</b>	Transaction Adapters for IMS/TM	Describes the Transaction Adapter for IMS, which enables connector applications to invoke IMS/TM transactions. Describes the syntax and use of the function CALLIMS, which is included in a Dialogue Manager procedure to establish a connection to the IMS/TM environment and invoke the transaction.
<b>6</b>	Platform-specific Commands	Describes the syntax and use of platform-specific commands that can be included in a Dialogue Manager procedure, such as DYNAM in MVS.
<b>A</b>	Dialogue Manager Quick Reference	Includes all Dialogue Manager commands, with their syntax, in alphabetical order for easy reference.
<b>B</b>	GENCPGM Usage	Describes how to use the script that has been created for UNIX, Windows NT/2000, and OpenVMS to assist in simple compilations.

## Documentation Conventions

---

The following conventions apply throughout this manual:

Convention	Description
<code>THIS TYPEFACE</code> or <code>this typeface</code>	Denotes syntax that you must enter exactly as shown.
<i>this typeface</i>	Represents a placeholder (or variable) in syntax for a value that you or the system must supply.
<u>underscore</u>	Indicates a default setting.
<i>this typeface</i>	Represents a placeholder (or variable) in a text paragraph, a cross-reference, or an important term.
<b>this typeface</b>	Highlights a file name or command in a text paragraph that must be lowercase.
<i>this typeface</i>	Indicates a button, menu item, or dialog box option you can click or select.
Key + Key	Indicates keys that you must press simultaneously.

Convention	Description
{ }	Indicates two or three choices; type one of them, not the braces.
[ ]	Indicates a group of optional parameters. None are required, but you may select one of them. Type only the parameter in the brackets, not the brackets.
	Separates mutually exclusive choices in syntax. Type one of them, not the symbol.
...	Indicates that you can enter a parameter multiple times. Type only the parameter, not the ellipsis points (...).
.	Indicates that there are (or could be) intervening or additional commands.

## Related Publications

---

Visit our World Wide Web site, <http://www.iwaysoftware.com>, to view a current listing of our publications and to place an order. You can also contact the Publications Order Department at (800) 969-4636.

## Customer Support

---

Do you have questions about iWay Stored Procedures?

Call Information Builders Customer Support Service (CSS) at (800) 736-6130 or (212) 736-6130. Customer Support Consultants are available Monday through Friday between 8:00 a.m. and 8:00 p.m. EST to address all your iWay Stored Procedures questions. Information Builders consultants can also give you general guidance regarding product capabilities and documentation. Please be ready to provide your six-digit site code (xxxx.xx) when you call.

You can also access support services electronically, 24 hours a day, with InfoResponse Online. InfoResponse Online is accessible through our World Wide Web site, <http://www.informationbuilders.com>. It connects you to the tracking system and known-problem database at the Information Builders support center. Registered users can open, update, and view the status of cases in the tracking system and read descriptions of reported software issues. New users can register immediately for this service. The technical support section of [www.informationbuilders.com](http://www.informationbuilders.com) also provides usage techniques, diagnostic tips, and answers to frequently asked questions.

To learn about the full range of available support services, ask your Information Builders representative about InfoResponse Online, or call (800) 969-INFO.

## Information You Should Have

---

To help our consultants answer your questions most effectively, be ready to provide the following information when you call:

- Your six-digit site code (xxxx.xx).
- Your iWay Software configuration:
  - The iWay Software version and release.
  - The communications protocol (for example, TCP/IP or LU6.2), including vendor and release.
- The stored procedure (preferably with line numbers) or SQL statements being used in server access.
- The database server release level.
- The database name and release level.
- The Master File and Access File.
- The exact nature of the problem:
  - Are the results or the format incorrect? Are the text or calculations missing or misplaced?
  - The error message and return code, if applicable.
  - Is this related to any other problem?
- Has the procedure or query ever worked in its present form? Has it been changed recently? How often does the problem occur?
- What release of the operating system are you using? Has it, your security system, communications protocol, or front-end software changed?
- Is this problem reproducible? If so, how?
- Have you tried to reproduce your problem in the simplest form possible? For example, if you are having problems joining two data sources, have you tried executing a query containing just the code to access the data source?
- Do you have a trace file?
- How is the problem affecting your business? Is it halting development or production? Do you just have questions about functionality or documentation?

## **User Feedback**

---

In an effort to produce effective documentation, the Documentation Services staff welcomes any opinion you can offer regarding this manual. Please use the Reader Comments form at the end of this manual to relay suggestions for improving the publication or to alert us to corrections. You can also use the Documentation Feedback form on our Web site, <http://www.iwaysoftware.com>.

Thank you, in advance, for your comments.

## **iWay Software Training and Professional Services**

---

Interested in training? Our Education Department offers a wide variety of training courses for iWay Software and other Information Builders products.

For information on course descriptions, locations, and dates, or to register for classes, visit our World Wide Web site (<http://www.iwaysoftware.com>) or call (800) 969-INFO to speak to an Education Representative.

Interested in technical assistance for your implementation? Our Professional Services department provides expert design, systems architecture, implementation, and project management services for all your business integration projects. For information, visit our World Wide Web site (<http://www.iwaysoftware.com>).



---

---

# Contents

<b>1. Introduction</b>	<b>1-1</b>
What Is a Stored Procedure?	1-2
Why Use Stored Procedures?	1-2
Calling Stored Procedures	1-2
Setting the Execution Order	1-3
Stored Procedure Libraries	1-4
Valid EXORDER Settings	1-5
Execution Order of Stored Procedures From Dialogue Manager	1-6
<b>2. Calling a Program as a Stored Procedure</b>	<b>2-1</b>
Calling a Compiled Program	2-2
Calling a Program With EDARPC	2-3
Calling a Program With CALLPGM	2-4
Switching Plans in DB2 (MVS Only)	2-6
Passing Parameters	2-7
Using CALLPGM	2-8
Using EDARPC	2-8
Program Communication	2-13
<b>3. Writing a 3GL Compiled Stored Procedure Program</b>	<b>3-1</b>
Program Requirements	3-2
Setting Up the Control Block	3-2
Control Block Fields	3-3
Storing Program Values	3-9
OpenVMS COBOL	3-18
Error Handling	3-19
Issuing the CREATE TABLE Command	3-21

<b>4. Writing a Dialogue Manager Procedure</b>	<b>4-1</b>
Commands Included in a Procedure	4-2
Commands and Processing	4-3
Dialogue Manager Processing	4-5
Commenting a Procedure	4-7
Sending a Message to a Client Application	4-8
Controlling Execution	4-9
Executing Stacked Commands: -RUN	4-9
Executing Stacked Commands and Exiting the Procedure: -EXIT	4-10
Canceling Execution: -QUIT	4-11
Using Variables	4-12
Naming Conventions	4-12
Local Variables	4-13
Global Variables	4-16
System Variables	4-17
Variables and Command Structures	4-20
Supplying Values for Variables	4-20
General Rules	4-21
Supplying Values in the EXEC Command	4-21
Debugging Execution Flow	4-25
The -DEFAULTS Command	4-25
The -SET Command	4-27
The -READ Command	4-28
Branching	4-31
Screening Values With -IF Tests	4-35
Looping	4-38
Ending a Loop	4-39
Calling Another Procedure	4-41
Nesting	4-43
The EXEC Command	4-44
The -REMOTE Commands	4-44
Reading From and Writing to an External File	4-45
The .EVAL Operator	4-46

Creating Expressions .....	4-48
Arithmetic Expressions .....	4-49
Alphanumeric Expressions .....	4-50
Logical Expressions .....	4-52
Compound Expressions .....	4-55
Using Functions .....	4-56
System-supplied Function Examples .....	4-56
System-supplied Function Table .....	4-57
Editing a Value .....	4-65
Decoding a Value .....	4-66
Using Commands Specific to an Operating System .....	4-68
ON TABLE HOLD .....	4-69
ON TABLE PCHOLD .....	4-69
<b>5. Transaction Adapters for IMS/TM .....</b>	<b>5-1</b>
Overview of Transaction Adapters .....	5-2
CALLIMS Adapter for IMS/TM .....	5-2
CALLITOC OTMA Adapter for IMS/TM .....	5-2
Transaction Processing With CALLIMS or CALLITOC .....	5-2
Using CALLIMS and CALLITOC .....	5-4
Transaction Processing With CALLIMS or CALLITOC .....	5-5
How Data Is Returned With CALLIMS and CALLITOC .....	5-5
Executing CALLIMS and CALLITOC .....	5-6
Installation of CALLIMS and CALLITOC .....	5-11
Installing CALLIMS .....	5-12
Installing CALLITOC .....	5-14
Storing Multiple Messages Into a Server File for Later Queries .....	5-15
ETPCIMS LU6.2 Adapter for IMS/TM .....	5-16
Using ETPCIMS LU6.2 .....	5-16
Transaction Processing With ETPCIMS LU6.2 .....	5-16
How Data Is Returned With ETPCIMS .....	5-18
Environment Parameters .....	5-18
Executing ETPCIMS .....	5-20
Installing ETPCIMS .....	5-21
Server Settings and Environment Variables .....	5-21
Solaris Platform .....	5-22
HP Platform .....	5-22
AIX Platform .....	5-22

<b>6. Platform-specific Commands</b> .....	<b>6-1</b>
DYNAM Command (MVS) .....	6-2
Use of Data Sets .....	6-4
DYNAM Allocation User Exit .....	6-4
The ALLOCATE Subcommand .....	6-5
The CONCAT Subcommand .....	6-14
The FREE Subcommand .....	6-15
The CLOSE Subcommand .....	6-16
The COPY Subcommand .....	6-17
The COPYDD Subcommand .....	6-19
The DELETE Subcommand .....	6-20
The RENAME Subcommand .....	6-21
The SUBMIT Subcommand .....	6-22
The COMPRESS Subcommand .....	6-23
Comparison of TSO Commands, JCL, and DYNAM .....	6-23
FILEDEF Command Under VM .....	6-25
FILEDEF Command Under UNIX, Windows, OS/400, OS/390 and z/OS, and OpenVMS .....	6-25
Other FILEDEF Features .....	6-27
OFFLINE Printing .....	6-27
<b>A. Dialogue Manager Quick Reference</b> .....	<b>A-1</b>
Dialogue Manager Commands .....	A-2
<b>B. GENCPGM Usage</b> .....	<b>B-1</b>
Using GENCPGM .....	B-2
<b>Index</b> .....	<b>I-1</b>

---

---

## CHAPTER 1

# Introduction

**Topics:**

- What Is a Stored Procedure?
- Why Use Stored Procedures?
- Calling Stored Procedures
- Setting the Execution Order

These topics provide an overview of stored procedures. They provide examples of stored procedures, as well as detailed descriptions of each function. They also describe how client applications execute stored procedures, and how to set this execution order.

## What Is a Stored Procedure?

---

A stored procedure is a program or procedure that resides on the execution path of a server. The procedure is generally called by a client application but can also be called (nested) by another explicitly requested procedure. It is executed on the server on which it resides.

A stored procedure is one of the following:

- A compiled program, written in a language such as C or COBOL, which is located and called on a server or gateway process.
- A file of executable commands written in the server's Dialogue Manager (DM) language.
- A transaction running under the control of a transaction-processing monitor such as CICS or IMS/TM.

**Note:** The ability to use the above methods is limited to what an underlying product supports (in the DBMS) and varies by platform. Any limitations will be noted in the documentation.



## Why Use Stored Procedures?

---

Stored procedures enable you to:

- Embed procedural logic in your server applications. The logic may be modular, eliminating the need to recreate it for each application.
- Update non-relational database management systems.

## Calling Stored Procedures

---

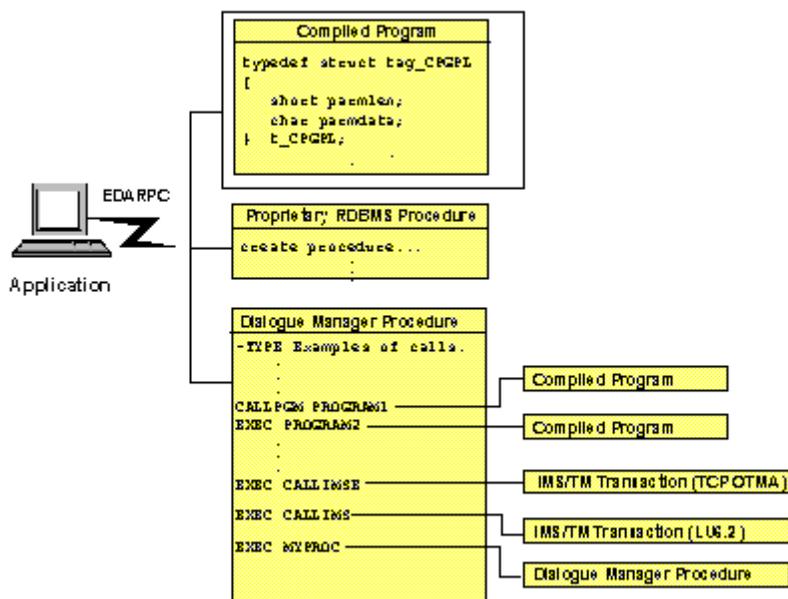
A client application typically executes a stored procedure using the API function call EDARPC. The EDARPC function directly calls one of the following:

- A compiled program.
- A Dialogue Manager procedure.

A Dialogue Manager procedure calls:

- A compiled program, using the command CALLPGM or EXEC.
- A proprietary RDBMS procedure, using SQL Passthru mode (where supported).
- An IMS/TM transaction, using the CALLIMS or CALLIMSC procedure.
- Another Dialogue Manager procedure, using the command EXEC.

The following figure illustrates calls to stored procedures made from EDARPC and Dialogue Manager.



## Setting the Execution Order

This section describes the order in which the server searches for and runs stored procedures, called the execution order. Understanding the execution order enables you to set it appropriately.

The server has a default search order. Change the execution order by:

- Adding the command SET EXORDER in the global or user profile. The server enforces the execution order specified in the profile that was last run. For details on the global and user profiles and how to customize each, see the *Server Administration* manual.
- Running a Dialogue Manager procedure on the server that contains the command SET EXORDER. This command sets the execution order appropriately for subsequent calls to stored procedures. If the Dialogue Manager procedure was the last run (that is, after the global or user profile), the execution order it specifies takes precedence over the execution order in the profile.

Make sure the execution order in effect includes a search of the Procedure Library if you set the execution order in a Dialogue Manager procedure before you run the procedure. Libraries are described in the next section.

Execution order may be reset as needed. When you disconnect or reconnect, the global profile setting for the execution order (or user profile setting, if applicable) will take effect. However, in a pooled environment, the last setting of the prior user is maintained (unless an agent refresh has occurred during the interim).

## Stored Procedure Libraries

Stored procedures must reside in certain libraries in order to be located by the server.

Type of Stored Procedure	Library
<b>Dialogue Manager</b>	Server Procedure Library. The external names, EDARPC (MVS) or EDAPATH (all other platforms), are used to locate Dialogue Manager procedures. Additionally, the methods supported by the APP feature may be used as an alternative method of locating and managing application code. This process is platform dependent. See the <i>Server Administration</i> manual for details.
<b>Compiled Program</b>	Server Program Library. The external name IBICPG or physical placement in the user directory of EDACONF is used to locate compiled programs. This process is platform dependent. A common past practice was to place compiled procedures in the installation home bin directory/library, since it was always searched by default. This practice is not recommended.
<b>IMS/TM Transaction</b>	Server Program Library. Underlying routines are part of the server installation home bin directory; no library configuration is required.

 **Note:** External name is a generic name for variables that are set at the operating system level. The various operating systems that support this feature have different names and methods (syntax) for setting and reviewing these variables. Some of the more commonly used terms for these external names and values are environment variables, registry variables, globals, symbols, defines, assignments, and ddnames. See the Configuration and Operations manual for your platform for specific aspects of working with external names.

## Valid EXORDER Settings

The following table describes valid settings for the execution order.

The recommended setting is either:

- `SET EXORDER=FEX/PGM`

or

- `SET EXORDER=PGM/FEX`

Either setting ensures that both the Procedure Library and Program Library are searched, providing you with the most flexibility.

Setting	Library Searched	Comments
<code>SET EXORDER=FEX</code>	Procedure Library only.	This is the default.
<code>SET EXORDER=PGM</code>	Program Library only.	
<code>SET EXORDER=FEX/PGM</code>	Procedure Library first, followed by Program Library.	If the call is to a program, the name of the program cannot be the same as the name of a Dialogue Manager procedure on the server's search path. If it is, the server will find the Dialogue Manager procedure in the Procedure Library and execute it, rather than the executing the program.
<code>SET EXORDER=PGM/FEX</code>	Program Library first, followed by Procedure Library.	If the call is to a Dialogue Manager procedure, the name of the procedure cannot be the same as the name of a program on the server's search path. If it is, the server will find the program in the Program Library and execute it, rather than executing the Dialogue Manager procedure.

### **Example** Querying the Execution Order

Issue the following Dialogue Manager command to query the current setting of EXORDER:

```
? EXORDER
```

## **Execution Order of Stored Procedures From Dialogue Manager**

This section describes the execution order used by the server to locate and run stored procedures called from Dialogue Manager.

### **Using CALLPGM**

If you use CALLPGM in a Dialogue Manager procedure to call a stored procedure, the server recognizes that the stored procedure is a compiled program, and uses IBICPG or the existence of EDACONF in the user directory to locate the procedure.

### **Using EXEC**

If you use EXEC in a Dialogue Manager procedure to call a stored procedure, the server adheres to the setting of the execution order specified by SET EXORDER, since EXEC could be calling either a compiled program or a Dialogue Manager procedure.

### **Using CALLIMS or CALLITOC**

The CALLIMS and CALLITOC programs contain Dialogue Manager procedures (called CALLIMS and CALLIMSC) to front-end the underlying stored procedures. If you use the CALLIMS or CALLITOC programs directly from a Dialogue Manager procedure, the server recognizes that you are calling a compiled program, and IBICPG does not need to be set.

---

---

## CHAPTER 2

# Calling a Program as a Stored Procedure

### Topics:

- Calling a Compiled Program
- Calling a Program With EDARPC
- Calling a Program With CALLPGM
- Passing Parameters
- Program Communication

These topics describe two ways to call a compiled program that is a stored procedure found and executed by the server. Call a program directly, using the EDARPC function call, or indirectly, using either the CALLPGM command or the EXEC command. Also discussed are ways to pass parameters to programs and Dialogue Manager procedures.

## Calling a Compiled Program

---

The API function call EDARPC enables a client application to call a compiled program stored on the server.

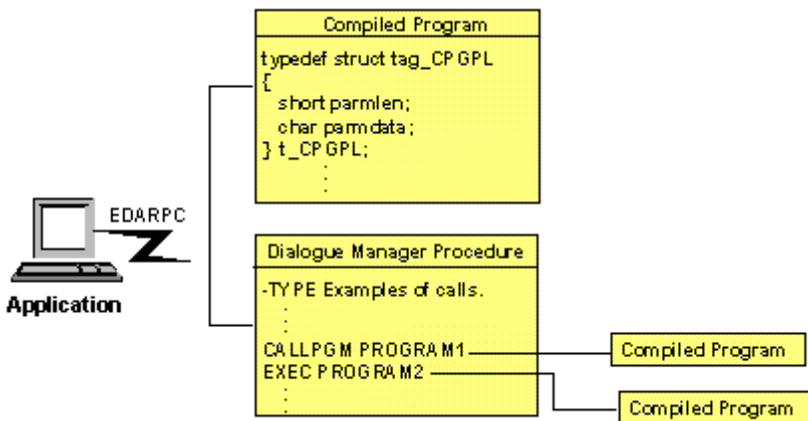
The program is called in two ways:

- Directly, with EDARPC specifying the program name.
- Indirectly, with EDARPC specifying the name of a Dialogue Manager procedure that, when executed, calls the program using one of the following commands:
  - CALLPGM
  - EXEC

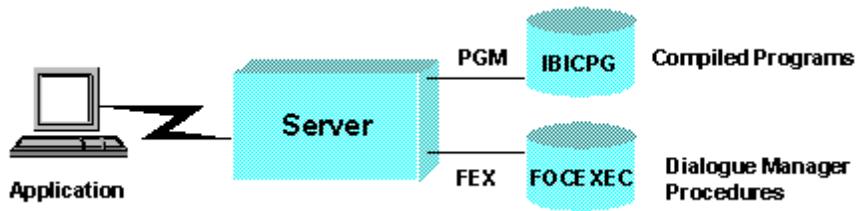
The command EXEC functions the same way as CALLPGM, except for the difference in execution order requirements as described in Chapter 1, *Introduction*. For simplicity, this chapter refers only to CALLPGM when both CALLPGM and EXEC apply.

Also, the term *program* is used to refer to a compiled program.

The following figure illustrates calls to programs made from EDARPC and Dialogue Manager.

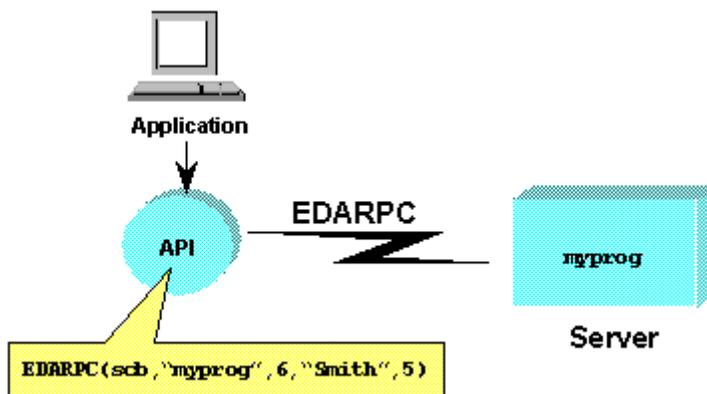


The following figure illustrates the libraries in which compiled programs and Dialogue Manager procedures reside. See Chapter 1, *Introduction*, for details on stored procedure libraries and stored procedure execution order.



## Calling a Program With EDARPC

The following figure illustrates a direct call to a program that could be either a compiled program or a Dialogue Manager EXEC. The program is called using the function call EDARPC. In the figure, the program is named myprog.



For details on the syntax and use of EDARPC, see the *API Reference* manual.

For specific requirements that may apply to your platform, see the *Server Administration* manual.

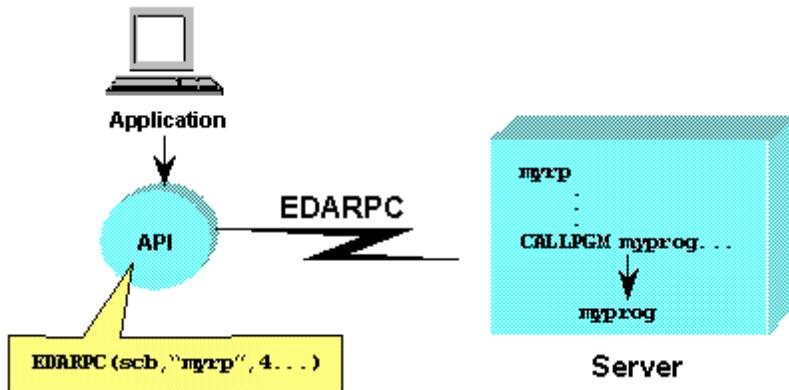
## Calling a Program With CALLPGM

Application developers use Dialogue Manager for program control and flexibility. Additionally, CALLPGM is used where needed.

CALLPGM also provides application developers with:

- A consistent call interface to any program on a server.
- A simple way to create full answer sets and messages.

The following figure illustrates the use of CALLPGM to call a program within a Dialogue Manager procedure.



The steps in this process are:

1. The client application issues the API function call EDARPC, specifying the name of a Dialogue Manager procedure (myrp).
2. The Dialogue Manager procedure is located and executed by the server. The command CALLPGM myprog within the procedure is run.
3. The program myprog executes and terminates.

**Note:**



CALLPGM may call the program several times to allow it to construct and return complete table data, a complete set of messages, or both. See *Passing Parameters* on page 2-7 for more information.

4. CALLPGM performs one or both of the following actions, which are transparent to the Dialogue Manager procedure:

- Passes a message or messages to the client application for processing. The client application issues the API function call EDAACCEPT to access the message(s).

**Note:**



The program must return messages to the client application before any table data (that is, description of an answer set and the rows of data), or at the end of any table data.

- Passes table data to the client application for processing. Table data consists of two components:

A CREATE TABLE that tells the server the format of the returned data. For more information on describing data, see Chapter 3, *Writing a 3GL Compiled Stored Procedure Program*.

Rows of data, which the client application retrieves using the API function call EDAFETCH.

The Dialogue Manager procedure itself does not need to create an answer set or message.

### **Syntax** How to Call a Program From Dialogue Manager

```
CALLPGM progrname[,parmval1][,...]
END
```

or

```
EX[EC] progrname[ parmval1][,...]
END
```

where:

*progrname*

Is the name of the program to be run. (If CALLPGM is used, it cannot be another Dialogue Manager procedure.)

*parmval1*

Is an optional positional Dialogue Manager parameter passed to *progrname*. A Dialogue Manager parameter is an alphanumeric value. See *Passing Parameters* on page 2-7 for examples.

The length of a single parameter (for example, *parmval1*) cannot exceed 32,000 characters. The total length of all specified parameters cannot exceed 32,000 characters.

END

Is a required command that terminates CALLPGM or EXEC.

### **Switching Plans in DB2 (MVS Only)**

DB2 requires that all programmed interaction with a database be controlled at the program module level. The program is represented to the database using an object called a *plan*. The installation procedure automatically creates a plan for a server. When the server accesses the RDBMS, it uses the plan name.

When a program executed by CALLPGM contains SQL statements, it may be necessary to switch from the plan named in the installation procedure to the plan required by the program.

### **Syntax** How to Switch Plans in DB2

The syntax is

```
SQL DB2 SET PLAN &progplan
CALLPGM &program...
SQL DB2 SET PLAN ' '
```

where:

*&progplan*

Is the name of the plan required by the program.

*&program*

Is the name of the program to be run.

SET PLAN *&progplan*

Specifies the plan required by the program.

SET PLAN ' '

Resets the plan.

An alternative is to use DB2 3.1 packages. Here, each CALLPGM program has its own package (called by the same name as the program), and all programs are included in the package list for the plan.

For example, assume that your server plan is called EDASQL. You wish to have two stored procedures, called SPG1 and SPG2, that use static SQL to access DB2.

In this case, there are three DB2 database resource modules (DBRMs) created: EDASQL, SPG1, and SPG2. Create three packages, called EDASQL.EDASQL, EDASQL.SPG1, and EDASQL.SPG2, using the command CREATE PACKAGE. Then bind the packages together into a plan using the command BIND PLAN with the package list option. When the server executes, DB2 automatically selects the package with the same name as the program.

For more information on plans, see the applicable DB2 manuals.

### **Example** Processing an Answer Set on the Server

When executing a CALLPGM stored procedure, it is sometimes desirable to retain the answer set on the server. The following example illustrates the method used to retain the answer set on the server:

1. SQL EDA SET SERVER *servername*
2. SQL EDA EX *programname parm1, ...;*
3. TABLE FILE SQLOUT  
PRINT \*  
ON TABLE HOLD AS *filename*  
END
4. TABLE FILE *filename*  
PRINT col2 AS 'COLUMN, 2'  
col3 AS 'COLUMN, 3'  
END

The procedure processes as follows:

1. Identifies the remote server name in which to execute remote requests.
2. Executes the program name on the remote server.
3. Specifies that the temporary information is to be retained on the server in an extract file.
4. Executes a TABLE request to generate an answer set containing column 2 and column 3 in the retained table.

**Note:** • The file specified must be allocated prior to being used. For more information on allocating a file, see Chapter 6, *Platform-specific Commands*.



- The above example is also valid when running CALLPGM locally.

## Passing Parameters

---

The following terminology is used in this section:

- Parameters passed on the EDARPC call by an API program are called API parameters, which specify Dialogue Manager (DM) and CALLPGM program (CPG) parameters (described below).
- Amper variables used in a Dialogue Manager procedure are also called DM variables.
- Parameters in a Dialogue Manager procedure not directly stored in amper variables are called DM parameters (that is, text parameters that get passed in and used).
- Parameters passed to a program called by CALLPGM are called CPG parameters.

## Using CALLPGM

When passing CPG parameters that contain embedded spaces or commas, the parameters must be enclosed in quotes. The following profile setting controls the stripping of quotes from parameters.

### **Syntax** How to Control the Stripping of Quotes From Parameters

```
SQL SPG SET STRIPQUOTE {ON|OFF}
```

where:

**ON**

Causes the quotes to be stripped from the parameters. This value is the default.

**OFF**

Prevents the stripping of the quotes from the parameters.

## Using EDARPC

EDARPC passes positional or keyword API parameters. Positional parameters work with Dialogue Manager procedures or compiled programs. Keyword API parameters only work with Dialogue Manager procedures.

**Note:** Positional and keyword API parameters are mixed if performed as described.



This section contains examples of positional and keyword API parameters passed by EDARPC.

### Example Passing Positional API Parameters

EDARPC passed one or more positional API parameters to a Dialogue Manager procedure or compiled program, which uses each in a variety of ways. Positional API parameters receive the values from the order in the EDARPC calling sequence.

Positional API parameters are passed as a string enclosed in double quotation marks, with the positional values separated by commas as shown in the following example:

```
EDARPC ( scb, " myproc", 6, " myprog, Sales, 20", 15 )
```

where:

*scb*

Is the session control block.

*myproc*

Is the name of a Dialogue Manager procedure or compiled program.

6

Is the length of the string *myproc*.

*myprog*, *Sales*, 20

Is a string (an API parameter) containing the three positional parameters.

15

Is the length of the above string.

For the purpose of this example, assume *myproc* is a Dialogue Manager procedure and the procedure uses the API parameter as DM variables &1, &2, and &3 to, in turn, issue a CALLPGM command as follows:

```
CALLPGM &1, &2, &3  
END
```

When the Dialogue Manager procedure executes, the server substitutes the values for the variables *&1*, *&2*, and *&3*, and the result is:

```
CALLPGM myprog, Sales, 20
END
```

The values Sales and 20 are passed to the underlying compiled program myprog.

### **Example** Passing Keyword API Parameters

EDARPC also passes one or more keyword API parameters to a Dialogue Manager procedure. The value of a keyword API parameter is determined by the name given before the equal sign (=) on the EDARPC function call.

Keyword DM parameters are specified as name=value pairs in the API parameter and are passed as a string enclosed in double quotation marks, with name=value pairs separated by commas. Keyword DM parameters are only used in Dialogue Manager procedures.

```
EDARPC ( scb, "myrp" , 4, "prog=myprog, parm1=Sales, parm2=20" , 32)
```

where:

*scb*

Is the session control block.

*myrp*

Is the name of a Dialogue Manager procedure.

4

Is the length of the string *myrp*.

```
prog=myprog, parm1=Sales, parm2=20
```

Is a string (an API parameter) containing three keyword DM parameter value pairs.

32

Is the length of the above string.

For the purpose of this example, assume that *myrp* is a Dialogue Manager procedure and the procedure puts the keyword DM parameters in the DM variables *&prog*, *&parm1*, and *&parm2*, and then uses each in a CALLPGM command:

```
CALLPGM &prog, &parm1, &parm2
END
```

When values are substituted at run time, the result is the same command as in the previous example:

```
CALLPGM myprog, Sales, 20
END
```

The advantage of keyword parameters is that the order of the parameters is positionally independent. An API program does not need this level of knowledge and Dialogue Manager is used to establish default values. Defaults values do not need to be established as part of the API program.

For more information on API positional and keyword parameters, see the *API Reference* manual.

### **Example** Combining Positional and Keyword API Parameters

EDARPC passes one or more positional API parameters mixed with one or more keyword parameters to a Dialogue Manager procedure. The server substitutes the values for the ampersand variables based on the relative position of the positional keywords to each other.

```
EDARPC ( scb, "myproc", 6, "prog=myprog, 000001, 000002, kparm=keyparm, 000003", 4  
7)
```

where:

*scb*

Is the session control block.

*myproc*

Is the name of a Dialogue Manager procedure.

6

Is the length of the string *myproc*.

*prog=myprog... , 000003*

Is a string (an API parameter) containing three positional and two keyword Dialogue Manager parameters.

47

Is the length of the above string.

For the purpose of this example, assume that *myproc* is a Dialogue Manager procedure and the procedure uses the API positional parameters *&1*, *&2*, and *&3*, and the keyword parameters *&prog* and *&kparm1* to, in turn, issue a CALLPGM command as follows:

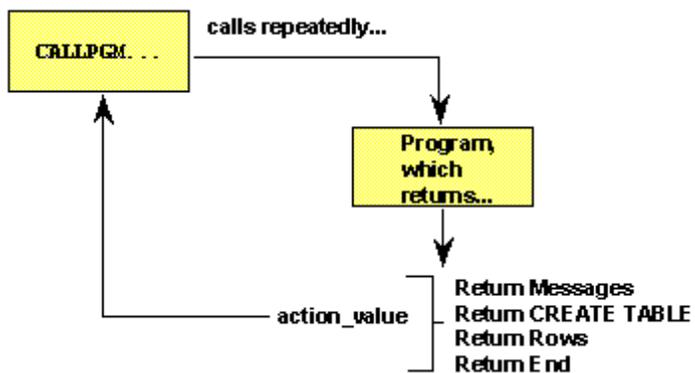
```
CALLPGM &prog, &1, &2, &kparm1, &3;  
END
```

When the Dialogue Manager procedure executes, the server substitutes the values for the variables *&1*, *&2*, *&3*, *&prog*, and *&kparm1*, and the result is:

```
CALLPGM myprog, 000001, 000002, keyparm, 000003;
```

The values 000001, 000002, keyparm, and 000003 are passed to the program *myprog*.





---

---

## CHAPTER 3

# Writing a 3GL Compiled Stored Procedure Program

### Topics:

- Program Requirements
- Setting Up the Control Block
- Storing Program Values
- Error Handling
- Issuing the CREATE TABLE Command

These topics describe the requirements for writing a 3GL compiled program to be called by the EDARPC function call or by the CALLPGM command. They explain how to set up control blocks for communication between the server and the program, and how to store program values so that the program retrieves addresses of allocated data storage. These topics also discuss the CREATE TABLE command, which the program issues in order to describe the answer set that it is returning.

## Program Requirements

---

If you are writing a program to be stored on a server and called as a 3GL stored procedure, you must:

- Write and compile a program as a loadable library.
- Create a control block for communication.
- Retain values used by your program.
- Issue the CREATE TABLE command to describe any answer set before returning it.

For details on calling a compiled program with EDARPC or CALLPGM, see Chapter 2, *Calling a Program as a Stored Procedure*.

**Note:** Loadable library is a generic term. The actual technical name varies by operating system.  Other commonly used terms for these types of file are dll, service program, shared library, and shared image. The script, gencpgm, is provided on UNIX, Windows, OS/400, and OpenVMS to assist in the actual compilation of a program, but any method is allowed provided that it links in the appropriate library and builds the file as a dynamic load library (for example, .so for UNIX, .dll for Windows, service program on OS/400, and shared library on OpenVMS). For more information, see Appendix B, *GENCPGM Usage*.

## Setting Up the Control Block

---

The server uses a control block for communication with a compiled program. The following applies:

- Under MVS, OpenVMS, UNIX, OS/400, or Windows NT, the address of the control block is sent to the program as the first parameter.
- Under CICS, the control block is the COMMAREA.

The following sections provide examples of a control block in C and COBOL.

Values for the fields in the control block are supplied by either the server or the called program. If a field is designated non-modifiable in the sample control block in C, its value is supplied by the server and cannot be changed by the program. This restriction also applies to the corresponding field in the sample control block in COBOL.

### Control Block Fields

Fields used in the sample control blocks in the following sections are described in the table below.

Field	Length (in bytes)	Data Type	Description
<code>input_CB_length</code>	2	Integer	Specifies the length of the <code>input_CB</code> passed by the server, including any passed parameters.  Non-modifiable. The server supplies the value; the called program cannot modify it.
<code>reserved</code>	2	Integer	Non-modifiable. Reserved for server use.
<code>flag_value</code>	4	Integer	Specifies whether this is the first time the server has called the program for this client application:  <code>1</code> First time.  <code>0 0</code> All other times (unless an error occurs; see the following error codes).  Non-modifiable. The server supplies the value; the called program cannot modify it.  If the server encounters a problem, it sets the <code>flag_value</code> to one of the following error codes, and calls the program again. The called program should check for these errors; if it receives one, it should clean up and log the <code>flag_value</code> .

Field	Length (in bytes)	Data Type	Description
<p><code>flag_value</code> (continued)</p>			<p>The server supplies the value; the called program cannot modify it.</p> <p>100 Program name invalid.</p> <p>101 Cannot get main parameter buffer.</p> <p>200 CS/2 error condition (a communications subsystem error).</p> <p>300 Cannot get memory.</p> <p>302 Cannot load program.</p> <p>305 Bad value from user program.</p> <p>306 Remote program abend.</p> <p>307 Client abend.</p> <p>308 CVT not found.</p> <p>309 Cxinit call failed (an internal API error).</p> <p>310 Cxdefault call error (an internal API error).</p> <p>311 Cxsetuser call error (an internal API error).</p> <p>312 Cxset call failed (an internal API error).</p> <p>313 Invalid blocking factor. An <code>action_value</code> of +14 was specified, but the blocking factor was <math>\leq 0</math>.</p>

Field	Length (in bytes)	Data Type	Description
<p><code>flag_value</code> (continued)</p>			<p>400 CS/3 error condition (a communications subsystem error).</p> <p>500 Cannot get memory.</p> <p>501 Unexpected message received.</p> <p>502 Cannot load program.</p> <p>503 Premature disconnect.</p> <p>600 NTK (tokenizer) error in a CREATE TABLE (an internal component error).</p> <p>602 Main buffer failure in a CREATE TABLE.</p> <p>603 Left parenthesis missing in a CREATE TABLE.</p> <p>604 Fieldname missing in a CREATE TABLE.</p> <p>605 Data type missing in a CREATE TABLE.</p> <p>606 Unidentified data type in a CREATE TABLE.</p> <p>607 Too many digits in column length in a CREATE TABLE.</p> <p>608 Right parenthesis missing in a CREATE TABLE.</p> <p>700 NTKOP call failed (an internal API error).</p> <p>701 More than 254 fields in a CREATE TABLE.</p> <p>702 Invalid Master File.</p>

Field	Length (in bytes)	Data Type	Description
<p><code>action_value</code></p> <p>Initial value on first call: 4</p>	4	Integer	<p>Specifies the type of response from the called program:</p> <p>1 It is returning a CREATE TABLE. See <i>Issuing the CREATE TABLE Command</i> on page 3-20 for information.</p> <p>2, 3 It is returning a row in mixed format.</p> <p>4 It is returning a message.</p> <p>9 It has run and terminated.</p> <p>14 It is returning a block of rows in mixed format. The blocking factor must be supplied in the <code>message_length</code> field.</p> <p>The called program supplies the value.</p>
<p><code>answer_area</code></p> <p>Initial value on first call: 0</p>	4	Pointer (address)	<p>The address of the data returned by the called program.</p> <p>The called program supplies the value when <code>action_value</code> is 1, 2, or 3.</p> <p>See <i>Storing Program Values</i> on page 3-9 for more information on the use of this field.</p>
<p><code>answer_length</code></p> <p>Initial value on first call: 0</p>	4	Integer	<p>The length of the data returned by the called program.</p> <p>The called program supplies the value when <code>action_value</code> is 1, 2, or 3.</p>
<p><code>message_area</code></p> <p>Initial value on first call: 0</p>	4	Pointer (address)	<p>The address of a message returned by the called program.</p> <p>The called program supplies the value when <code>action_value</code> is 4.</p> <p>See <i>Storing Program Values</i> on page 3-9 for more information on the use of this field.</p>

Field	Length (in bytes)	Data Type	Description
<p><code>message_length</code></p> <p>Initial value on first call: 0</p>	4	Integer	<p>The length of the message returned by the called program.</p> <p>The number of answer set rows returned as a block and only used when <code>action_value</code> is +14.</p> <p>The called program supplies the value when <code>action_value</code> is 4.</p>
<p><code>parmlen</code></p>	4	Integer	<p>The length of a parameter passed to the called program.</p> <p>The server supplies the value.</p> <p>This field is paired with <code>parmdata</code> (see next item). Twelve pairs are permitted per program call.</p>
<p><code>parmdata</code></p>	Variable	Any type	<p>The value of the parameter passed to the program.</p> <p>The server supplies the value (from EDARPC or a Dialogue Manager procedure).</p> <p>This field is paired with <code>parmlen</code>. Twelve pairs are permitted per program call.</p>

**Example**    **Setting Up a Control Block in C**

```

typedef struct parm_struct
{
    long parmlen;                /*                               */
    char parmdata[n];           /*                               */
};

typedef struct input_CB {       /* CALLPGM User Block           */
    const short input_CB_length; /* Non-modifiable              */
    const short reserved;       /* Non-modifiable              */
    const long flag_value;      /* Non-modifiable              */
#define CPGUB_flag_frst 1      /* First-time value            */
/* for flag                    */
#define CPGUB_flag_nfst 0      /* Not first-time              */
/* value for flag              */
    long action_value;         /* Action to be taken          */
/* on callback                  */
#define CPGUB_action_CT 1     /* Action Create Table         */
#define CPGUB_action_DA 2     /* Action Data (Mixed)         */
#define CPGUB_action_CD 3     /* Action Character Data       */
#define CPGUB_action_MS 4     /* Action Message              */
#define CPGUB_action_EX 9     /* Action Exit                  */
#define CPGUB_action_DAB 14   /* Action Data Block of Tuples */
/* NOTE: Any undefined action  */
/* is treated as CPGUB_action_EX, */
/* that is, Exit                */
    int * answer_area;         /* Modifiable                  */
    long answer_length;        /* Modifiable                  */
    int * message_area;        /* Modifiable                  */
    long message_length;       /* Modifiable                  */
    struct parm_struct parml;   /* Parameter 1                  */
    .                           /* . NOTE: 0 to 12 parameters  */
    .                           /* . may be input to the      */
    .                           /* . program                  */
    struct parm_struct parm[n]; /* Parameter n                  */
};

```

**Example**    **Setting Up a Control Block in COBOL**

Place the control block in the LINKAGE SECTION and address it by the first passed parameter.

```

05  FIXED-LENGTH-PART .
    10  INPUT-CB-LENGTH          PIC S9(4)  COMP-4 .
    10  FILLER                   PIC S9(4)  COMP-4 .
    10  FLAG-VALUE               PIC S9(8)  COMP-4 .
        88  FLAG-FIRST-TIME      VALUE +1 .
        88  FLAG-NOT-FIRST-TIME  VALUE 0 .
        88  FLAG-ERROR           VALUE +2 THRU +1999 .
    10  ACTION-VALUE            PIC S9(8)  COMP-4 .
        88  CREATE-TABLE-ACTION  VALUE +1 .
        88  RETURNING-MIXED-DATA VALUE +2 .
        88  RETURNING-CHAR-DATA  VALUE +3 .
        88  RETURNING-MESSAGE    VALUE +4 .
        88  PROGRAM-FINISHED     VALUE +9 .
        88  RETURNING-MIXED-BLOCK VALUE +14 .
    10  ANSWER-AREA             POINTER .
    10  ANSWER-LENGTH           PIC S9(8)  COMP-4 .
    10  MESSAGE-AREA            POINTER .
    10  MESSAGE-LENGTH          PIC S9(8)  COMP-4 .
05  PARAMETERS-PART .
    10  PARM1LEN                 PIC S9(8)  COMP-4 .
    10  PARM1DATA                PIC X(n) .
        .
        .
        .
    10  PARMLEN                  PIC S9(8)  COMP-4 .
    10  PARMMDATA                PIC X(n) .

```

## Storing Program Values

---

When running in a multi-user environment, programs called by CALLPGM may be multi-threaded. If so, data returned to the server must be returned in dynamically allocated storage, and the program must know how to retrieve the address of that storage. This is illustrated in the sample code for MVS and CICS in the following subsections. The sample code for OpenVMS assumes a single-user environment.

Programs called by CALLPGM typically return the following data to the server:

- Messages (up to 80 bytes).

Messages returned by the program are pointed to by the control block field `message_area`. The length is given in the field `message_length`.

- Answer set descriptions, that is, CREATE TABLEs (up to 1,000 bytes).

Answer set descriptions or rows (see below) returned by the program are pointed to by the control block field `answer_area`. The length is given in the field `answer_length`.

- Rows or tuples (up to 32,000 bytes).

A program returns data by placing it in dynamically acquired storage.

It may also be necessary for subsequent invocations of a program to retrieve previously stored values, which also requires the use of dynamically acquired storage. It is the program's responsibility to free such storage at its last invocation.

Dynamic storage is acquired using:

- malloc in C.
- EXEC CICS GETMAIN in COBOL or C under CICS.
- 'GETCOR' in COBOL under VTAM.
- "LIB\$GET\_VM" in COBOL under OpenVMS.

By placing the address of the storage in the control block fields message\_area and answer\_area, the server returns the values to you on the next call, and then re-address the variables. Always point the message\_area and answer\_area to valid data when control is returned to the server.

The examples in the following sections show how values are saved across invocations of a program. The first time a program is called, it allocates dynamic storage for the values to be saved. Each subsequent time the program is called, the address of the dynamic storage is retrieved using the message\_area or answer\_area.

Sample programs are supplied in your software installation location as described below.

Type of Program	Supplied As
C	MVS CPGC370 in <a href="#">qualif.EDALIB.DATA</a> All other platforms <a href="#">cpt.c</a>
CICS COBOL	CPGCICS in <a href="#">qualif.EDACICS.DATA</a>
VTAM COBOL	CPGVTAM in <a href="#">qualif.EDACTL.DATA</a>
OpenVMS COBOL	CPGVMS in <a href="#">EDAHOME:CPGVMS.COB, SPDEMO1.DAT</a>

### **Example** Storing Program Values in C

The following sample C code illustrates the allocation of dynamic storage on the first call, and addressability to program variables on subsequent calls.

```

typedef struct message_buffer
{ char    message[80] ;
  } message_buffer;

typedef struct answer_tuple
{ char    customer_name[40]    ;
  char    customer_address[90] ;
  char    balance_due[20]     ;
  char    comments[300]       ;
  } answer_tuple;

typedef struct answer_buffer
{ int          *program_variable_buffer_ptr ;
  struct answer_tuple  answer_set_tuple      ;
  } answer_buffer;

typedef struct program_variable_buffer
{ long    number_of_rows    ;
  long    last_record      ;
  short   reserved         ;
  short   close_pending_flag ;
  } program_variable_buffer;
      .
      .
      .
      .
      .
/* On the first call, allocate message, answer, and program variable
*/
/* buffers and anchor them in the input control block. The program's
*/
/* local variables are anchored by saving a pointer immediately
*/
/* preceding the answer_area. The pointer saved in the answer_area is
*/
/* actually 4 bytes into the answer_buffer, providing the correct
*/
/* interface to the server for processing answer set requests, */
/* while still anchoring the program's local variables by "hiding" the
*/
/* pointer in the memory immediately preceding the answer_area. By
*/
/* placing the pointer before the answer_set_tuple, it is not seen
*/
/* by the server. */
/*
*/

```

## Storing Program Values

```
/* Check for first call of this program.
*/
if ( flag_value = CPGUB_flag_first )

{ /* Allocate answer_buffer.
*/
  answer_buffer_ptr = ( answer_buffer * )
                    malloc(sizeof(answer_buffer), 1);

  answer_area = ( int * ) ( ((long) (answer_buffer_ptr)) + 4 );
  answer_length = sizeof(answer_buffer) - 4;

  /* Allocate buffer for program variables.
*/
  program_variable_buffer_ptr = ( int * )
                                malloc(sizeof(program_variable_buffer), 1);

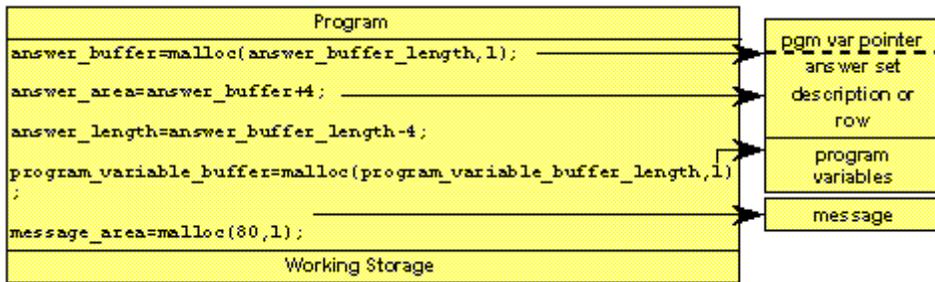
  /* Allocate buffer for messages.
*/
  message_area = ( int * ) malloc(sizeof(message_buffer),1);

  message_length = sizeof(message_buffer);
}
/* On subsequent calls, locate addressability to the program's local
*/
/* variables via the pointer saved immediately before the answer_area
*/
else
{ answer_buffer_ptr = ( answer_buffer * ) (((long) (answer_area)) - 4);
}
.
.
.
```

On the first call, the sample code allocates dynamic storage for:

- Answer set descriptions or rows returned by the program (pointed to by the control block field `answer_area`).
- Program variables to be saved across invocations of the program.
- Messages returned by the program (pointed to by the control block field `message_area`).

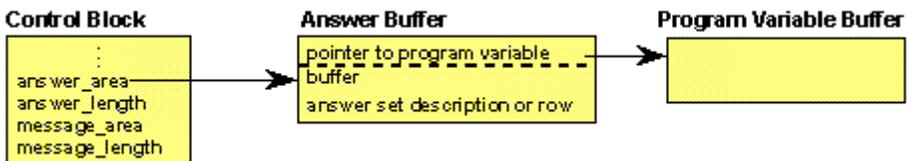
The pointer to the program variable buffer is saved at a fixed location (a known offset), in the first  $n$  bytes of the buffer, for the answer set description or row (called the answer buffer). This is illustrated in the figure below.



For example, on MVS, you might allocate an answer buffer of 1,004 bytes with 1,000 bytes used to store the largest answer set description and the 4 extra bytes used to store the pointer to the program variable buffer.

As shown in the following figure, the pointer stored in the control block's answer\_area points to the answer buffer, excluding the 4 bytes used to store the pointer to the program variable buffer. That is, the pointer is directed toward the beginning of an answer set description or row. (The message\_area could also be used to store the pointer to the program variable buffer, but for the purpose of illustration, the answer\_area was chosen.)

The length of bytes to be stored in the control block's answer\_length would be 1,004 minus 4, or a value of 1,000, to reflect the value of the largest answer set description or row.



To determine the address of the program variable buffer on subsequent calls, the program would subtract the size of the pointer to the program variable buffer (4 bytes on most machines) from the answer\_area in the control block.

When freeing memory on exit, the program determines the size of the answer buffer by adding the answer\_length to the size of the pointer to the program variable buffer.

When using this technique, it is important to keep the answer\_area in the control block consistent with the definition in the interface. Always point the answer\_area and message\_area to valid data when control is returned to the server. Program variables are kept in any allocated memory buffer using this technique.

The program must free all memory allocated during execution before returning an action\_value of 9 (exit) to the server. This requirement applies to the memory for program variables, messages, answer set descriptions, and rows. If all memory is not freed at program exit, server failure may result at a later time.

### **Example Storing Program Values in CICS COBOL**

In COBOL, one way to save program variables across invocations of a program is to allocate one block of storage big enough to hold:

- Any returned messages (up to 80 bytes).
- Answer set descriptions, that is, CREATE TABLEs (up to 1,000 bytes).
- Rows or tuples (up to 32,000 bytes).
- Program variables.

Dynamic storage is acquired using EXEC CICS GETMAIN in COBOL or C under CICS.

The following sample COBOL code describes a MESSAGEAREA. It provides the field MESSAGE-OUT for messages, answer set descriptions (CREATE TABLEs), and rows. It provides the fields NUM-ROWS, LAST-REC, and CLOSE-PENDING-FLAG for program values to be retrieved in subsequent invocations.

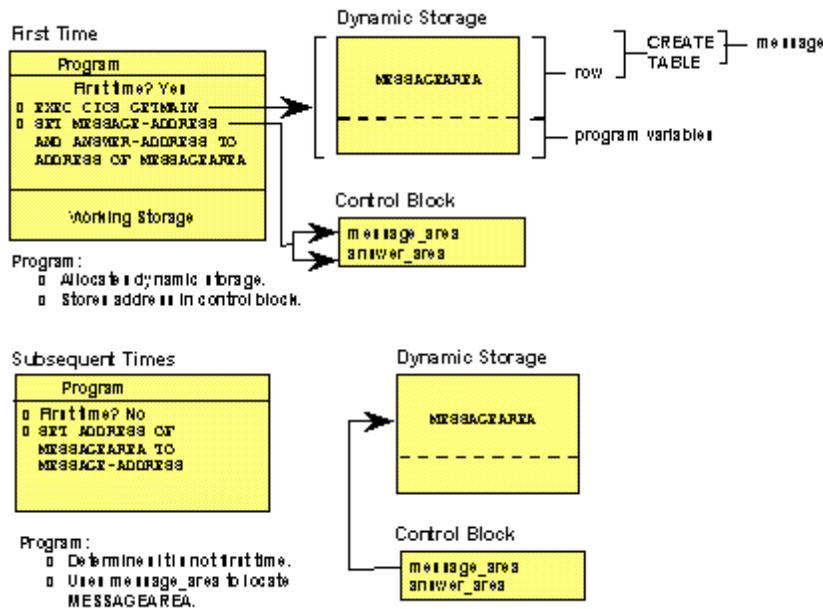
```
01 MESSAGEAREA.
   05 MESSAGE-OUT          PIC X(1000) .
   05 NUM-ROWS             PIC S9(8)  COMP-4 .
   05 LAST-REC             PIC S9(8)  COMP-4 .
   05 CLOSE-PENDING-FLAG  PIC X.
       88 CLOSE-PENDING    VALUE "1" .
       88 CLOSE-NOT-PENDING VALUE "0" .
   05 FILLER               PIC X(15) .
```

The code to store values is:

```
IF FLAG-FIRST-TIME
  MOVE LENGTH OF MESSAGEAREA TO MESSAGE-LENGTH
***** GETMAIN, SET LENGTH, ADDRESSES
  EXEC CICS GETMAIN SET (ADDRESS OF MESSAGEAREA)
          FLLENGTH (MESSAGE-LENGTH)
          INITIMG (INITVALUE)
  END-EXEC
  SET MESSAGE-ADDRESS TO ADDRESS OF MESSAGEAREA
  SET ANSWER-ADDRESS TO ADDRESS OF MESSAGEAREA
ELSE
***** IF NOT THE FIRST TIME, RETRIEVE THE GETMAIN ADDRESS
***** FROM EITHER COMMAREA ADDRESS, AND SET THE ADDRESS
***** OF THE GETMAIN AREA SO IT IS ADDRESSABLE IN COBOL.
  SET ADDRESS OF MESSAGEAREA TO MESSAGE-ADDRESS.
```

The previous code fragment is executed each time the program is invoked. The first time, the program uses EXEC CICS GETMAIN to allocate the storage to the length of the MESSAGEAREA. On each subsequent execution, it gets the address of the MESSAGEAREA from the field MESSAGE-ADDRESS.

The following figure illustrates the program logic in the code fragment. In the figure, the field MESSAGE-ADDRESS in the code is represented as message\_area in the control block.



In this example, the program allocates a buffer (MESSAGEAREA) of 1,000 bytes (for the largest message, answer set description, or row to be returned), plus 24 bytes for the program variables.

In the control block:

- The message\_area and answer\_area are set to the address of the beginning of the buffer.
- The message\_length reflects the size of the messages returned to the client application.
- The answer\_length reflects the size of the answer set descriptions or rows returned to the client application.

To address program variables stored between invocations in this way, use

[SET ADDRESS OF MESSAGEAREA TO MESSAGE-ADDRESS](#)

as shown in the preceding sample code. This code enables the program to refer to the variables NUM-ROWS, LAST-REC, and CLOSE-PENDING-FLAG.

To free storage allocated this way, use:

```
EXEC CICS FREEMAIN (MESSAGEAREA) END-EXEC
```

CICS frees the correct length.

Below is output from a sample session that runs CPGCICS using RDAAPP, a test program supplied on your distribution media.

```
<<< RDAAPP : Initializing API SQL, Version x >>>
<<< Initialization Successful >>>
Trace level ?

Enter User Name :

Enter Password :

Enter Server name (Hit return for 'CICS ') :

<<< Successfully connected to server >>>
Enter (S/P <sql stmt;> / X <RPC> <parms> / D <tbl> / E <prep id> / C/R /
Q) :
x cpgcics 1
Please Wait.
000100
S. D. BORMAN
SURREY, ENGLAND
3215677826
11 81
$0100.11
*****
<<< 1 record(s) processed. >>>
Enter (S/P <sql stmt;> / X <RPC> <parms> / D <tbl> / E <prep id> / C/R /
Q) :
***
```

### **Example** Storing Program Values in VTAM COBOL

To allocate dynamic storage in VTAM COBOL, use the 'GETCOR' function, supplied on your distribution media in the module CPGUSRO.

Specify the following three parameters on the function call:

- The address of the length of the storage to be allocated.
- The address of the allocated memory to be returned.
- The address of an area in which to place the return code.

The following is the code for allocating dynamic storage:

```

01  COR-DATA.
    05  MESSAGEAREA-LENGTH          PIC S9(8) COMP-4.
    05  MESSAGEAREA-ADDRESS        POINTER.
    05  COR-RESP                   PIC S9(8) COMP-4.
        .
        .
        .
MOVE LENGTH OF MESSAGEAREA TO MESSAGEAREA-LENGTH
CALL 'GETCOR' USING
BY REFERENCE MESSAGEAREA-LENGTH,
  BY REFERENCE MESSAGEAREA-ADDRESS,
  BY REFERENCE COR-RESP

```

To free dynamic storage on program exit, use the 'FRECOR' function, also supplied on your distribution media.

Specify the following three parameters on the function call:

- The address of the allocated memory to be freed.
- The address of the length of the storage to be freed.
- The address of the area that held the return code.

The following is the code for freeing dynamic storage:

```

CALL 'FRECOR' USING
  BY CONTENT LENGTH OF MESSAGEAREA,
  BY REFERENCE MESSAGEAREA,
  BY REFERENCE COR-RESP

```

**Note:** Use the COR-RESP return code, not the COBOL RETURN-CODE, as the latter has an arbitrary value.



To link edit the sample program (supplied as CPGVTAM on your distribution media), use the statements below:

```

INCLUDE EDALIB(CPGUSRO)
  INCLUDE OBJECT
  MODE AMODE(31), RMODE(ANY)
  ENTRY CPGVTAM
  NAME CPGVTAM(R)

```

CPGUSRO is a non-executable module that provides dynamic linkage to 'GETCOR' and 'FRECOR'.

## OpenVMS COBOL

The sample OpenVMS COBOL program, CPGVMS.COB, included on the distribution media, assumes a single-user environment and does not store program values in dynamic storage. CPGVMS.COB accepts up to 5 parameters (all employee IDs), and the salary field for each employee ID passed (by increments of 2000). The data resides in SPDEMO1.DAT, an RMS flat file.

### **Example** Linking Program Variables to the Control Block

The following code fragment illustrates how to link program variables to the answer and message pointers, defined in the control block in *Control Block Fields* on page 3-2.

```
WORKING-STORAGE SECTION.  
01 MESSAGE-BUFFER          PIC X(100) VALUE SPACES.  
01 ANSWER-BUFFER          PIC X(100) VALUE SPACES.  
.  
.  
.  
SET ANSWER-ADDRESS TO REFERENCE OF ANSWER-BUFFER  
SET MESSAGE-ADDRESS TO REFERENCE OF MESSAGE-BUFFER
```

### **Example** Checking for First-time Execution

This code checks for the initial execution of the program so that it initializes program variables on the first call:

```
PROCEDURE DIVISION USING CPGUB.  
A010-BEGIN.  
    IF FLAG-FIRST-TIME  
        PERFORM A020-INIT-DATA  
    ELSE  
        IF PARM-COUNT < 5 AND PARM-REMAIN > ZERO PERFORM A030-READ-DATA.  
    EXIT PROGRAM.
```

### **Example** Allocating and Freeing Dynamic Storage

The following code illustrates how to allocate and free dynamic storage used for storing program values:

```

01 NUMBER-OF-BYTES PIC S9(9) COMP.
01 BASE-ADDRESS    PIC S9(9) COMP.
01 RET-STATUS      PIC S9(9) COMP.
.
.
.
A080_ALLOC_STORAGE.
  MOVE +1000 TO NUMBER-OF-BYTES.
  CALL "LIB$GET_VM"
      USING BY REFERENCE NUMBER-OF-BYTES, BASE-ADDRESS
      GIVING RET-STATUS.
.
.
.
A090_FREE_STORAGE.
  MOVE +1000 TO NUMBER-OF-BYTES.
  CALL "LIB$FREE_VM"
      USING BY REFERENCE NUMBER-OF-BYTES, BASE-ADDRESS
      GIVING RET-STATUS.

```

## Error Handling

---

When the server encounters an error during the execution of a program, it calls the program again, indicating the error condition in the control block field `flag_value`. The program then does one of the following, indicating its response in the `action_value` field:

- Free any memory allocated during program execution, and exit, issuing an `action_value` of 9 (exit). The program must allocate and free its own dynamic storage. Make sure that the program frees any allocated resources (especially memory) before issuing an `action_value` of 9. Not freeing memory may cause the server to fail at a later point in time.
- Return a message to the server to explain the error, issuing an `action_value` of 4. The server then attempts to return the message to the client application and call the program again, which must free its resources and end, as described above.

Messages are retrieved by the client application before the processing of an answer set, or after the completion of answer set processing.

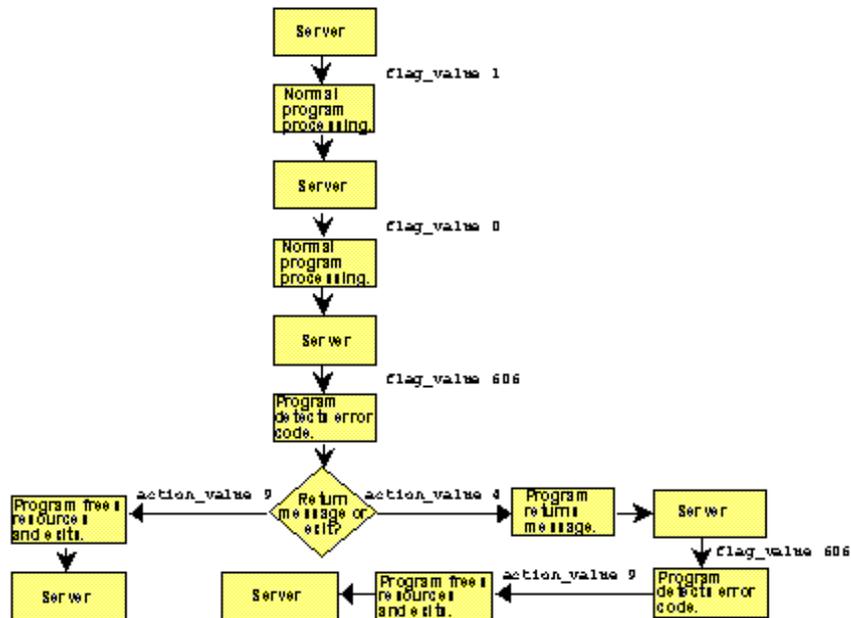
Only the `action_values` for returning a message, or for exiting, are valid after the server has reported an error. Any other `action_value` returned by the program causes the server to end without further calls to the program. If the program does return another `action_value`, the server attempts to report the program's incorrect behavior to the client application using a server-initiated message.

The following figure illustrates the correct error handling sequence. In the figure, the following `flag_values` are shown:

## Issuing the CREATE TABLE Command

- 1 Indicates the first call to the program.
- 0 Indicates a subsequent call to the program, without an error.
- 606 Indicates an error.

The program's choices when it receives the error code 606 are also illustrated.



## Issuing the CREATE TABLE Command

To return rows of table data to a client application, a program must first issue a CREATE TABLE command. It is a description of the answer set, telling the server the format of the row being returned (that is, the column name and type of data). The server uses that information to inform the client application, converting it to a format the client retrieves with the API function call EDAINFO.

The program then returns the actual rows of data in the table. The client application retrieves the data rows with the function call EDAFETCH.

A CREATE TABLE may not exceed 1,000 bytes in length.

### **Syntax** How to Issue a Create Table

```
CREATE TABLE table_name ( col_name col_type[,...] )
```

where:

*table\_name*

Is the name of the table to be created. The length and format of *table\_name* must comply with standard SQL requirements.

*col\_name*

Is the name of a column to be created. The length and format of *col\_name* must comply with standard SQL requirements. The maximum number of columns permitted in one CREATE TABLE is 254.

*col\_type*

Is the data type of the column. Possible values are:

**CHAR**(*n*) for fixed-length alphanumeric, where *n* is less than 254. The value CHAR(10) is used for date formats.

**SMALLINT** for two-byte binary integer.

**INTEGER** for four-byte binary integer.

**DECIMAL**(*p, s*) for packed decimal containing *p* digits with an implied number *s* of decimal points.

**REAL** for four-byte, single-precision floating point.

**FLOAT** for eight-byte, double-precision floating point.

**Note:** As shown in the syntax, you must include a blank:



- After *table\_name* (before the left parenthesis).
- After the left parenthesis (before *col\_name*).
- Before the right parenthesis.

Blanks are not permitted in *col\_type* definitions. For example:

- DECIMAL(15,2) is valid.
- DECIMAL ( 15 ,2 ) is invalid.

**Note:** When the CREATE TABLE specifies a DECIMAL value, the associated row must pass back the value as an eight-byte packed field. For example,



**DECIMAL**(13,2)

and

**DECIMAL**(5,2)

would require an eight-byte packed field.

In COBOL, both the above fields are defined as:

**PIC S9(13)V99 COMP-3**

*Issuing the CREATE TABLE Command*

or

PIC S9 (15) COMP-3

---

---

## CHAPTER 4

# Writing a Dialogue Manager Procedure

### Topics:

- Commands Included in a Procedure
- Commands and Processing
- Commenting a Procedure
- Sending a Message to a Client Application
- Controlling Execution
- Using Variables
- Supplying Values for Variables
- Branching
- Looping
- Calling Another Procedure
- The -REMOTE Commands
- Reading From and Writing to an External File
- The .EVAL Operator
- Creating Expressions
- Using Functions
- Using Commands Specific to an Operating System
- ON TABLE HOLD
- ON TABLE PCHOLD

A Dialogue Manager procedure is a file of commands that resides on a server. It typically includes SQL statements that perform tasks such as report generation or file maintenance, or it simply generates messages.

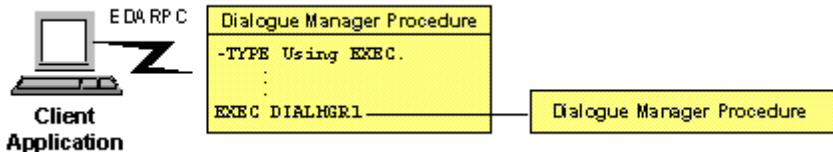
In these topics, a Dialogue Manager procedure is referred to simply as a *procedure*.

## Commands Included in a Procedure

---

A Dialogue Manager procedure must reside in a Procedure Library. See Chapter 1, *Introduction*, for details on stored procedure libraries and stored procedure execution order.

With the EXEC command, a Dialogue Manager procedure is called by the API function call EDARPC or by another Dialogue Manager procedure, or issued by a client application. This is illustrated below.



In addition to Dialogue Manager commands (described later), include the following in a procedure:

- SQL statements allowed by the server platform.
- Server commands, for example, CALLPGM, EXEC, and END. For details on CALLPGM and EXEC when used to call a program, see Chapter 2, *Calling a Program as a Stored Procedure*. This chapter discusses the use of EXEC to call another Dialogue Manager procedure.
- Commands allowed in a server profile, such as SET SQLENGINE and SET EXORDER. For details on the profile and its allowable commands, see the *Server Administration* manual.
- Commands that enable portions of a procedure to be executed on a target server. See *The -REMOTE Commands* on page 4-38 for details on the syntax and use of those commands. Also see the *Server Administration* manual for commands that connect to a target server, such as SQL EDA SET SERVER.
- The ON TABLE HOLD command, which holds an answer set in a temporary file on a server. See *ON TABLE HOLD* on page 4-60 for details on syntax and use.
- The ON TABLE PCHOLD command, which sends an answer set to a client application. See *ON TABLE PCHOLD* on page 4-61 for details on the syntax and use.
- The CALLIMS function. See Chapter 5, *Transaction Adapters for IMS/TM*, for details on the syntax and use.
- Platform-specific commands (for example, DYNAM in MVS). See Chapter 6, *Platform-specific Commands*, for details.

## Commands and Processing

The following table summarizes the available Dialogue Manager commands. Notice that every command begins with a hyphen (-).

The following sections describe the syntax and use of the commands. Appendix A, *Dialogue Manager Quick Reference*, provides an alphabetical list for your convenience.

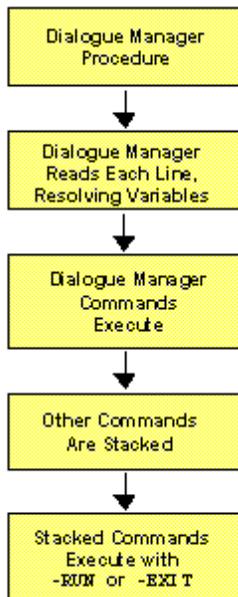
Command	Function
-*	Signals a comment.
-?	Displays the value of local variables.
-CLOSE	Closes an external file opened for reading or writing (an external file is a sequential file in the platform's file system).
-AS/400	Executes an OS/400 operating system command.
-CMS	Executes a CMS operating system command.
-DEFAULTS	Sets a variable to an initial value.
-DOS	Executes a DOS operating system command.
-EXIT	Executes stacked commands and terminates the procedure. See <i>Dialogue Manager Processing</i> on page 4-4 for a definition of stacked commands.
-GOTO	Forces an unconditional branch to a label.
-IF	Determines the execution flow based on the evaluation of an expression (a conditional branch).
-INCLUDE	Calls another Dialogue Manager procedure.
-label	Identifies a section of code that is the target of a -GOTO or -IF.
-QUIT	Terminates the procedure without executing stacked commands.
-READ	Reads data from an external file.
-REMOTE BEGIN	Signals the start of commands on an originating server that are to be sent to a target server. Only available with Hub Services.
-REMOTE END	Signals the end of commands from an originating server.
-REPEAT	Executes a loop.

<b>Command</b>	<b>Function</b>
<code>-RUN</code>	Executes stacked commands and closes any external files opened with <code>-READ</code> or <code>-WRITE</code> .
<code>-SET</code>	Sets a variable to a literal value or to a value computed in an expression.
<code>-TSO RUN</code>	Executes an MVS operating system command.
<code>-TYPE</code>	Sends a message to a client application.
<code>-UNIX</code>	Executes a UNIX operating system command.
<code>-VMS</code>	Executes a VMS operating system command.
<code>-WINNT</code>	Executes a Windows NT operating system command.
<code>-WRITE</code>	Writes data to an external file.
<code>-</code>	Line continuation of prior Dialogue Manager command.

## Dialogue Manager Processing

A procedure processes as follows:

- Dialogue Manager reads each line of the procedure, one by one. Values are substituted for variables encountered in any line.
- All Dialogue Manager commands (commands that start with a “-”) execute as they are encountered.
- Other commands are temporarily stored for subsequent execution and are called *stacked* commands.
- The Dialogue Manager commands `-RUN` and `-EXIT` execute any stacked commands.



### **Example** Issuing an API Function Call (EDARPC)

The following is an example of a procedure, with an explanation of the way it processes.

To execute this procedure, a client application issued the API function call EDARPC, specifying the procedure name SLRPT, and the parameters "COUNTRY=ENGLAND,CAR=JAGUAR":

1. `-IF &COUNTRY EQ 'DONE' THEN GOTO GETOUT;`
2. `SQL`  
`SELECT COUNTRY, CAR, MODEL, BODY`  
`FROM CAR`  
`WHERE COUNTRY='&COUNTRY' AND CAR='&CAR'`  
`ORDER BY CAR;`
3. `TABLE`  
`ON TABLE PCHOLD`  
`END`
4. `-RUN`
5. `-EXIT`  
`-GETOUT`  
`-TYPE NO PROCESSING DONE: EXITING SP`

The procedure processes as follows:

1. Values for the variables &COUNTRY and &CAR are passed to the procedure by the function call EDARPC before the first line executes. Dialogue Manager substitutes the value ENGLAND for the variable &COUNTRY in the first line and tests for the value DONE. The test fails, so Dialogue Manager proceeds to the next line.  
  
If the value were DONE instead of ENGLAND, control would pass to the label -GETOUT, and the message NO PROCESSING DONE: EXITING SP would be sent to the client application. (Dialogue Manager would skip the intervening lines of code.)
2. The next five lines are SQL. Dialogue Manager scans each for the presence of variables, substituting the value ENGLAND for &COUNTRY and the value JAGUAR for &CAR (remember, those values were passed by EDARPC). As each line is processed, it is placed on a stack to be executed later by the server.
3. The command ON TABLE PCHOLD sends the answer set to the client application.

The command END delimits ON TABLE PCHOLD.

After Dialogue Manager processes the command END, the stacked commands look like this:

```
SQL
SELECT COUNTRY, CAR, MODEL, BODY
FROM CAR
WHERE COUNTRY= 'ENGLAND' AND CAR='JAGUAR'
ORDER BY CAR;
TABLE
ON TABLE PCHOLD
END
```

The next line is then processed by Dialogue Manager.

4. The Dialogue Manager command -RUN sends the stacked commands to the server for execution.
5. The Dialogue Manager command -EXIT terminates the procedure.

## Commenting a Procedure

---

It is good practice to include comments in a procedure for the benefit of others who may use it.

It is particularly recommended that you use comments in a procedure header to supply the date, the version, and other relevant information.

Comments are preceded with a hyphen and an asterisk (-\*). You can:

- Include any text after the -\*.
- Start the text immediately after the -\*, omitting any space.

- Place comments at the beginning or end of a procedure, or in between commands. A comment cannot be on the same line as a command (for example, -RUN -\*Comment is invalid).

The following example illustrates the use of comments at the beginning of a procedure to supply information about it.

```

-* Version 1 07/28/99 SLRPT
-* Component of Retail Sales Reporting Module
SQL
.
.
.

```

## Sending a Message to a Client Application

---

The command -TYPE enables you to send a message to a client application while a procedure is processing. The message:

- Explains the purpose of the procedure.
- Displays the results of a calculation.
- Presents any kind of useful information.

### **Syntax** How to Send a Message to a Client Application

```
-TYPE text
```

where:

*text*

Is the message to be sent, followed by a line feed. If you include quotation marks around text, the quotes are displayed as part of the message. The length of text can be up to 256 bytes. The message is sent as soon as -TYPE is encountered in the processing of the procedure.

Use the following syntax

```
-label [TYPE text]
```

where:

-*label*

Is the target of a -GOTO or -IF.

TYPE *text*

Optionally sends a message to a client application.

**Example Using the -TYPE Command to Inform a Client Application About Report Content**

The following example illustrates the use of -TYPE to inform a client application about the content of a report.

```
-* Version 1 07/28/99 SLRPT
-* Component of Retail Sales Reporting Module
-TYPE This report calculates percentage of returns.
SQL
.
.
.
```

## Controlling Execution

---

Dialogue Manager enables you to manage the flow of execution with these commands:

- -RUN
- -EXIT
- -QUIT

### Executing Stacked Commands: -RUN

The Dialogue Manager command -RUN causes immediate execution of all stacked commands and closes any external files opened with -READ or -WRITE. Following execution, processing of the procedure continues with the line that follows -RUN.

**Example Using the -RUN Command**

The following example illustrates the use of -RUN to execute stacked SQL code and then return to the procedure.

```
1. -TYPE This report calculates percentage of returns.
2. SQL
.
.
.
END
3. -RUN
4. -TYPE This routine adds data to the sales file.
SQL
.
.
.
```

The procedure processes as follows:

1. The command `-TYPE` sends a message to the client application.
2. The SQL code is stacked.
3. The command `-RUN` sends the stacked commands to the server, which then executes the stacked command and sends the output to the client application.
4. Processing continues with the line following `-RUN`. In this case, another message is sent to the client application and a second SQL request is initiated.

## Executing Stacked Commands and Exiting the Procedure: `-EXIT`

Like the `-RUN` command, the Dialogue Manager command `-EXIT` forces execution of stacked commands as soon as it is encountered. However, instead of returning to the procedure, `-EXIT` closes all external files, terminates the procedure, and exits. If the procedure that is processing was called by another procedure, control returns to the calling procedure.

### *Example* Using the `-EXIT` Command

In the following example, either the first SQL request or the second SQL request executes, but not both.

1. `-TYPE This report calculates percentage of returns.`
2. `-IF &PROC EQ 'UPDATE' GOTO UPDATE;`
3. `-REPORT`  
`SQL`  
`.`  
`.`  
`.`  
`END`
4. `-EXIT`
5. `-UPDATE`  
`SQL`  
`.`  
`.`  
`.`  
`END`

The procedure processes as follows:

1. The command `-TYPE` sends a message to the client application.
2. Assume the value passed to `&PROC` is `REPORT`.

The `-IF` test checks the value of `&PROC`. Since it is not equal to `UPDATE`, control passes to the label `-REPORT`.

3. The SQL code is stacked. Control passes to the next line, -EXIT.
4. The command -EXIT executes the stacked commands. The output is sent to the client application and the procedure is exited.
5. The SQL request under the label -UPDATE is not executed.

This example also illustrates an *implicit exit*. If the value of &PROC were UPDATE, control would pass to the label -UPDATE after the -IF test, and the procedure would never encounter the -EXIT. The second SQL request would execute and the procedure would automatically terminate.

## Canceling Execution: -QUIT

The Dialogue Manager command -QUIT cancels execution of any stacked commands and causes an immediate exit from the procedure. If the procedure that is processing was called by another procedure, control returns directly to the client application, not to the calling procedure.

This command is useful if tests or computations generate results that make additional processing unnecessary.

### Example Using the -QUIT Command

The following example illustrates the use of -QUIT to cancel execution based on the results of an -IF test.

1. `-TYPE This report calculates percentage of returns.`  
`SQL`  
`.`  
`.`  
`.`
2. `-IF &CODE GT 'B10' OR &CODE EQ 'DONE' GOTO QUIT;`  
`END`
3. `-QUIT`

The procedure processes as follows:

1. The command -TYPE sends a message to the client application. The SQL code is stacked.  
The command -IF tests the value and passes control to -QUIT.  
The command END is a delimiter.
2. Assume that the value of &CODE is B11.  
The command -QUIT cancels execution of the stacked commands and exits the procedure.

## Using Variables

---

This section describes how to use variables in a procedure.

Variables fall into two categories:

- Local and global variables, whose values must be supplied by the procedure at run time.
- System and statistical variables, whose values are automatically supplied by the system when referenced.

The following features apply to all variables:

- A variable stores numbers or a string of text, and is placed anywhere in a procedure.
- A variable refers to a command, a database field, a verb, or a phrase. *Variables and Command Structures* on page 4-19 contains examples.
- The maximum number of variables allowed in a procedure is 1,024. Because approximately 30 are reserved for server use, the maximum number of user-named variables allowed in a procedure is 994.

## Naming Conventions

This section describes how to use variables in a procedure.

Variables fall into two categories:

- Local and global variables, whose values must be supplied by the procedure at run time.
- System and statistical variables, whose values are automatically supplied by the system when referenced.

Local and global variable names are user-defined, while system and statistical variables have predefined names.

The following rules apply to the naming of local and global variables:

- A local variable name is always preceded by an ampersand (&).
- A global variable name is always preceded by a double ampersand (&&).
- The maximum number of characters permitted in a name is 12, excluding the first ampersand.
- Embedded blanks are not permitted.
- If an anticipated value for a variable might contain an embedded blank, enclose the variable in single quotation marks when you refer to it.

- A variable name may be any combination of the characters A through Z, 0 through 9, and the underscore (\_). The first character of the name must be A through Z.
- Assign a number to a variable, instead of a name, to create a positional variable.

## **Syntax** How to Use Variables in a Procedure

`&[&] name`

where:

`&name`

Is the user-defined name of a local variable. The first character of *name* must be A through Z.

`&&name`

Is the user-defined name of a global variable. The first character of *name* must be A through Z.

The following variables are properly named:

```
&WHICHPRODUCT
&WHICH_CITY
' &CITY '
&&CITY
```

The following variables are improperly named for the reason given:

<b>Invalid</b>	<b>Reason</b>
<code>&amp;CORPORATECITY</code>	Too long (exceeds 12 characters).
<code>&amp;WHICH CITY</code>	Contains embedded blank.
<code>&amp;WHICH-CITY</code>	Contains a hyphen (-).
<code>WHICHCITY</code>	Leading ampersand(s) is missing.

## **Local Variables**

Once supplied, values for local variables remain in effect throughout a single procedure. The values are lost after the procedure finishes processing and are *not* passed to other procedures that contain the same variable name.

### **Example** Using Local Variables

Consider the following procedure in which &CITY, &CODE1, and &CODE2 are local variables.

```

.
.
.
SQL
SELECT SUM (UNIT_SOLD) ,
        SUM (RETURNS)
FROM SALES
WHERE CITY = '&CITY'
AND PROD_CODE >= '&CODE1 '
AND PROD_CODE <= '&CODE2 '
.
.
.

```

Assume you supply the following values when you call the procedure:

```
CITY=STAMFORD, CODE1=B10, CODE2=B20
```

Dialogue Manager substitutes the values for the variables as follows:

```

.
.
.
SQL
SELECT SUM (UNIT_SOLD) ,
        SUM (RETURNS), CITY
FROM SALES
WHERE CITY = STAMFORD
AND PROD_CODE >= B10
AND PROD_CODE <= B20
GROUP BY CITY, PROD_CODE
.
.
.

```

After the procedure executes and terminates, the values STAMFORD, B10, and B20 are lost.

### **Example** **Creating an Indexed Variable**

Append the value of one variable to the name of another, creating an indexed variable. This feature applies to both local and global variables.

If the index value is numeric, the effect is similar to that of an array in traditional computer programming languages. For example, if the value of index &K varies from 1 to 10, the variable &AMOUNT.&K refers to one of ten variables, from &AMOUNT1 to &AMOUNT10.

A numeric index is used as a counter; it is set, incremented, and tested in a procedure.

You create an indexed variable with the command -SET

```
-SET &name.&index[.&index...] = expression;
```

where:

*&name*

Is a variable.

*.&index*

Is a numeric or alphanumeric variable whose value is appended to *&name*. The period (.) is required.

*[.&index...]*

Represents any number of indices. When more than one index is used, all index values are concatenated and the string appends to the name of the variable. For example, *&V.&I.&J.&K* is equivalent to *&V1120* when *&I=1*, *&J=12*, and *&K=0*.

*expression*

Is a valid expression. See *Creating Expressions* on page 4-41 for information on the kinds of expressions allowed.

An indexed variable is used in a loop. The following example creates the equivalent of a DO loop used in traditional programming languages.

```
-SET &N = 0;  
-LOOP  
-SET &N = &N+1;  
-IF &N GT 12 GOTO OUT;  
-SET &MONTH.&N=&N;  
-TYPE &MONTH.&N  
-GOTO LOOP  
-OUT
```

In this example, *&MONTH* is the indexed variable and *&N* is the index. The value of the index is supplied through the command *-SET*; the first *-SET* initializes the index to 0, and the second *-SET* increases the index by increments each time the procedure goes through a loop.

If the value of an index is not defined prior to reference, a blank value is assumed. As a result, the name (and value) of the indexed variable does not change.

Indexed variables are included in the system limit of 994.

### **Example** Displaying the Value of Local Variables

To display the current value of a local variable, enter the following in a procedure

```
-? &[string]
```

where:

*string*

Is an optional variable name of up to 12 characters. If this parameter is not specified, the current values of all local, global, and defined system and statistical variables are displayed.

## Global Variables

Once a value is supplied for a global variable, it remains in effect throughout the session of a processing service, unless cleared by the server. All procedures that contain the same global variable name receive the supplied value until you terminate the session.

### Example Using Global Variables

The following example illustrates the use of three global variables: &&CITY, &&CODE1, &&CODE2.

```

.
.
.
SQL
SELECT SUM (UNIT_SOLD) ,
        SUM (RETURNS)
FROM SALES
WHERE CITY = &&CITY
AND PROD_CODE >= &&CODE1
AND PROD_CODE <= &&CODE2
;
TABLE
ON TABLE PCHOLD
END

```

### Example Displaying the Value of Global Variables

To display the current value of all global variables, enter the following command in a procedure:

```
? &&
```

## System Variables

The table in this section describes system variables that are used in a procedure. Dialogue Manager automatically supplies values for system variables whenever the variables are encountered.

Unless otherwise noted in the table, override system-supplied values by replacing the values with values specified:

- In the function call EDARPC when you execute the procedure.

- In an EXEC command. See *Supplying Values for Variables* on page 4-19 for information.

System Variable	Description	Format or Value
&DATE	Current date (four-digit year).	MM/DD/YY
&DMY &DATE <i>fmt</i>	Current date.	DDMMYY <i>fmt</i> is any combination of YYMD, MDYY, etc.
&DMYY	Current date (four-digit year).	DDMMCCYY
&MDY	Current date. Useful for numeric comparisons.	MMDDYY
&YMD	Current date.	YYMMDD
&YYMD	Current date (four-digit year).	CCYYMMDD
&FOCFOCEXEC		Manages reporting operations involving many similarly named requests that are executed using EX. &FOCFOCEXEC enables you to easily determine which procedure is running. &FOCFOCEXEC is specified within a request or in a Dialogue Manager command to display the name of the currently running procedure.
&FOCINCLUDE		Manages reporting operations involving many similarly named requests that are included using -INCLUDE. &FOCINCLUDE is specified within a request or in a Dialogue Manager command to display the name of the current included procedure.

&MDYY	Current date (four-digit year).	MMDDCCYY
&DMY	Current date.	DDMMYY
&DMYY	Current date (four-digit year).	DDMMCCYY
&YMD	Current date.	YYMMDD
&YYMD	Current date (four-digit year).	CCYYMMDD
&FOCFOCEXEC		Manages reporting operations involving many similarly named requests that are executed using EX. &FOCFOCEXEC enables you to easily determine which procedure is running. &FOCFOCEXEC is specified within a request or in a Dialogue Manager command to display the name of the currently running procedure.
&FOCINCLUDE		Manages reporting operations involving many similarly named requests that are included using -INCLUDE. &FOCINCLUDE is specified within a request or in a Dialogue Manager command to display the name of the current included procedure.

System Variable	Description	Format or Value
<code>&amp;FOCPRINT</code>	Current print setting.	<code>ONLINE</code> <code>OFFLINE</code>
<code>&amp;FOCREL</code>	Source code release number.	Release number (for example, R720520B).
<code>&amp;IORETURN</code>	Value returned after the last Dialogue Manager -READ or -WRITE operation.	0 Successful operation 1 End or failure
<code>&amp;RETCODE</code>	Value returned after a server or operating system command is executed.  &RETCODE executes all stacked commands, like the command -RUN.	Any value returned by the server command is valid (for example, CALLPGM flag values).
<code>&amp;TOD</code>	Current time. When you enter FOCUS, this variable is updated to the current system time only when you execute a MODIFY, SCAN, or FSCAN command. To obtain the exact time during any process, use the HHMMSS subroutine.	<code>HH.MM.SS</code>
<code>&amp;FOCNET</code>	Environment.	<code>CLIENT</code> , <code>SERVER</code>  You cannot override the system-supplied value.

**Example Using System Variables**

The following example incorporates the system variable `&DATE` into an SQL request, testing a user-supplied variable (`IDATE`) against it.

```
SQL
  SELECT '&DATE', IDATE
  FROM filename
  WHERE IDATE < '08/08/1996'
-EXIT
```

## Variables and Command Structures

A variable refers to a command, a database field, a verb, or a phrase. In this way, the command structure of a procedure is determined by the value of the variable.

### **Example** Using Variables to Alter Commands

In the following example, the variable &FIELD determines which field to SELECT in the SQL request. For example, &FIELD could have the value RETURNS, DAMAGED, or UNIT\_SOLD from a database named SALES.

```
SQL
.
.
.
SELECT &FIELD
ORDER BY PROD_CODE
.
.
.
```

## Supplying Values for Variables

---

You must supply values for variables in a procedure even if the value is a blank. For instance, some server commands are invalid without values but process normally with blanks.

Supply values for variables in the following ways:

- In the function call EDARPC. See the *API Reference* manual for information on the syntax of EDARPC.
- Within the procedure itself:
  - In the command EXEC.
  - With a command such as -DEFAULTS, -SET, or -READ.

This section describes the second method, within the procedure itself.

## General Rules

The following general rules apply to values for variables:

- The maximum length is 80 characters.

- A physical stack line with values substituted for variables cannot exceed 80 characters.
- Once a value is supplied for a local variable, it is used throughout the procedure unless it is changed with a command such as -SET or -READ.
- Once a value is supplied for a global variable, it is used throughout the session in all procedures unless it is changed with -SET, -READ, or another command.

## Supplying Values in the EXEC Command

The command EXEC enables you to call one procedure from another and set values for variables in the called procedure, using:

- Keyword parameters.
- Positional parameters.
- A combination of keyword and positional parameters.

A parameter list specified on the command line has a maximum string length of up to 4096 bytes including the "EX" and file name portions. A string of this length is typically built using concatenation of values.

**Note:** EX lines have special treatment. Normally, all other lines are limited to 80 characters.



### Example Supplying Values With the EXEC Command

Consider the following procedure named SLRPT:

```
.  
. .  
. .  
SQL  
SELECT SUM(UNIT_SOLD) , SUM(RETURNS) , PROD_CODE, CITY  
FROM SALES  
WHERE PROD_CODE BETWEEN '&CODE1' AND '&CODE2'  
AND CITY = '&CITY'  
GROUP BY PROD_CODE, CITY  
. .  
. .  
. .
```

This procedure is called by another procedure using EXEC, with the values for &CODE1, &CODE2, and &CITY supplied on the command line as keyword parameters:

```
EXEC SLRPT CODE1=A, CODE2=D, CITY=NYC
```

### Syntax How to Pass Keyword Parameters

```
EX[EC] procedure name=value[,...]
```

where:

*procedure*

Is the name of the called procedure.

*name=value*

Is a keyword parameter.

If *value* contains an embedded comma, blank, or equal sign, it must be enclosed in single quotation marks. For example:

```
EX SLRPT AREA=S, CITY='NY, NY'
```

Name=value pairs must be separated by commas. You do not need to enter pairs in the order in which they are encountered in the procedure.

If the list of parameters exceeds the width of the command line, insert a comma as the last character on the line and enter the rest of the list on the following line, as shown here:

```
EX SLRPT AREA=S,CITY=STAMFORD,VERB=COUNT,
FIELDS=UNIT_SOLD, CODE1=B10, CODE2=B20
```

## Syntax

### How to Pass Positional Parameters

```
EX[EC] procedure parm1 [, ...]
```

where:

*procedure*

Is the name of the called procedure.

*parm1*

Is a positional parameter. You do *not* need to specify the number in the parameter list. Dialogue Manager matches the values, one by one, to the positional variables as they are encountered in the called procedure.

However, you must specify the parameters in the order in which to be used in the called procedure.

Consider the following called procedure:

```
.
.
.
SQL
SELECT SUM(UNIT_SOLD) , SUM(RETURNS) , RETURNS/UNIT_SOLD
FROM SALES
WHERE PROD_CODE BETWEEN '&1' AND '&2'
AND CITY = '&3'
```



## Debugging Execution Flow

Dialogue Manager implements IF THEN ELSE and other flow logic such as -GOTO. Dynamically display the flow of execution using the &ECHO variable.

### **Syntax** How to Display Command Lines While Executing

`&ECHO = display`

Valid values are:

`ON`

Displays lines that are expanded and stacked for execution.

`ALL`

Displays Dialogue Manager commands as well as lines that are expanded and stacked for execution.

`OFF`

Suppresses display of both stacked lines and Dialogue Manager commands. This value is the default.

Set &ECHO through -DEFAULTS, -SET, or on the command line. For example, set ECHO to ALL for the execution of the procedure SLRPT using any of the following commands:

`-DEFAULTS &ECHO = ALL`

or

`-SET &ECHO = ALL;`

or

`EX SLRPT ECHO = ON`

If you use -SET or -DEFAULTS in the procedure, display begins from that point in the procedure and is turned off and on again at any other point.

Note that if the procedure is encrypted, &ECHO automatically receives the value OFF, regardless of the value that is assigned explicitly.

## The -DEFAULTS Command

The Dialogue Manager command -DEFAULTS supplies an initial (default) value for a variable that had no value before the command was processed. It ensures that values are passed to variables whether or not they are provided elsewhere.

### **Syntax** How to Supply Values With the -DEFAULTS Command

`-DEFAULTS [&]&name=value [...]`

where:

*&name*

Is the name of the variable.

*value*

Is the default value assigned to the variable.

### **Example** Setting Default Values With the **-DEFAULTS** Command

In the following example, **-DEFAULTS** sets default values for **&CITY** and **&REGIONMGR**.

```
-DEFAULTS &CITY=STAMFORD, &REGIONMGR=SMITH
-TYPE Default values are Stamford, Smith.
SQL
.
.
.
```

### **Reference** Overriding Default Values

Override default values by supplying new values:

- In the function call EDARPC.
- In the command EXEC.
- With the command **-SET** subsequent to **-DEFAULTS**.

For example, if you issued the following EXEC command, the specified value for REGIONMGR (JONES) would override the value SMITH in the previous example:

```
EX SLRPT REGIONMGR=JONES
```

## **The -SET Command**

With the **-SET** command, assign a value computed in an expression.

### **Syntax** How to Supply Values With the **-SET** Command

```
-SET &[&]name=expression;
```

where:

*&name*

Is the name of the variable.

*expression*;

Is a valid expression. Expressions occupy several lines, so end the command with a semicolon (;).

A literal value to a variable is also assigned. Single quotation marks around the literal value are optional unless it contains embedded blanks or commas, in which case you must include quotation marks:

```
-SET &NAME='JOHN DOE';
```

To assign a literal value that includes a single quotation mark, place two single quotation marks where you want one to appear:

```
-SET &NAME='JOHN O 'HARA';
```

The length of a literal value is limited by how the value is constructed. A single simple value (such as the 'JOHN DOE' example) is limited to a line length of 80. However, using concatenation, several lines of separate values may be joined together. For example,

```
-SET &LONG = 'xxxxx...';
-SET &LONGER = &LONG|&LONG|&LONG;
-SET &VERYLONG = &LONGER|&LONGER;
```

Through this technique of building strings, the total maximum length is 4K (2K to the right of the equal sign and 2K to the left of the equal sign). In the case of an expression, the left side of the equal sign could be of a minimal length, with the remainder of the length on the right side of the equal sign.

### **Example** Assigning Values With the **-SET** Command

In the following example, **-SET** assigns the value 14Z or 14B to the variable **&STORECODE**, as determined by the logical **IF** expression. The value of **&CODE** is supplied by the user.

```
-TYPE THIS REPORT IS FOR &CODE.
-SET &STORECODE = IF &CODE GT C2 THEN '14Z' ELSE '14B';
  SQL
  SELECT SUM(UNIT_SOLD) ,SUM(RETURNS) ,RETURNS/UNIT_SOLD
  FROM SALES
  WHERE PROD_CODE BETWEEN '&CODE1' AND '&CODE2'
  AND STORE_CODE = '&STORECODE'
  GROUP BY PROD_CODE, STORECODE
  .
  .
  .
```

### **The -READ Command**

With the **-SET** command, assign a value computed in an expression.

Supply values for variables by reading each from an external file. The file is fixed format (the data is in fixed columns) or free format (the data is delimited by commas).

### **Syntax** How to Supply Values With the **-READ** Command

```
-READ filename[,] [NOCLOSE] &name[.format.][,]...
```

where:

*filename[, ]*

Is the name of the external file, which must be defined to the operating system. A space after *filename* denotes a fixed-format file, while a comma denotes a free-format file. For more information, see Chapter 6, *Platform-specific Commands*.

**NOCLOSE**

Optionally keeps the external file open until the -READ operation is complete. Files kept open with NOCLOSE are closed using the command -CLOSE *filename*.

**Note:** The -RUN command does not close an external file if NOCLOSE is specified.



*&name[, ]...*

Is a list of variables. For free-format files, you may optionally separate the variable names with commas.

*.format.*

Is the format of the variable. For free-format files, you do not have to specify this value, but you may. For fixed-format files, *format* is the length or the length and type of the variable (A is the default type). The value of *format* must be delimited by periods. See *Specifying Length* on page 4-26.

If the list of variables is longer than one line, end the first line with a comma (,) and begin the next line with a hyphen (-) when you are reading a free-format file:

```
-READ EXTFILE, &CITY, &CODE1,  
- &CODE2
```

When you are reading a fixed-format file, begin the next line with a hyphen and comma (-,):

```
-READ EXTFILE &CITY.A8. &CODE1.A3.,  
-, &CODE2.A3.
```

The line immediately following a -READ is typically a check of &IORETURN for end of file, as shown in the following example.

### Example Specifying Length

Instead of using *.format.*, specify the length of a variable using -SET. For example:

```
-SET &CITY='          ' ;  
-SET &CODE1='      ' ;  
-SET &CODE2='      ' ;
```

**Example Using the -READ Command**

The file name parameter is a symbolic name for a physical file known to the server operating system. Each operating system has its own method for associating a symbolic name with a physical file.

For instance, under MVS, the DYNAM command is used:

```
DYNAM ALLOC FILE EXTFILE DSNNAME EDAUSER.EXTFILE.DATA SHR
```

On UNIX, Windows, OpenVMS, and OS/400 platforms, the FILEDEF command is used:

```
FILEDEF MYFILE DISK /home/edauser/extfile.dat
```

where the data file portion would use the actual native file name convention of the platform.

Assume that EXTFILE is a fixed-format file containing the data:

```
STAMFORDB10B20
```

To detect the end of file, the following code tests the system variable &IORETURN. When no records remain to be read, its value is not equal to zero.

```
-READ EXTFILE &CITY.A8. &CODE1.A3. &CODE2.A3.
-IF &IORETURN NE 0 GOTO RUN;
  SQL
  SELECT SUM(UNIT_SOLD) ,SUM(RETURNS)
  FROM SALES
  WHERE PROD_CODE BETWEEN '&CODE1' AND '&CODE2'
  AND CITY = '&CITY'
  GROUP BY PROD_CODE,CITY
-RUN
```

**Branching**

The execution flow of a procedure is determined using the following commands:

- `-GOTO`. Used for unconditional branching, `-GOTO` transfers control to a label.
- `-IF...GOTO`. Used for conditional branching, `-IF...GOTO` transfers control to a label depending on the outcome of a test.

**Syntax How to Use the -GOTO Command for Unconditional Branching**

```
-GOTO label
.
.
.
-label [TYPE text]
```

where:

*label*

Is a user-defined name of up to 12 characters. Do not use embedded blanks or the name of any other Dialogue Manager command except -QUIT or -EXIT. Do not use words that may be confused with functions or arithmetic or logical operations.

The label may precede or follow the -GOTO command in the procedure.

TYPE *text*

Optionally sends a message to a client application.

**Reference Processing a -GOTO Command**

Dialogue Manager processes a -GOTO as follows:

- It searches forward through the procedure for the target label. If it reaches the end without finding the label, it continues the search from the beginning of the procedure.
- The first time through a procedure, Dialogue Manager notes the addresses of all the labels so that they are found immediately if needed again.
- If a -GOTO does not have a corresponding label, execution halts and an error message is displayed.

**Example Using the -GOTO Command for Unconditional Branching**

The following example comments out all the SQL code using an unconditional branch rather than -\* in front of every line.

```
-START TYPE PROCESSING BEGINS
-GOTO DONE
SQL
SELECT SUM(UNIT_SOLD) , SUM(RETURNS)
FROM SALES
WHERE PROD_CODE BETWEEN '&CODE1' AND '&CODE2'
AND PRODUCT = '&PRODUCT'
GROUP BY PROD_CODE,CITY
-RUN
-DONE
```

The next example illustrates two labels with TYPE messages appended:

```
.
.
.
-PRODSALES TYPE TOTAL SALES BY PRODUCT
.
.
-PRODRETURNS TYPE TOTAL RETURNS BY PRODUCT
```

**Syntax**    **How to Use the -IF..GOTO Command for Conditional Branching**

```
-IF expression [THEN] GOTO label1[:] [ELSE GOTO label2[:]]
                               [ELSE IF...[:]]
```

where:

*label*

Is a user-defined name of up to 12 characters. Do not use embedded blanks or the name of any other Dialogue Manager command except -QUIT or -EXIT. Do not use words that may be confused with functions or arithmetic or logical operations.

The label may precede or follow the -IF command in the procedure.

*expression*

Is a valid expression. Literals need not be enclosed in single quotation marks unless they contain embedded blanks or commas.

THEN

Is an optional keyword that increases readability of the command.

ELSE GOTO

Optionally passes control to *label2* when the -IF test fails.

ELSE IF

Optionally specifies a compound -IF test. See *Using Compound -IF Tests* on page 4-30.

The command -IF must end with a semicolon (;) to signal that all logic has been specified. Continuation lines must begin with a hyphen (-) and lines must break between words. A space after the hyphen is not required, but adds to readability.

**Example**    **Using the -IF..GOTO Command for Conditional Branching**

In the following example, control passes to the label -PRODSALES if &OPTION is equal to S. Otherwise, control falls through to the label -PRODRETURNS, the line following the -IF test.

```
-IF &OPTION EQ 'S' GOTO PRODSALES;
-PRODRETURNS
    SQL
    .
    .
    END
-EXIT
-PRODSALES
    SQL
    .
    .
    END
-EXIT
```

The following command specifies both transfers explicitly:

```
-IF &OPTION EQ 'S' GOTO PRODSALES ELSE  
- GOTO PRODRETURNS;
```

Notice that the continuation line begins with a hyphen (-).

### **Example** Using Compound -IF Tests

Use compound -IF tests provided each test specifies a target label.

In the following example, if the value of &OPTION is neither R nor S, the procedure terminates (GOTO QUIT). The -QUIT serves both as a target label for the GOTO and as an executable command.

```
-IF &OPTION EQ 'R' THEN GOTO PRODRETURNS ELSE IF  
- &OPTION EQ 'S' THEN GOTO PRODSALES ELSE  
- GOTO QUIT;  
.  
.  
.  
-QUIT
```

### **Reference** Operators and Functions in -IF Tests

Expressions in an -IF test include arithmetic and logical operators, as well as available functions. See *Creating Expressions* on page 4-41 and *Using Functions* on page 4-47 for details.

## Screening Values With -IF Tests

To ensure that a supplied value is valid in a procedure, test for its:

- Presence
- Type
- Length

For instance, you would not want to perform a numerical computation on a variable for which alphanumeric data has been supplied.

### **Syntax** How to Test for the Presence of a Value

```
-IF &name.EXIST expression GOTO label...;
```

where:

*&name*

Is a user-supplied variable.

`.EXIST`

Indicates that you are testing for the presence of a value. If a value is not present, a zero (0) is passed to the expression. Otherwise, a non-zero value is passed.

*expression*

Is a valid expression that uses *&name*. EXIST as an amper variable.

`GOTO label`

Specifies a label to branch to.

### **Example** Testing for the Presence of a Variable

In the following example, if no value is supplied, `&OPTION.EXIST` is equal to zero and control is passed to the label `-CANTRUN`. The procedure sends a message to the client application and then exits. If a value is supplied, control passes to the label `-PRODSALES`.

```
-IF &OPTION.EXIST GOTO PRODSALES ELSE GOTO CANTRUN;
.
.
.
-PRODSALES
  SQL
.
.
.
  END
-EXIT
-CANTRUN
-TYPE TOTAL REPORT CAN'T BE RUN WITHOUT AN OPTION.
-EXIT
```

### **Syntax** How to Test for the Length of a Value

```
-IF &name.LENGTH expression GOTO label...;
```

where:

*&name*

Is a user-supplied variable.

`.LENGTH`

Indicates that you are testing for the length of a value. If a value is not present, a zero (0) is passed to the expression. Otherwise, the number of characters in the value is passed.

*expression*

Is a valid expression.

`GOTO label`

Specifies a label to branch to.

### **Example**    **Testing for Variable Length**

In the following example, if the length of &OPTION is greater than one, control passes to the label -FORMAT, which informs the client application that only a single character is allowed.

```
-IF &OPTION.LENGTH GT 1 GOTO FORMAT ELSE  
-GOTO PRODSALES;  
.  
.  
.  
-PRODSALES  
  SQL  
  .  
  .  
  .  
  END  
-EXIT  
-FORMAT  
-TYPE ONLY A SINGLE CHARACTER IS ALLOWED.
```

### **Syntax**    **How to Test for the Type of a Value**

```
-IF &name.TYPE expression GOTO label...;
```

where:

*&name*

Is a user-supplied variable.

`.TYPE`

Indicates that you are testing for the type of a value. The letter N (numeric) is passed to the expression if the value is interpreted as a number up to  $10^9-1$  and is stored in four bytes as a floating point format. In Dialogue Manager, the result of an arithmetic operation with numeric fields is truncated to an integer after the whole result of an expression is calculated. If the value could not be interpreted as numeric, the letter A (alphanumeric) is passed to the expression.

*expression*

Is a valid expression.

`GOTO label`

Specifies a label to branch to.

**Example Testing for Variable Type**

In the following example, if &OPTION is not alphanumeric, control passes to the label -NOALPHA, which informs the client application that only alphanumeric characters are allowed.

```
-IF &OPTION.TYPE NE A GOTO NOALPHA ELSE
- GOTO PRODSALES;
.
.
.
-PRODSALES
  SQL
.
.
.
  END
-EXIT
-NOALPHA
-TYPE ENTER A LETTER ONLY.
```

**Looping**

The Dialogue Manager command -REPEAT allows looping in a procedure.

**Syntax How to Use the -REPEAT Command**

```
-REPEAT label {n TIMES [FROM fromval] [TO toval] [STEP s]}
-REPEAT label {WHILE condition [FROM fromval] [TO toval] [STEP s]}
-REPEAT label {FOR &variable [FROM fromval] [TO toval] [STEP s]}
```

where:

*label*

Identifies the code to be repeated (the loop). A label includes another loop if the label for the second loop has a different name from the first.

*n* TIMES

Specifies the number of times to execute the loop. The value of *n* is a local variable, a global variable, or a constant. If it is a variable, it is evaluated only once, so the only way to end the loop early is with -QUIT or -EXIT (the number of times to execute the loop cannot be changed).

WHILE *condition*

Specifies the condition under which to execute the loop. The condition is any logical expression that is either true or false. The loop is run if the condition is true.

### FOR *&variable*

Is a variable that is tested at the start of each execution of the loop. It is compared with the value of *fromval* and *toval* (if supplied). The loop is executed only if *&variable* is less than or equal to *toval* (STEP is positive), or greater than or equal to *toval* (STEP is negative).

### FROM *fromval*

Is a constant that is compared with *&variable* at the start of each execution of the loop. The default value is 1.

### TO *toval*

Is a value that is compared with *&variable* at the start of each execution of the loop. The default is 1,000,000.

### STEP *s*

Is a constant used to increment *&variable* at the end of each execution of the loop. It may be positive or negative. The default value is 1.

The parameters FROM, TO, and STEP appear in any order.

## Ending a Loop

A loop ends in one of three ways:

- It executes in its entirety.
- A -QUIT or -EXIT is issued.
- A -GOTO is issued to a label outside of the loop.

**Note:** If you later issue another -GOTO to return to the loop, the loop proceeds from the point where it left off.



## Example Using the -REPEAT Command

This section illustrates how to write each of the syntactical elements of -REPEAT.

1. `-REPEAT label n TIMES`

Example:

```
-REPEAT LAB1 2 TIMES
-TYPE INSIDE
-LAB1 TYPE OUTSIDE
```

The output is:

```
INSIDE
INSIDE
OUTSIDE
```

2. `-REPEAT label WHILE condition`

Example:

```
-SET &A = 1;
-REPEAT LABEL WHILE &A LE 2;
- TYPE &A
-SET &A = &A + 1;
-LABEL TYPE END: &A
```

The output is:

```
1
2
END: 3
```

3. `-REPEAT label FOR &variable FROM fromval TO toval STEP s`

Example:

```
-REPEAT LABEL FOR &A FROM 1 TO 4 STEP 2
- TYPE INSIDE &A
-LABEL TYPE OUTSIDE &A
```

The output is:

```
INSIDE 1
INSIDE 3
OUTSIDE 5
```

## Calling Another Procedure

---

One procedure calls another procedure using:

- The command `-INCLUDE`, which incorporates a whole or partial procedure and executes immediately when encountered. (A partial procedure might contain header text, or code to include at run time based on a test in the calling procedure.)
- The command `EXEC`. The command is stacked and executed when the appropriate Dialogue Manager command is encountered. The called procedure must be fully executable.

### **Syntax** How to Use the `-INCLUDE` Command

Lines incorporated with a `-INCLUDE` are processed as though they had been placed in the calling procedure originally.

```
-INCLUDE filename
```

where:

*filename*

Is the name of the called procedure.

A calling procedure cannot branch to a label in a called procedure, and vice versa.

### **Example** Using the **-INCLUDE** Command

In the following example, Dialogue Manager searches for a procedure named DATERPT as specified on the command -INCLUDE.

```
-IF &OPTION EQ 'S' GOTO PRODSALES
- ELSE GOTO PRODRETURNS;
.
.
.
-PRODRETURNS
-INCLUDE DATERPT
-RUN
.
.
```

Assume that DATERPT contains the following SQL code:

```
SQL
SELECT PROD_CODE UNIT_SOLD
FROM SALES
WHERE PROD_CODE = '&PRODUCT';
TABLE
ON TABLE PCHOLD
END
```

Dialogue Manager incorporates this code into the original procedure. It substitutes a value for the variable &PRODUCT as soon as the -INCLUDE is encountered. The ensuing command -RUN executes the SQL request.

The following is an example of a -INCLUDE that calls a partial procedure named OBJECTS:

```
SQL
SELECT
-INCLUDE OBJECTS
FROM CAR
WHERE RETAIL_COST < 10000;
```

The procedure OBJECTS contains the fields to use:

```
COUNTRY, CAR, MODEL
```

The resulting stacked commands are:

```
SQL
SELECT
COUNTRY, CAR, MODEL
FROM CAR
WHERE RETAIL_COST < 10000;
```

## Nesting

Any number of different procedures is invoked from a single calling procedure. Nest `-INCLUDE` commands within each other, up to four levels deep:

```
-PRODSALES
-INCLUDE FILE1
-RUN

      FILE1
      -INCLUDE FILE2
      -RUN

            FILE2
            -INCLUDE FILE3
            -RUN

                  FILE3
                  -INCLUDE FILE4
                  -RUN

                          FILE4
                          -RUN
```

Files one through four are incorporated into the original procedure. The server views all of the included files as part of the original procedure.

## Reference **Other Uses of the `-INCLUDE` Command**

The command `-INCLUDE` is also be used to:

- Control the server environment. For example, the called procedure may set some switches before the calling procedure continues execution.
- Shorten the code when there are several possible procedures that may be called. For example, the command `-INCLUDE &NEWLINES` could be used to determine the called procedure, reducing the number of `GOTO` commands (`&NEWLINES` is a variable whose substitutable value is a file name).

## The `EXEC` Command

A procedure also calls another one with the command `EXEC`. The called procedure must be fully executable.

See *Supplying Values for Variables* on page 4-19 for a description of the syntax.

**Example Using the DATERPT Command**

In the following example, a procedure calls DATERPT:

```
-IF &OPTION EQ 'S' GOTO PRODSALES ELSE GOTO PRODRETURNS;  
.  
.  
.  
-PRODRETURNS  
  EX DATERPT  
.  
.  
.  
-RUN
```

---

## The -REMOTE Commands

A procedure uses -REMOTE BEGIN and -REMOTE END to delimit commands that are sent from an originating server and executed on a target server.

**Syntax How to Use the -REMOTE Commands**

```
-REMOTE BEGIN  
commands  
-REMOTE END
```

where:

*commands*

Are a set of command lines to be processed by Dialogue Manager and then sent to the target server for execution.

**Note:** The following conditions apply when using the -REMOTE commands:



- Another -REMOTE command cannot be included within the -REMOTE BEGIN and -REMOTE END delimiters; that is, -REMOTE commands cannot be nested.
- Dialogue Manager commands within the delimiters are executed, and variable substitution takes place before the stack is sent to the target server. An -INCLUDE command takes a Dialogue Manager procedure residing on the originating server and includes the procedure commands in the stack, as in normal procedure processing.
- The resulting stack of server commands must be a complete server request. Any command that is valid on the target server is included in the stack.
- The command EXEC may be included within the delimiters:

```
-REMOTE BEGIN
EXEC SPNAME &PARM1, &PARM2
-REMOTE END
```

The second command line above is processed by Dialogue Manager (which substitutes real parameters for the amper variables), and sent to the target server. Therefore, the Dialogue Manager procedure (SPNAME) must exist on the target server.

## Reading From and Writing to an External File

---

Dialogue Manager reads information from an external file and write information to it. This section describes the command -WRITE. For information on -READ, see *Supplying Values for Variables* on page 4-19.

### **Syntax** How to Use the -WRITE Command

```
-WRITE filename [NOCLOSE] text
```

where:

*filename*

Is the name of the file being written to. For more information, see Chapter 6, *Platform-specific Commands*.

NOCLOSE

Keeps the external file open until the -WRITE operation is complete. A file kept open with NOCLOSE is closed using the command:

```
-CLOSE filename
```

**Note:** The -RUN command does not close an external file if NOCLOSE is specified.



*text*

Is any combination of variables and text. If the command continues over several lines, put a comma at the end of the line and a hyphen at the beginning of each succeeding line.

### **Example** Using the -WRITE Command

The file name parameter is a symbolic name for a physical file known by the server operating system. Each operating system has its own method for associating a symbolic name with a physical file.

For instance, under MVS, the DYNAM command is used:

```
DYNAM ALLOC FILE MYFILE DSNAME EDAUSER.MYFILE.DATA SHR
```

On UNIX, Windows, OpenVMS, and OS/400 platforms, the FILEDEF command is used:

```
FILEDEF MYFILE DISK /home/edauser/myfile.dat
```

where the data file portion would use the actual native file name convention of the platform.

This example sets the physical name EDAUSER.MYFILE.DATA to the symbolic name MYFILE. Consequently, you could issue the following command:

```
-WRITE CAR &ALINE
```

## **The .EVAL Operator**

---

The .EVAL operator enables you to change a procedure dynamically.

### **Syntax** How to Use the .EVAL Operator

```
[&]&variable.EVAL
```

where:

*variable*

Is either local (&) or global (&&).

Consider the following example:

```
-SET &A = '-TYPE';  
&A HELLO
```

The resulting stack for the above is:

```
-TYPE HELLO
```

which generates an error, because it is a Dialogue Manager command, not a server command.

Adding the .EVAL operator to the preceding example enables the server to interpret the ampersand variable correctly and generate the expected result:

```
-SET &A = '-TYPE';
&A.EVAL HELLO
```

The output with the .EVAL operator is:

```
HELLO
```

The .EVAL operator is typically used in:

- Record selection tests.
- Calculations.

In the following example, the .EVAL operator is used in a record selection test. It forces early substitution of the value for &R (that is, before parsing of the SQL code).

```
-SET &R = 'WHERE COUNTRY = ' || ''ENGLAND''';
-IF &OPTION EQ 'YES' GOTO START;
-SET &R = '-*';
-START
SELECT COUNTRY
FROM CAR
&R.EVAL
;
TABLE
ON TABLE HOLD
END
```

The next example illustrates the use of the .EVAL operator to perform a calculation. It forces early substitution of the value for &OPER, converting the -SET command to a calculation.

```
-SET &A = &OPERANDA &OPER.EVAL &OPERANDB;
-TYPE &OPERANDA &OPER &OPERANDB IS &A
```

## Creating Expressions

---

Dialogue Manager reads information from an external file and write information to it. This section describes the command -WRITE. For information on -READ, see *Supplying Values for Variables* on page 4-19.

An expression consists of variables and literals (numeric or alphanumeric constants) that are combined arithmetically, logically, or in some other way to create a new value.

This section describes how to create:

- Arithmetic expressions.
- Alphanumeric expressions.

- Logical expressions.
- Compound expressions.

Dialogue Manager has few restrictions on creating expressions. However, keep in mind that an expression cannot exceed 40 lines or 16 -IF...THEN...ELSE commands.

## Arithmetic Expressions

An arithmetic expression is:

- A numeric constant, for example, 1.
- Two variables joined by one of the following arithmetic operators:

- + Addition
- Subtraction
- \* Multiplication
- / Division
- \*\* Exponentiation

An example is:

```
&DELIVER_AMT / &OPENING_AMT
```

- Two or more arithmetic expressions, joined by one of the operators in the preceding list. An example is:  

```
(&RATIO - 1) ** 2
```
- A compound expression or function that gives an arithmetic result. See *Using Functions* on page 4-47 for more information.

### **Example** Using Arithmetic Expressions

Following are three arithmetic expressions used in the command -SET:

```
-SET &COUNT = 1;  
-SET &NEWVAL = (&RATIO - 1) ** 2;  
-SET &RATIO = (&DELIVER_AMT * 100) / (&OPENING_AMT);
```

### **Reference** Guidelines for Using Arithmetic Expressions

Keep the following in mind as you create arithmetic expressions:

- If you attempt to divide by 0, Dialogue Manager sets the result to 0.
- Arithmetic operations are performed before logical operations, in the following order:

- \*\* Exponentiation
  - / \* Division and multiplication
  - + - Addition and subtraction
- For operations on the same level (for example, division and multiplication), the evaluation is performed from left to right.
  - An expression in parentheses is evaluated before any other expression.
  - Values for local and global variables (amper variables) are stored internally as character strings, including numeric values. If a calculation is performed on an amper variable, the variable is first converted from a character string into a numeric. After the whole result is calculated, the result of arithmetic operations with numeric fields is truncated to the integer field. Finally, the result is converted back into a character string.

## Alphanumeric Expressions

An alphanumeric expression is:

- A literal enclosed in single quotation marks, for example 'Smith John'.
- A logical expression that yields an alphanumeric result.
- A function that yields an alphanumeric result.
- Two or more alphanumeric variables or literals combined into a single string. See *How to Concatenate Alphanumeric Variables and Literals* on page 4-43 for the syntax and an example.

### Syntax

#### How to Concatenate Alphanumeric Variables and Literals

```
variablename = {alphaexp1 | 'literal'} concatenation {alphaexp2 | 'literal'}
[...]
```

where:

*variablename*

Is the name of the variable assigned to the result of the concatenation.

*alphaexp*

Is a local or global variable that forms part of the concatenation.

*literal*

Is a literal that forms part of the concatenation. It must be enclosed in single quotation marks.

*concatenation*

Is one of the following symbols:

|| indicates strong concatenation, which suppresses trailing blanks.

| indicates weak concatenation, which preserves individual field lengths, including trailing blanks.

**Example Concatenating Alphanumeric Variables and Literals**

```
-SET &NAME = &LASTNAME || ' ' || &FIRST_INIT;
```

If &LASTNAME is equal to DOE and &FIRST\_INIT is equal to J, &NAME is set to:

Doe, J

**Syntax How to Use Date Fields**

System-supplied date functions enable you to calculate the number of days between start and end dates, including leap years. The date format must be either alphanumeric or integer.

*datefield (begin, end)*

where:

*datefield*

Is one of the following:

**YMD** is the number of days between two dates stored as year-month-day (for example, 850522).

**MDY** is the number of days between two dates stored as month-day-year (for example, 052285).

**DMY** is the number of days between two dates stored as day-month-year (for example, 220585).

*begin*

Is the start date.

*end*

Is the end date.

In the following example, &LOSRV is set to the number of days between &HIRE\_DATE and the literal 850101:

```
-SET &LOSRV = YMD(&HIRE_DATE, 850101);
```

**Logical Expressions**

A logical expression contains logical and relational operators and is evaluated to a value that is true or false.

**Example Forming a Logical Expression**

This example shows various elements that are used to form a logical expression. The abbreviation exp stands for expression.

```
{arithmetic exp|alphanumeric exp} operator1 {numeric lit|alphanumeric lit} OR...
```

```
expression operator2 expression
```

```
logical exp {AND|OR} logical exp
```

```
NOT logical exp
```

where:

*operator1*

Is one of the following: EQ, NE, OMITS, or CONTAINS.

*expression*

Is either an arithmetic, alphanumeric, or logical expression.

*operator2*

Is one of the following: EQ, NE, LE, LT, GE, or GT.

The following table defines valid operators (EQ, NE, and so on) used in this example.

<b>Operator</b>	<b>Description</b>
EQ	Tests for a value equal to another value.
NE	Tests for a value not equal to another value.
OMITS	Tests for a value that does not contain a matching character string.
CONTAINS	Tests for a value that does contain a matching character string.
LE	Tests for a value less than or equal to another value.
LT	Tests for a value less than another value.
GE	Tests for a value greater than or equal to another value.
GT	Tests for a value greater than another value.
AND	Returns a value of true if both of its operands are true.
OR	Returns a value of true if either of its operands is true.
NOT	Returns a value of true if the operand is false.

## Reference Guidelines for Alphanumeric and Logical Expressions

Keep the following in mind:

- An alphanumeric literal with embedded blanks or commas must be enclosed in single quotation marks. For example:

```
-IF &NAME EQ 'JOHN DOE' GOTO QUIT;
```

To produce a single quotation mark within a literal, place two single quotation marks where you want one to appear:

```
-IF &NAME EQ 'JOHN O''HARA' GOTO QUIT;
```

- A computational field may be assigned a value by equating it to a logical expression. If the expression is true, the field has a value of 1; if the expression is false, the field has a value of 0.
- Use OR to connect literals or other expressions. You must also use parentheses to separate expressions connected with OR.
- Logical operations are done after arithmetic operations, in the following order:

```
EQ NE LE LT GE GT NOT CONTAINS OMITs
```

```
AND
```

```
OR
```

- Separate a collection of test values with OR:

```
-IF &STATE EQ 'NY' OR 'NJ' OR 'WA' GOTO QUIT;
```

In this case, OR and EQ are evaluated at the same level.

- Use parentheses to specify a desired order. An expression in parentheses is evaluated before any other expression. For example, the command

```
-IF &STATE EQ 'NY' AND &COUNTRY EQ 'US' OR 'UK' THEN...
```

is evaluated as:

```
IF &STATE EQ 'NY' IF &COUNTRY EQ 'US'...
```

Dialogue Manager then evaluates the phrase OR UK and indicates that it is a syntax error.

To write the command correctly, add parentheses:

```
-IF ((&STATE EQ 'NY') AND (&COUNTRY EQ 'US' OR 'UK')) THEN...
```

## Compound Expressions

A compound expression has the following form:

```
-IF expression THEN expression ELSE expression;
```

The following restrictions apply:

- Each of the expressions may itself be a compound expression, although the expression following -IF may not be an -IF...THEN...ELSE expression (for example, -IF...-IF...).
- If the expression following THEN is itself a compound expression, it must be enclosed in parentheses; this rule does not apply to an expression following ELSE.
- Compound expressions only have up to 16 -IF commands.

### Example Using Compound Expressions

If the following example is executed without an input parameter list, the client application receives the message NONE. If it executes with the parameter BANK='FIRST NATIONAL', the client application receives the message FIRST NATIONAL.

```
-DEFAULTS &BANK = ' '
-SET &BANK = IF &BANK EQ ' ' THEN 'NONE'
-ELSE &BANK;
-TYPE &BANK
```

The next example uses a compound expression to define a truth condition (1 is true and 0 is false).

```
-DEFAULTS &CURR_SAL = 900,&DEPARTMENT=MIS
-SET &MYTEST = (&CURR_SAL GE 1000) OR (&DEPARTMENT EQ MIS);
-IF &MYTEST EQ 1 THEN GOTO YES ELSE GOTO NO;
-YES
-TYPE YES
-EXIT
-NO
-TYPE NO
```

When this code is executed, the client application receives the message YES.

## Using Functions

This section describes system-supplied functions that is used in expressions. Write your own functions to solve specific application problems.

### System-supplied Function Examples

The arguments for the following functions are amper variables, expressions, or other functions.

Function	Description
ABS	Returns the absolute value of a number. Computes on one argument: <pre>-SET &amp;PRICE = (ABS (&amp;AMOUNT-&amp;OLDAMOUNT)) / 100;</pre>

Function	Description
INT	Returns the integer part of a number. Computes on one argument: -SET &YEAR = INT(&DATE/10000);
MAX	Returns the maximum value. Computes on one or more arguments, each separated by a comma: -SET &LARGE = IF &FACTOR GT 10 THEN MAX(10, &AMOUNT) - ELSE MAX(0, &AMOUNT, &VALUE/10);
MIN	Returns the minimum value. Computes on one or more arguments, each separated by a comma: -SET &LOW = MIN(0, &AMOUNT, &NEWAMOUNT, &OTHER);
LOG	Returns the logarithm of a number, base e. Computes on one argument: -SET &VAL = 100*&AMOUNT*LOG(&PRICE);
SQRT	Returns the square root of a number. Computes on one argument: -SET &VALUE = 100*&AMOUNT/SQRT(&TOTA);

## System-supplied Function Table

This table summarizes additional system-supplied functions and arguments. Contact your iWay representative to request full documentation on these functions.

Function	Arguments	Description
ABS	<i>value</i>	Returns the absolute value of a number.
ARGLEN	<i>inlength, infield, outfield</i>	Calculates non-blank length of an alphanumeric field.
ASIS	<i>value</i>	Distinguishes between a blank and zero.
ATODBL	<i>number, inlength, outfield</i>	Converts an alphanumeric field containing numeric data to a double-precision decimal field.
AYM	<i>indate, months, outfield</i>	Adds or subtracts a number of months from a given date.

<b>Function</b>	<b>Arguments</b>	<b>Description</b>
AYMD	<i>indate, days, outfield</i>	Adds or subtracts a number of days from a given date.
BAR	<i>barlength, infield, maxvalue, char, outfield</i>	Includes a bar graph in a tabular report. Available for MVS and VM only.
BITSON	<i>bitnumber, infield, outfield</i>	Interprets multi-punch data (data that cannot be represented alphanumerically).
BITVAL	<i>infield, startbit, number, outfield</i>	Obtains decimal value of string of bits.
BYTVAL	<i>character, outfield</i>	Obtains decimal equivalent of alphanumeric character.
CHGDAT	<i>oldformat, newformat, indate, outfield</i>	Changes the format of a date.
CHKFMT	<i>numchar, infield, mask, outfield</i>	Checks character strings for incorrect character types.
CHKPCK	<i>inlength, infield, error, outfield</i>	Validates packed field format.
CNTCTUSR	<i>outfield</i>	Indicates the connected user.
CTRAN	<i>inlength, infield, incode, outcode, outfield</i>	Substitutes characters in a string.
CTRFLD	<i>infield, inlength, outfield</i>	Centers character strings within fields.
DADMY	<i>indate, outfield</i>	Calculates number of days from 1/1/00 with input in day-month-year format.
DADYM	<i>indate, outfield</i>	Calculates number of days from 1/1/00 with input in day-year-month format.
DAMDY	<i>indate, outfield</i>	Calculates number of days from 1/1/00 with input in month-day-year format.

<b>Function</b>	<b>Arguments</b>	<b>Description</b>
DAMYD	<i>indate, outfield</i>	Calculates number of days from 1/1/00 with input in month-year-day format.
DAYDM	<i>indate, outfield</i>	Calculates number of days from 1/1/00 with input in year-day-month format.
DAYMD	<i>indate, outfield</i>	Calculates number of days from 1/1/00 with input in year-month-day format.
DATEADD	<i>YYMDdate, unit, #units</i>	Adds or subtracts a unit to or from a date format.
DATECVT	<i>indate, infmt, outfmt</i>	Converts date formats within applications without requiring intermediate calculations.
DATEDIF	<i>fromYYMD, toYYMD, unit</i>	Returns the difference between two dates in units.
DATEMOV	<i>YYMDdate, move-point</i>	Moves a date to a significant point on the calendar.
DMOD	<i>dividend, divisor, outfield</i>	Calculates the remainder from a division operation and returns a double-precision value.
DMY	<i>begin, end</i>	Calculates the difference between two dates in integer, alphanumeric, or packed format.
DOWK	<i>indate, outfield</i>	Provides the day of the week (in 4-character alphanumeric format) based on input date.
DOWKL	<i>indate, outfield</i>	Provides the day of the week (in 12-character alphanumeric format) based on input date.
DTDMY	<i>number, outfield</i>	Calculates the date from the number of days since 1/1/00 in day-month-year format.

<b>Function</b>	<b>Arguments</b>	<b>Description</b>
DTDYM	<i>number, outfield</i>	Calculates the date from the number of days since 1/1/00 in day-year-month format.
DTMDY	<i>number, outfield</i>	Calculates the date from the number of days since 1/1/00 in month-day-year format.
DTMYD	<i>number, outfield</i>	Calculates the date from the number of days since 1/1/00 in month-year-day format.
DTYDM	<i>number, outfield</i>	Calculates the date from the number of days since 1/1/00 in year-day-month format.
DTYMD	<i>number, outfield</i>	Calculates the date from the number of days since 1/1/00 in year-month-day format.
EXP	<i>power, outfield</i>	Raises "e" to a given power.
EXPN	<i>n.nn {E D} {+ -} p</i>	Evaluates an argument expressed in scientific notation.
FEXERR	<i>nnnnn, A72</i>	Retrieves an error message.
FGETENV	<i>envnamelen, envname, outfieldlen, outfldformat</i>	Retrieves the value of an environment variable and returns it as an alphanumeric string.
FINDMEM	<i>dname, member, outfield</i>	Determines whether a partitioned data set contains a specified member. Available for MVS only.
FMOD	<i>dividend, divisor, outfield</i>	Calculates the remainder from a division operation and returns a single-precision value.
FORECAST	<i>fld2, interval, npredict, method</i>	Uncovers trends in numeric data. Methods are: Simple Moving Average (MOVAVE), Exponential Moving Average (EXPAVE), and Linear Regression Analysis (REGRESS).

<b>Function</b>	<b>Arguments</b>	<b>Description</b>
FPUTENV	<i>namelength, name, valuelength, value, outfield</i>	Assigns a character string to an environment variable.
FTOA	<i>number, usage, outfield</i>	Converts a numeric field to alphanumeric format without inserting leading zeros.
GETPDS	<i>ddname, member, outfield</i>	Determines whether a specific member of a partitioned data set (PDS) exists and returns the PDS name.
GETSECID	<i>outfield</i>	Retrieves the security ID. Available for MVS only.
GETTOK	<i>infield, inlen, toknum, delim, outlen, outfield</i>	Extracts a token from a data string.
GETUSER	<i>outfield</i>	Retrieves the user ID from the system. Use CNTCTUSR for pooled environment.
GREGDT	<i>indate, outfield</i>	Converts a Julian date to a Gregorian date.
HADD	<i>dtfield, component, increment, length, Hformat</i>	Specifies a date-time field by a given number of units.
HCNVRT	<i>dtfield, Hfmt, rlength, Ann</i>	Converts a date-time field to alphanumeric format for use with operators such as EDIT, CONTAINS, and LIKE.
HDATE	<i>dtfield, dateformat</i>	Extracts the date portion of a date-time field and converts it to a date format.
HDIFF	<i>dtfield1, dtfield2, component, Dformat</i>	Finds the number of boundaries of a given type crossed going from date 2 to date 1.
HDTTM	<i>datefield, length, Hformat</i>	Converts a date field to a date-time field.

<b>Function</b>	<b>Arguments</b>	<b>Description</b>
HEXBYT	<i>number, outfield</i>	Obtains the character equivalent of a numeric value.
HGETC	<i>length, Hformat</i>	Stores the current date and time in a date-time field.
HHMMSS	<i>outfield</i>	Retrieves the current time from the system.
HINPUT	<i>inputlength, inputstring, length, Hfmt</i>	Converts an alphanumeric string to a date-time value.
HMIDNT	<i>dtfield, length, Hformat</i>	Changes the time portion of a date-time field to midnight.
HNAME	<i>dtfield, component, Aformat</i>	Extracts a specified component from a date-time field and returns it in alphanumeric format.
HPART	<i>dtfield, component, Iformat</i>	Extracts a specified component from a date-time field and returns it in numeric format.
HSETPT	<i>dtfield, component, value, length, Hformat</i>	Inserts the numeric value of a specified component into a date-time field.
HTIME	<i>length, dtfield, Dformat</i>	Converts the time portion of a date-time field to a numeric number of milliseconds or microseconds.
IMOD	<i>dividend, divisor, outfield</i>	Calculates the remainder from a division operation and returns an integer value.
INT	<i>value</i>	Returns the integer part of an argument.
ITONUM	<i>sigbytes, infield, outfield</i>	Converts large binary integers in non-FOCUS files to double-precision format.

<b>Function</b>	<b>Arguments</b>	<b>Description</b>
ITOPACK	<i>sigbytes, infield, outfield</i>	Converts large binary integers in non-Dialogue Manager files to packed format.
ITOZ	<i>outlength, number, outfield</i>	Converts a numeric field to a zoned decimal.
JULDAT	<i>indate, outfield</i>	Converts a Gregorian date to a Julian date.
LCWORD	<i>length, instring, outstring</i>	Translates uppercase characters in alphanumeric fields to lowercase on a word-by-word basis.
LJUST	<i>inlength, infield, outfield</i>	Left-justifies a character string within a field.
LOCASE	<i>length, infield, outfield</i>	Translates uppercase characters in alphanumeric fields to lowercase characters.
LOG	<i>value</i>	Returns the logarithm of a number, base e.
MAX	<i>value1, value2...</i>	Returns the maximum value.
MDY	<i>begin, end</i>	Calculates the difference between two dates in integer, alphanumeric, or packed format.
MIN	<i>value1, value2...</i>	Returns the maximum value.
MVS DYNAM	<i>command, length, rc</i>	Transfers a specified FOCUS DYNAM command to the DYNAM command processor.
OVRLAY	<i>base, baselen, substring, sublen, position, outfield</i>	Overlays a character string on a character string.
PARAG	<i>inlength, infield, delimiter, subsize, outfield</i>	Inserts a delimiter into a character string.
PCKOUT	<i>infield, outlength, outfield</i>	Outputs a packed field.

<b>Function</b>	<b>Arguments</b>	<b>Description</b>
POSIT	<i>parent, inlength, substring, sublength, outfield</i>	Finds the position of a character string in another string.
PRDNOR	<i>seed, outfield</i>	Generates repeatable random numbers for normal distribution.
PRDUNI	<i>seed, outfield</i>	Generates repeatable random numbers for uniform distribution.
RDNORM	<i>outfield</i>	Generates random numbers for normal distribution.
RDUNIF	<i>outfield</i>	Generates random numbers for uniform distribution.
REVERSE	<i>length, input, output</i>	Reverses the characters that were input.
RJUST	<i>inlength, infield, outfield</i>	Right-justifies an alphanumeric field.
SOUNDEX	<i>inlength, string, outfield</i>	Searches for character strings phonetically.
SPELLNUM	<i>outlength, infield, outfield</i>	Takes an alphanumeric string or a numeric value with two decimal places and spells it out with dollars and cents.
SQRT	<i>value</i>	Returns the square root of a number.
STRIP	<i>length, source_string, strip_char, result</i>	Removes all occurrences of a specific character from an input string.
SUBSTR	<i>inlength, parent, start, end, outlength, outfield</i>	Extracts a substring.
TEMPPATH	<i>outlength, outfield</i>	Retrieves the physical directory name of the current agent process.
TODAY	<i>outfield</i>	Retrieves the current date from the system.

Function	Arguments	Description
TRIM	<i>location, string, string_length, pattern pattern_length, , result</i>	Removes leading and/or trailing occurrences of a pattern within a string. The location parm indicating where to trim the pattern is L (leading), T (trailing), or B (both leading and trailing).
TRUNCATE	<i>var1</i>	Removes trailing blanks from Dialogue Manager amper variables and adjusts the length accordingly.
UFMT	<i>infield, inlength, outfield,</i>	Converts characters in alphanumeric fields to HEX representation.
UPCASE	<i>length, infield, outfield</i>	Translates lowercase characters in alphanumeric fields to uppercase.
YM	<i>fromdate, todate, outfield</i>	Returns the number of months between two dates.
YMD	<i>begin, end</i>	Calculates the difference between two dates in integer, alphanumeric, or packed format.

In most cases, the *outfield* may be expressed as a format such as 'A10'.

Custom routines may be written in a 3GL and added to the servers search path provided they are:

- Compiled properly and placed in the server's user directory of EDACONF.
- or
- Located on the IBICPG path and compiled as a shared library with one routine for each library with the same name.

The script, `gencpgm`, is provided to assist in the actual compilation of a program on some platforms, but any method is allowed provided that it links to the appropriate library and files. For more information, see Appendix B, *GENCPGM Usage*.

On MVS, build routines into a loadlib and allocate as `ALLOCATE F(IBICPG) DA('USER.LIBRARY.LOAD') SHR`, unless they are REXX based (see MVS REXX below).

On VM, build routines into a loadlib and execute `GENSUBLL EXEC` to generate the new loadlib, unless they are REXX based (see VM REXX below).

On MVS and VM, routines may also be written in REXX. On MVS, REXX routines must be stored in a PDS with a FUSREXX ddname allocated to the PDS. On VM, each routine must be in a file with a file type of FUSREXX on an accessible disk. On VM, compiled REXX is also supported and uses the same file type of FUSREXX.

## Editing a Value

The mask option of the EDIT function inserts characters in an alphanumeric value, or extracts certain characters from the value.

### **Syntax** How to Use the EDIT Function

```
EDIT(fieldname, 'mask');
```

where:

*fieldname*

Is the name of the field to be edited.

'*mask*'

Is a value that the field name matches against, enclosed in single quotation marks. If *mask* contains the number 9, the corresponding character in *fieldname* is moved to the new field. If *mask* contains a dollar sign (\$), the corresponding character in *fieldname* is ignored. If a character in *mask* is neither the number 9 nor a dollar sign, the character is inserted in the new field.

### **Example** Using the EDIT Function

In the following example, assume that &EMP\_ID is a 9-character alphanumeric field and &FIRST\_NAME is a 10-character alphanumeric field. Suppose you want to edit &EMP\_ID by inserting hyphens, then display the first initial and last name of an employee.

```
-SET &EMPIDEDIT = EDIT(&EMP_ID, '999-99-9999');
-SET &FIRST_INIT = EDIT(&FIRST_NAME, '9$');
-TYPE &EMPIDEDIT &FIRST_INIT &LAST_NAME
```

Assume that &EMP\_ID is 516888704 and &FIRST\_NAME is 'EDWARD'. The client application receives:

```
516-88-8704 E Jones
```

## Decoding a Value

Many times the value of a field is coded. For example, the field &SEX may contain code F for female employees and code M for male employees, reducing the storage requirement for the value.

One method for decoding (expanding) these values is to include a series of nested -IF...THEN...ELSE commands (for example, -IF &SEX IS 'M' THEN 'MALE' ELSE 'FEMALE'), but this becomes very cumbersome. As an alternative, Dialogue Manager provides the DECODE function.

### **Syntax**    **How to Decode a Value**

```
DECODE fieldname (code1 result1 code2 result2...[ELSE default]);
```

where:

*fieldname*

Is an alphanumeric or numeric field to be decoded.

*code*

Is the code to be expanded.

*result*

Is the expanded value to be substituted for code. If this value has embedded blanks or commas, or if it is a negative number, enclose it in single quotation marks.

Use either commas or blanks to separate the code from the result, or one pair from another pair.

*default*

Is the value to be assigned if the code is not found. If you do not supply a default, Dialogue Manager assigns a blank or zero.

Code up to 40 lines of pairs of elements (a pair is a code and a result), or 39 if you include an ELSE.

### **Example**    **Using the DECODE Function**

In the following example, values (results) are substituted for the codes FED, STAT, CITY, FICA, HLTH, and SAVE:

```
-SET &DEDUCTION = DECODE &DED_CODE(FED 'TAXES' STAT 'TAXES'  
- CITY 'TAXES' FICA 'FICA' HLTH 'INSURANCE' SAVE 'PERSONAL'  
- ELSE 'OTHER');  
-TYPE &DEDUCTION
```

### **Syntax**    **How to Store Codes and Results in a Separate File**

```
DECODE &testvar (filename [ELSE default]);
```

where:

*filename*

Is the symbolic name of a physical file.

Each record in the file must contain one pair of elements (a code and a result), separated by a comma or blanks. For example:

```
F FEMALE
M MALE
```

DECODE tests each record in *filename*; if the value of *&testvar* matches a value in the first column of *filename*, DECODE returns the value in the second column. For example, if the above file is named GENDER, the following results in MALE:

```
-SET &SEX = M;
-SET &SEX = DECODE &SEX(GENDER);
-TYPE &SEX
```

- If an element itself contains a comma or a blank, enclose it in single quotation marks.
- Leading and trailing blanks are ignored.
- Include up to 31,000 characters in the file.
- If a record contains only one element (with the remainder of the record entirely blank), the element is interpreted as the code. The result defaults to either blank or zero, as needed.

In the following example, &TAKE is set to 0 for &SELECT values found in *filename*, and is set to 1 in all other cases:

```
&TAKE = DECODE &SELECT (filename ELSE 1);
&VALUE = IF &TAKE IS 0 THEN...ELSE...;
```

## Using Commands Specific to an Operating System

---

A Dialogue Manager procedure executes commands that are specific to an operating system. Operating systems include OS/400, VM, MVS/TSO, Windows, UNIX, and OpenVMS.

### **Syntax** How to Execute Commands With a Dialogue Manager

```
[operating system] command
```

where:

## ON TABLE HOLD

*[operating system]*

Specifies the operating system. Possible values are:

-AS/400, AS/400 or CMD specifies the OS/400 operating system.

-CMS or CMS specifies the CMS operating system.

-DOS or DOS specifies the DOS operating system.

-TSO RUN or TSO RUN specifies the MVS/TSO operating system.

-UNIX or UNIX specifies the UNIX operating system.

-VMS or VMS specifies the VMS operating system.

-WINNT or WINNT specifies the Windows NT operating system.

! for a generic request to specify a non-specific operating system.

*command*

Is an operating system command.

Specifications starting with a dash (-) are executed in the normal flow of Dialogue Manager commands. Commands that do not start with a dash (-) are stacked until execution is forced by an end of file or a -RUN. Note that there is no -!.

## ON TABLE HOLD

---

When a server receives the results of an SQL request (an answer set) from another server, the answer set will either:

- Be returned to the client application using ON TABLE PCHOLD. That command is described in *ON TABLE PCHOLD* on page 4-61.
- Be held on the initiating server, without sending it back to the client application, using ON TABLE HOLD. A corresponding Master File for the file that holds the answer set is also created.

### **Syntax**

### **How to Use the ON TABLE HOLD Command**

```
ON TABLE HOLD [AS filename] FORMAT format  
END
```

where:

*filename*

Is the name of the file that holds the answer set. If *filename* is omitted, the name of the held file on the server is HOLD, and subsequent creations of HOLD files overlay each other. The file name is a symbolic name known to the operating system for the server environment.

*format*

Is one of the format options valid for the server. Possible values are: ALPHA, BINARY, COMMA, DBASE, DB2, DIF, DOC (WebFOCUS ONLY), EXCEL, EXL2K (WebFOCUS ONLY), EXL2K PIVOT (WebFOCUS ONLY), FOCUS, FUSION, HTML, HTMTABLE, INGRES, INTERNAL, LOTUS, PDF, POSTSCRIPT, REDBRICK, SQL, SQLDBC, SQLINF, SQLMAC, SQLMSS, SQLODBC, SQLORA, SQLSYB, SYLK, TABT, WK1, WP.

END

Is required on a separate line.

## ON TABLE PCHOLD

---

In order for a Dialogue Manager procedure to return an answer set to a client application, a certain set of commands must be issued directly after the SQL request in the syntax of the procedure.

### **Syntax**

#### **How to Use the ON TABLE PCHOLD Command**

```
SQL
SQL request;
TABLE
ON TABLE PCHOLD [FORMAT ALPHA]
END
```

where:

*SQL request*

Is a valid SQL request, ending with a semicolon.

FORMAT ALPHA

Optionally specifies that the hold file on the client is a text file. Use any valid format available on the client, but the underlying transfer is in alpha format. FORMAT ALPHA is the default.

END

Is required on a separate line.

### **Example**

#### **Using the ON TABLE PCHOLD Command**

This example shows how the ON TABLE PCHOLD command requests information from a table in a catalog.

```
SQL
SELECT NAME, CREATOR, COLCOUNT, RECLENGTH FROM SYSTABLE
TABLE
ON TABLE PCHOLD FORMAT ALPHA
END
```

The result of the request is an answer set sent to the client application by the server.

*ON TABLE PCHOLD*

---

---

## CHAPTER 5

# Transaction Adapters for IMS/TM

### Topics:

- Overview of Transaction Adapters
- CALLIMS Adapter for IMS/TM
- CALLITOC OTMA Adapter for IMS/TM
- Transaction Processing With CALLIMS or CALLITOC
- Using CALLIMS and CALLITOC
- Installation of CALLIMS and CALLITOC
- Storing Multiple Messages Into a Server File for Later Queries
- ETPCIMS LU6.2 Adapter for IMS/TM
- Using ETPCIMS LU6.2
- Transaction Processing With ETPCIMS LU6.2
- Executing ETPCIMS
- Installing ETPCIMS
- Server Settings and Environment Variables

These topics contain information about the Transaction Adapters for IMS/Transaction Manager (TM). Overview, configuration, and installation information are included for the following IMS/TM Adapters:

- **CALLIMS LU6.2 Adapter for IMS/TM.** Originally known as the EDA SQL Transaction Server for IMS, this adapter connects to IMS/TM using an IMS/APPC connection from a server for MVS.
- **CALLITOC OTMA Adapter for IMS/TM.** CALLITOC is available on all servers beginning with Version 5 Release 1.0.
- **ETPCIMS LU6.2 Adapter for IMS/TM.** Originally known as the EDA TP Client Interface for IMS/TM, this adapter connects to IMS/TM using an IMS/APPC connection from a non-MVS server.

The CALLIMS, CALLITOC, and ETPCIMS Adapters for IMS/TM hide the complexity of accessing IMS/TM transactions from the client application. The client application uses any major *standard* access protocol to connect to a server and invoke the adapter to access the IMS/TM transaction. The standard protocols include ODBC, JDBC, OLE DB, or any enabled client or connector.

## Overview of Transaction Adapters

---

The Transaction Adapters allow any client or connector to invoke a transaction running under the control of an IMS/TM transaction-processing monitor. It provides a means of building on existing applications that perform transaction processing against IMS/TM, to create new Web or client/server applications while reusing existing IMS transactions and program logic.

Before attempting to create an application using any one of the Transaction Adapters for IMS/TM, you should have available the *API Reference* manual for API method calls and this manual for writing Dialogue Manager procedures.

### CALLIMS Adapter for IMS/TM

---

The CALLIMS Adapter is part of the server for MVS. It is attached to and extend from any compatible API environment, including client applications and Hub Servers.

The CALLIMS Adapter uses LU6 communications from a server for MVS to execute IMS/TM transactions, and passes the output to any client that connects to that server.

### CALLITOC OTMA Adapter for IMS/TM

---

Starting with Version 5 Release 1.0, the OTMA Adapter for IMS will be available for servers for MVS, Windows NT, and UNIX. It is attached to and extend from any compatible API environment, including client applications and Hub Servers.

The OTMA Adapter fully utilizes the IMS TOC functionality. The following is a brief description of IMS TOC, which is equivalent to the IMS Connect for IMS 7.1 and higher.

The IMS TCP/IP Connector (ITOC) and its equivalent product, IMS Connect, is available with IMS Version 7.1 and higher, and allows client TCP/IP communications to and from one or more IMS/TM regions. TOC and IMS Connect are IBM's implementation for allowing the execution of IMS/TM transactions from TCP/IP clients.

## Transaction Processing With CALLIMS or CALLITOC

---

The following steps are performed when a client application calls an IMS/TM transaction using either the CALLIMS or CALLITOC Adapter. The difference between the CALLIMS and the CALLITOC Adapters is the communications method used to connect to the IMS/TM region from the client application. Also, CALLIMS requires a server for MVS to initiate the IMS/TM request. Any client on any platform can connect to the server for MVS.

1. The client application issues the API method call, EDARPC, to run a Dialogue Manager procedure residing on a server (either a Hub Server or a Full-Function Server).

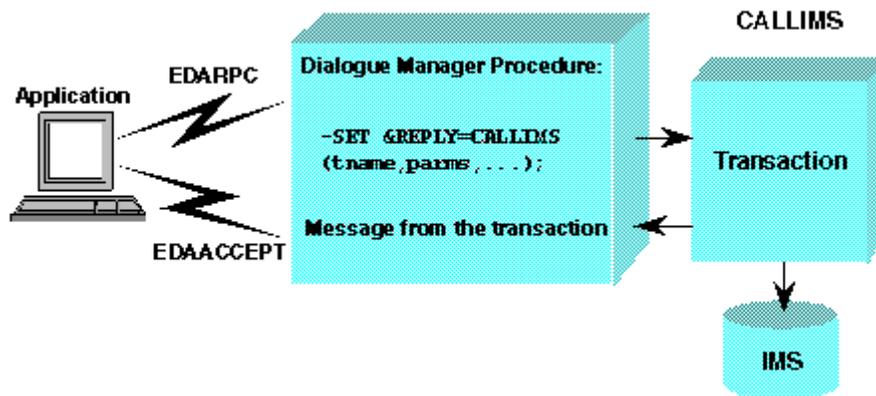
Parameters are sent as part of the calling sequence for use by the procedure.

2. The Dialogue Manager procedure contains the command `-SET &REPLY=CALLIMS`, which executes `CALLIMS` or `CALLITOC`.
3. `CALLIMS` or `CALLITOC` establishes a connection to the IMS/TM environment and invokes the transaction, passing an IMS message as a parameter.
4. The IMS transaction receives and then processes the input message. It then passes one IMS message back to the server. The `CALLIMS` or `CALLITOC` Adapter returns the message to the client application.
5. The client application issues the method call, `EDAACCEPT`, to accept the message.

For details on the syntax and use of API method calls, see the *API Reference* manual.

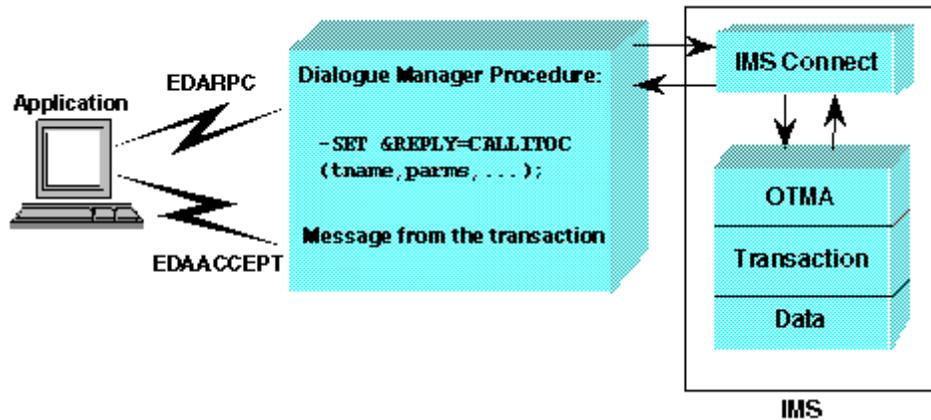
### **Example** Processing a Transaction With `CALLIMS`

The following figure illustrates transaction processing as initiated by the client application running a `CALLIMS` Adapter request.



### **Example** Processing a Transaction With `CALLITOC OTMA`

The following figure illustrates transaction processing as initiated by the client application running a `CALLITOC OTMA` Adapter request.



## Using CALLIMS and CALLITOC

To call an IMS/TM transaction, execute either CALLIMS or CALLITOC from a Dialogue Manager procedure.

- CALLIMS invokes an IMS/TM transaction through the use of APPC/IMS. CALLIMS requires IMS Version 4.1 or higher (IMS/TM) and APPC/MVS.
- CALLITOC invokes an IMS/TM transaction through the use of IMS TOC or IMS Connect. CALLITOC requires IMS Version 5.1 or higher (IMS/TM) and a TOC or IMS connection to an IMS OTMA address space.

A sample RPC called CALLIMS will be supplied within the server for MVS. It executes the IMS/TM PART transaction for verification and installation purposes.

A sample RPC called CALLIMSC will be supplied within all servers in future releases. It also executes the IMS/TM PART transaction for verification and installation purposes.

## Transaction Processing With CALLIMS or CALLITOC

The following steps are performed when a client application calls an IMS/TM transaction using either the CALLIMS or CALLITOC Adapter.

There are several differences between the CALLIMS and the CALLITOC Adapters.

- The communications method used to connect to the IMS/TM region from the client application. CALLIMS uses LU6.2, CALLITOC uses TCP/IP to the mainframe using IMS Connect.
- CALLIMS requires a server for MVS to initiate the IMS/TM request.

- For CALLIMS, you must run a separate link job to allow CALLIMS to communicate with APPC/MVS. See the *Server for OS/390 and z/OS Configuration and Operations* manual for the steps needed to configure the Transaction Server for IMS.

## How Data Is Returned With CALLIMS and CALLITOC

CALLIMS and CALLITOC return codes and data to the client application in a series of 72-byte messages. To retrieve the messages, the client application issues the method call EDAACCEPT.

Repeated calls to EDAACCEPT retrieve all messages from CALLIMS and CALLITOC.

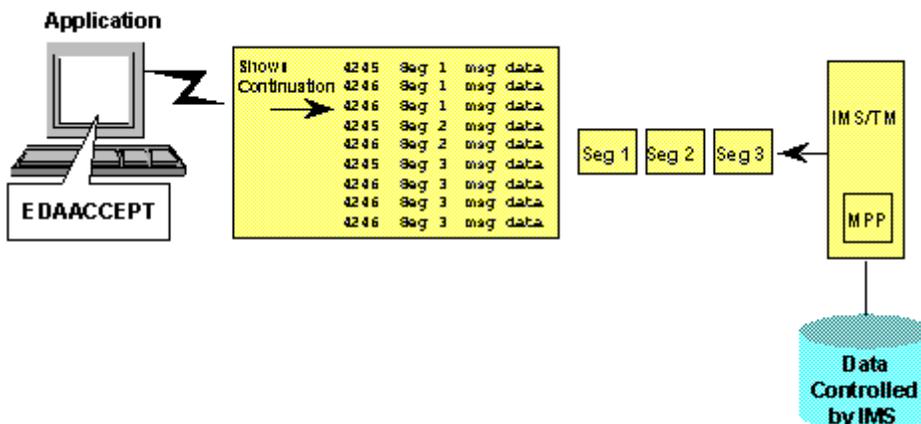
The client application should check the message code in the field `scb.msg_code`. (The session control block contains several fields pertaining to message processing. For example, `scb.msg_code` contains a message code and `scb.msg_text` contains the message text.)

A code of 4245 indicates that the message is coming from CALLIMS or CALLITOC.

For the client application to detect multiple output segments, the Transaction Server for IMS returns a message code of 4246 following the 4245, until the end of the segment. At that time, the message code is again 4245, indicating the beginning of a new segment.

### Example Viewing Message Codes From CALLIMS or CALLITOC

The following figure illustrates the message codes from CALLIMS or CALLITOC when there are multiple output segments. The client application normally changes the received format to a format acceptable to the end user.



## Executing CALLIMS and CALLITOC

CALLIMS or CALLITOC is executed from a Dialogue Manager procedure with the command -SET.

For details on -SET and other Dialogue Manager commands mentioned, see Chapter 4, *Writing a Dialogue Manager Procedure*.

Any of the variables described in the syntax that follows may be implicitly set in the Dialogue Manager procedure using the command -DEFAULTS, or explicitly set on the CALLIMS or CALLITOC call. The variables work the same way, whether set implicitly or explicitly. If both ways are used, the explicit setting takes precedence over the implicit setting. For examples of implicit settings, see *Using -SET to Execute CALLITOC* on page 5-10 and *Using -SET to Execute CALLIMS* on page 5-9.

### **Syntax** How to Execute the CALLIMS Adapter

The dash (-) in the syntax below is used to allow multiple lines for a parameter list in a Dialogue Manager command, as in -SET.

```
-SET &REPLY = CALLIMS ( &SYM, &TP, &MLEN, &MSG, &DELIM, &OPT, &UID,  
- &PW, &SG, &PLU, &LMODE, &HEXCONV, ' A1 ' ) ;
```

where:

*&SYM*

Is the symbolic destination name. This is a key value for the side information data set configuration file defined to APPC/MVS. APPC/MVS supports a symbolic destination name for determining the default APPLID for the IMS/TM region, with a log mode and transaction name. If a value is supplied for &SYM, either implicitly or explicitly, then &TP, &PLU, and &LMODE may be left blank. This field must be eight (8) characters in length.

*&TP*

Is the name of the IMS MPP transaction. This field must be eight (8) characters in length. Set &TP to blanks if you supply a value for &SYM.

*&MLEN*

Is the length of the IMS MPP message for the transaction. This is set with the Dialogue Manager LENGTH function in a -SET command.

*&MSG*

Is the input message for the IMS transaction. This message must exactly match the layout expected by the transaction.

*&DELIM*

Is a non-blank character string used to delimit individual segments in this IMS message. Each delimited segment has its own SEND. This variable is designed for MPP transactions that expect to retrieve multiple segments from the message queue. The length of &DELIM must be four (4) characters.

*&OPT*

Specifies whether CALLIMS waits for a response. Possible values are:

*REPL* assumes synchronous operation. CALLIMS waits for a return code or other response from IMS/TM.

*NORP* sends the message, then immediately returns control to the server. CALLIMS does not wait for a response.

The length of &OPT must be four (4) characters.

The following parameters are optional. If not used, the parameters must be padded with blanks.

*&UID*

Is the user ID that invokes the IMS MPP transaction. The length of &UID must be eight (8) characters.

*&PW*

Is the password used to invoke the IMS MPP transaction. The length of &PW must be eight (8) characters.

*&SG*

Is the security group that the user ID belongs to or is part of. The length of &SG must be eight (8) characters.

*&PLU*

Is the LU6.2 APPLID for the IMS/TM region. The length of &PLU must be eight (8) characters. Set &PLU to blanks if you supply a value for &SYM.

*&LMODE*

Is the log mode table entry name. The length of &LMODE must be eight (8) characters. Set &LMODE to blanks if you supply a value for &SYM.

*HEXCONV*

A value of ON will truncate the message returned after any hex character data. OFF is the default.

*'A1'*

Sets the format of the response to alphanumeric. It is required by -SET.

## **Syntax**      **How to Execute the CALLITOC OTMA Adapter**

The dash (-) in the syntax below is used to allow multiple lines for a parameter list in a Dialogue Manager command, as in -SET.

```
-SET &REPLY = CALLITOC(&HOST, &PORT, &DSID, &TPNAME, &MLEN, &MESSAGE,  
- &USERID, &RUSERID, &RGROUP, &PASSWD, &DELIM, &HEXCONV, &OPTION, 'A1');
```

where:

### *&HOST*

Is the symbolic destination name. TCP/IP is the host address where IMS Connect or ITOC is running. *&host* is the value in the HOSTNAME=*host* in the HWS configuration file.

### *&PORT*

Is the port number that IMS Connect or ITOC is listening on. *&port* is the value in the PORTID=*port* in the HWS configuration file.

### *&DSID*

Is the value in the DATASTORE ID=*dsid* in the HWS configuration file.

### *&TPNAME*

Is the name of the IMS MPP transaction. This field must be eight (8) characters in length.

### *&MLEN*

Is the length of the IMS MPP message for the transaction. This is set with the Dialogue Manager LENGTH function in a -SET command.

### *&MESSAGE*

Is the input message for the IMS transaction. This message must exactly match the layout expected by the transaction.

### *&USERID*

Is the ITOC user ID. This user ID is reserved for future use.

### *&RUSERID*

Is the user ID that is validated by RACF based on the RGROUP value.

### *&RGROUP*

Is the XCF group validated by RACF. *&rgroup* is the value in the GROUP=*rgroup* in the HWS configuration file.

### *&PASSWD*

Is the RACF password based on the RUSERID value.

*&DELIM*

Is a non-blank character string used to delimit individual segments in this IMS message. Each delimited segment has its own SEND. This variable is designed for MPP transactions that expect to retrieve multiple segments from the message queue. The length of &DELIM must be four (4) characters.

*&HEXCONV*

A value of ON will truncate the message returned after any hex character data. OFF is the default.

*&OPTION*

Specifies whether CALLITOC waits for a response. Possible values are:

*REPL* assumes synchronous operation. CALLITOC waits for a return code or other response from IMS/TM.

*NORP* sends the message, then immediately returns control to the server. CALLITOC does not wait for a response.

The length of &OPT must be four (4) characters.

*'A1'*

Sets the format of the response to alphanumeric. It is required by -SET.

**Example Using -SET to Execute CALLIMS**

The following is a sample Dialogue Manager procedure that uses the command -SET to execute CALLIMS. It is member CALLIMS of the data set.

In this example, values for variables are padded with blanks as required. The command -DEFAULTS enables you to specify default values in the Dialogue Manager procedure, but may be overridden by a client application.

```
-DEFAULTS &SYM      = '          '
-DEFAULTS &TP       = 'PART      '
-DEFAULTS &MSG      = 'AN960C10 '
-DEFAULTS &DELIM    = '          '
-DEFAULTS &OPT      = 'REPL     '
-DEFAULTS &UID      = '          '
-DEFAULTS &PW       = '          '
-DEFAULTS &SG       = '          '
-DEFAULTS &PLU      = 'SCMI41AX '
-DEFAULTS &LMODE    = 'LU62APPC '
-SET &MLEN = &MSG.LENGTH;
-SET &REPLY =
CALLIMS (&SYM,&TP,&MLEN,&MSG,&DELIM,&OPT,&UID,&PW,&SG,&PLU,&LMODE,'A1');
```

### Example Using -SET to Execute CALLITOC

The following is a sample Dialogue Manager procedure that uses the command -SET to execute CALLITOC. For a server for MVS, it is member CALLIMS of the data set.

In this example, values for variables are padded with blanks as required.

```
-DEFAULT  &HOST=      'IBIMVS.IBI.COM  ' ;
-DEFAULT  &PORT=      6683 ;
-DEFAULT  &DSID=      'IMS61          ' ;
-DEFAULT  &TPNAME=    'PART           ' ;
-DEFAULT  &OPTION=    'REPL           ' ;
-DEFAULT  &MESSAGE=   'AN960C10      ' ;
-DEFAULT  &USERID=    '                ' ;
-DEFAULT  &RUSERID=   '                ' ;
-DEFAULT  &RGROUP=    'IMSGRP61      ' ;
-DEFAULT  &PASSWD=    '                ' ;
-DEFAULT  &DELIM=     '                ' ;
-DEFAULT  &HEXCONV=   '                ' ;
-SET  &MLEN = &MESSAGE.LENGTH;

-SET  &REPLY = CALLITOC(&HOST,
-                               &PORT,
-                               &DSID,
-                               &TPNAME,
-                               &MLEN,
-                               &MESSAGE,
-                               &USERID,
-                               &RUSERID,
-                               &RGROUP,
-                               &PASSWD,
-                               &DELIM,
-                               &HEXCONV,
-                               &OPTION,
-                               'A1' ) ;
```

## Installation of CALLIMS and CALLITOC

---

Once you have created a Dialogue Manager procedure that includes CALLIMS or CALLITOC for the IMS/TM OTMA Adapter, you must place it in the appropriate libraries for a server for MVS or a directory for a non-MVS server. See Chapter 1, *Introduction*, for details on stored procedure libraries and stored procedure execution order.

**Note:** •  CALLIMS is run from a server for MVS only. Therefore, CALLIMS is also placed in a library allocated to EDARPC on the server. The CALLIMS or CALLITOC load module is placed in a library allocated to STEPLIB on the server. If a function executes CALLITOC from a server for MVS, the procedure must also be placed in a library allocated to ddname EDARPC.

- CALLITOC runs from Windows NT or UNIX as well as MVS. The following is an example of the Windows NT directories that include the CALLITOC Adapter as well as the CALLIMSC RPC function within a Version 5 Release 1.0 server:

```
IBI\svr51\ffs\user\CALLITOC.DLL
```

```
IBI\svr51\ffs\catalog\CALLIMSC.FEX
```

## Installing CALLIMS

CALLIMS is executed as a Dialogue Manager procedure. A Dialogue Manager procedure is referred to as a Remote Procedure Call (RPC) or a FOCEXEC. This procedure must be made available on a server for MVS. It is recommended that you assign the procedure to a library allocated to the ddname EDARPC. A sample procedure is located in the EDARPC.DATA library, member CALLIMS. This procedure is modified and run, or it is used as a base for other procedures.

You must first generate an APPC connection for CALLIMS before it is run. The following JCL is located in member GENEAPPC in the library EDACTL.DATA. It must be modified and run. Make sure that the load module CALLIMS that is created in SYSLMOD is made available to the EDASERVER STEPLIB ahead of any load libraries.

```

/*****
/* NAME:      GENEAPPC JCL
/*
/* FUNCTION: LINKEDIT APPC/MVS STUBS INTO CALLIMS
/*
/* PROC SYMBOLIC PARAMETERS:
/*      1. qualif MUST BE CHANGED TO THE HIGH LEVEL QUALIFIER
/*      USED FOR THE PROGRAM DATASETS UNLOAD FROM THE MEDIA
/*
/*****

```

```
//APPROC PROC PREFIX='qualif'
//*
//LKED EXEC PGM=IEWL, PARM='RENT,LIST,NOXREF,LET'
//SYSLIB DD DISP=SHR,DSN=SYS1.CSSLIB <- change this to your APPC
                                             library that contains member ATBPBI
//SYSIN DD DISP=SHR,DSN=&PREFIX..EDALIB.LOAD <- EDALIB libraries from
                                             the install

//SYSLMOD DD DISP=SHR,DSN=&PREFIX..EDALIB.LOAD
//MAINTAIN DD DISP=SHR,DSN=&PREFIX..EDALIB.DATA
//SYSUT1 DD UNIT=SYSDA,SPACE=(100,(50,50))
//SYSPRINT DD SYSOUT=A
//*
//APPROC PEND
//GENFAPPC EXEC APPROC
//LKED.SYSLIN DD *
INCLUDE SYSIN(CALLIMS)
INCLUDE SYSLIB(ATBPBI)
INCLUDE MAINTAIN(CALLIMS)
NAME CALLIMS(R)
/*
```

See Chapter 1, *Introduction*, for details on stored procedure libraries and stored procedure execution order.

## Installing CALLITOC

CALLITOC is executed as a Dialogue Manager procedure. A Dialogue Manager procedure is referred to as an RPC or a FOCEXEC. This procedure must be made available to a server, but it is not limited to a server for MVS.

To run CALLITOC from a server for MVS, it is recommended that you assign the procedure to a library allocated to the ddname EDARPC. A sample procedure is located in the EDARPC.DATA library, member CALLIMSC. This procedure is modified and run or it is used as a base for other procedures.

CALLITOC runs from Windows NT or UNIX as well as MVS. The following is an example of the Windows NT directories that include the CALLITOC Adapter as well as the CALLIMSC RPC function within a Version 5 Release 1.0 server:

```
IBI\svr51\ffs\user\CALLITOC.DLL
```

```
IBI\svr51\ffs\catalog\CALLIMSC.FEX
```

There are no pre-installation requirements for the CALLITOC Adapter.

See Chapter 1, *Introduction*, for details on stored procedure libraries and stored procedure execution order.

## Storing Multiple Messages Into a Server File for Later Queries

---

If you are using a front-end application, such as PowerBuilder, which cannot handle multiple messages, a method is needed to convert the messages into an answer set. This is especially important when using CALLIMS or CALLITOC, which only return messages.

The following stored procedure executes a CALLIMS procedure that returns five (5) messages. Instead of sending the messages directly to the client, the stored procedure stores the messages in a file. A subsequent SELECT is performed to extract data, as an answer set, from the file. The stored procedure statements required for saving messages in the file are in **bold** letters.

```
-SET &EMGSRV = 'FILE';
DYNAM ALLOC FILE EMGFILE DA qualif.EMGFIL SPACE 2,2 TRACKS UNIT SYSDA -
RECFM FB LRECL 80 BLKSIZE 1600
SET EMGSRV=&EMGSRV
SET PAUSE=OFF
-RUN
-DEFAULT &SYM = ' '
-DEFAULT &TP = 'PART '
-DEFAULT &MSG = 'AN960C10'
-DEFAULT &DELIM= ' '
-DEFAULT &OPT = 'REPL '
-DEFAULT &UID = ' '
-DEFAULT &PW = ' '
-DEFAULT &SG = ' '
-DEFAULT &PLU = 'SCMI41AX'
-DEFAULT &LMODE= 'LU62APPC'
-SET &MLEN = &MSG.LENGTH ;
-SET &REPLY = CALLIMS (&SYM, &TP, &MLEN, &MSG, &DELIM, &OPT, &UID,
- &PW, &SG, &PLU, &LMODE, 'A1' ) ;
DYNAM FREE DDN EMGFILE
-RUN
DYNAM ALLOC FILE EMGSRV1 DA qualif.EMGFIL SHR REU
-RUN
SQL
SELECT COL1 FROM EMGSRV1;
TABLE
ON TABLE PCHOLD FORMAT ALPHA
END
DYNAM FREE DDN EMGSRV1
-RUN
```

For the SELECT COL1 FROM EMGSRV1 statement to work, a Master File has to be predefined for the file. The following is the Master File, EMGSRV1, used for this example:

```
FILENAME=EMGSRV1 , SUFFIX=FIX  
SEGNAME=ORIGIN , SEGTYPE=S1  
FIELDNAME=COL1 , FIRST40 , A40 , $  
FIELDNAME=COL2 , LAST40 , A40 , $
```

The following is the output, from CALLIMS, which is placed into the file:

```
(FOC4245) :      Part.....      AN960C10; Desc..... WASHER  
(FOC4246) :  
(FOC4245) :      Proc Code.....      74; Inv Code.....      2  
(FOC4245) :      Make Dept.....      12-00; Plan Rev Num...  
(FOC4245) :      Make Time.....      63; Comm Code.....      14
```

**Note:** The second line is a FOC4246 message line, which indicates a continuation of the previous line. The sample stored procedure returns the first 40 bytes of all five (5) records to the client. The stored procedure can be set to return the entire message.



## ETPCIMS LU6.2 Adapter for IMS/TM

---

The ETPCIMS LU6.2 Adapter for IMS is positioned to provide high-speed, highly scalable, and reliable access to IMS transactions. In addition, the ETPCIMS LU6.2 Adapter for IMS returns the IMS messages to the client application in the form of an SQL data set. This allows for greater flexibility in client applications accessing the IMS transaction.

## Using ETPCIMS LU6.2

---

The ETPCIMS LU6.2 Adapter for IMS provides functionality similar to the CALLIMS feature available on the server for MVS. The ETPCIMS LU6.2 Adapter for IMS utilizes IMS/APPC for communications to IMS, and therefore only supports APPC communications.

The ETPCIMS LU6.2 Adapter for IMS allows any client on any platform to invoke a transaction running under the control of an IMS/TM transaction processing monitor. It provides a means of building on existing applications that perform transaction processing against IMS/TM to create new client/server applications that reside on the desktop. In addition, the ETPCIMS LU6.2 Adapter for IMS allows for the access of IMS/TM transactions from internet- and intranet-based applications.

## Transaction Processing With ETPCIMS LU6.2

---

The following steps are performed when a client application accesses a server to invoke the ETPCIMS LU6.2 Adapter for IMS and execute an IMS transaction.

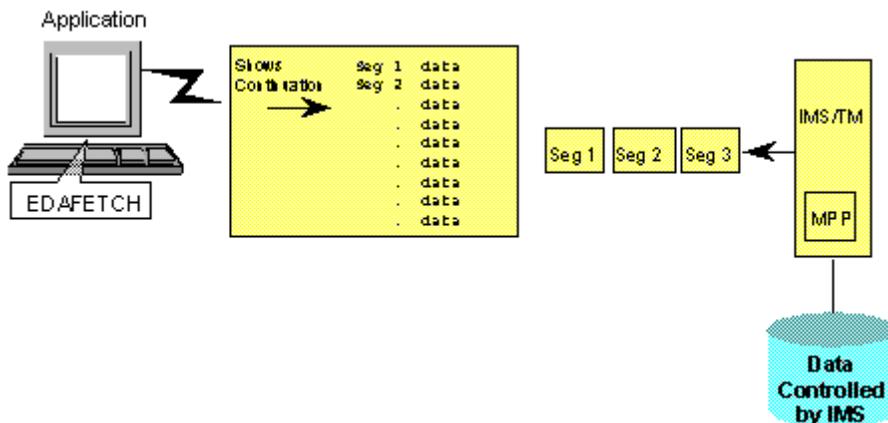
1. The client application connects to the server.
2. The client application executes the transaction as an RPC. The executed RPC is an RPC staged on the server, or it is executed by calling the ETPCIMS LU6.2 Adapter for IMS (ETPCIMS) directly. Parameters are sent as part of the calling sequence for use by the procedure.

3. If a stage RPC is used, the RPC then executes the ETPCIMS. ETPCIMS establishes a connection to the IMS/TM environment and invokes the transaction, passing an IMS message as a parameter.
4. The IMS transaction receives its input message(s) and performs its processing. It then passes the IMS message(s) back to the server. ETPCIMS returns the message(s) to the client application in the form of an answer set.
5. The client application retrieves the metadata and the answer set from the server.
6. The client application disconnects from the server, runs another RPC, or performs any other processing.

For details on the syntax and use of API method calls, see the *API Reference* manual.

### Example Accessing a Staged RPC on a Server

The following figure illustrates a client application accessing a staged RPC on a server. To call an IMS/TM transaction, execute ETPCIMS from a Dialogue Manager procedure. ETPCIMS invokes an IMS/TM transaction through the use of APPC/IMS. ETPCIMS requires IMS Release 4.1 or higher (IMS/TM) and APPC/MVS.



### How Data Is Returned With ETPCIMS

ETPCIMS returns codes and data to the client application in a series of rows. To retrieve the rows, the client application issues a method call to retrieve the data from the server. In an API transaction, the method call is the EDAFETCH.

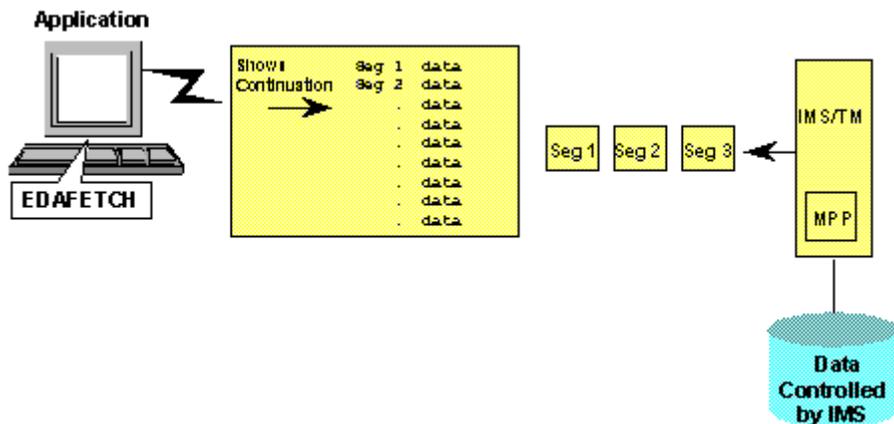
Repeated method calls retrieve all rows from ETPCIMS.

The rows returned to the client application is described by metadata from one of two sources:

- The ETPCIMS client interface creates dynamic metadata for the segments being returned from IMS. The dynamic metadata is in the form of columns up to 250 characters each for the length of the segment. For example, if a segment of 525 bytes is returned from IMS, the metadata would describe three columns: COL1 and COL2, each with 250 characters, and COL3 which is 25 characters long. The client application will then EDAFETCH the entire row and parse the row into the fields it requires.
- If more descriptive metadata is required, the answer set from IMS is staged on the server, and then allocated against an existing Master File that is used to TABLE the required named columns.

### Example Accessing IMS Through a Server

The following figure is an example of an application accessing IMS through a server.



### Environment Parameters

ETPCIMS uses several environment parameters to customize the IMS transaction processing. These parameters are set in the server configuration file as a default for all ETPCIMS LU6.2 Adapter users. The parameters include IMSDELIMITER, IMSRECORDLENGTH, and IOTRACE.

- IMSDELIMITER sets a string value to be used by the ETPCIMS LU6.2 Adapter to define the end of an IMS segment being sent to IMS. Without this environment parameter, all parameters being sent to the ETPCIMS LU6.2 Adapter are sent to IMS as separate segments. This feature is important when running multiple screen IMS transactions.

If IMSDELIMITER is not set for transaction and the executed statement is:

```
ETPCIMS &tp, &plu, &llu, &lmode, &user, &pswd, &parm1, &parm2
```

then the two parameters, &parm1 and &parm2, are sent as separate segments.

If `IMSDELIMITER = $DELIM$` is set for the transaction and the executed statement is:

```
ETPCIMS &tp, &plu, &llu, &lmode, &user, &pswd, &parm1, &parm2
```

then both parameters, `&parm1` and `&parm2`, are combined and sent as one segment.

If you add the parameters `$DELIM$$` and `&parm3`, the executed statement is:

```
ETPCIMS
```

```
&tp, &plu, &llu, &lmode, &user, &pswd, &parm1, &parm2, $$DELIM$$, &parm3
```

The parameters are then sent as two segments. The first segment is the combined `&parm1` and `&parm2` parameters. The second segment is the `&parm3` parameter.

- `IMSRECORDLENGTH` sets a default size for the dynamic metadata generated by the ETPCIMS LU6.2 Adapter. In a transaction where more than one segment is returned from IMS, the segments might be of different sizes. For example, the first segment may be smaller than a subsequent segment. If this environment parameter is not set, then the length of the first segment is used. By using this environment parameter, the dynamic metadata is created to handle the largest piece of data.

For example, if `IMSRECORDLENGTH = 80` is set, the dynamic metadata will always define one column with a length of 80 characters, regardless of the size of the actual segment.

Value of <code>IMSRecordLength</code>	Size of Returned Segments
0	72 bytes
Not set	The row is equal to the length of the first segment.
A non-zero value	The non-zero value is used as the length of the row.

- `IOTRACE` controls the generation of a trace, which contains the buffers sent to and from IMS. This environment parameter also identifies the name of the file to which the trace output is written. If this parameter is not set, then no trace is produced. For example, if `IOTRACE = A.TXT`, the I/O trace output is put into a file named `A.TXT`.

## Executing ETPCIMS

ETPCIMS is executed from a Dialogue Manager procedure with the `EX` command.

For details on the `EX` command and other Dialogue Manager commands mentioned in this section, see Chapter 4, *Writing a Dialogue Manager Procedure*.

Any or all of the variables described in the syntax that follows may be implicitly set in the Dialogue Manager procedure using the command -DEFAULTS, or explicitly set on the ETPCIMS call. The variables work the same way, whether set implicitly or explicitly. If both ways are used, the explicit setting takes precedence over the implicit setting.

## **Syntax**    **How to Execute ETPCIMS From a Dialogue Manager Procedure**

*EX ETPCIMS &TP, &PLU, &LLU, &LMODE, &UID, &PW, &MSG, &MSGn, . . .*

where:

*&TP*

Is the name of the IMS MPP transaction.

*&PLU*

Is the LU6.2 APPLID for the IMS/TM region.

*&LLU*

Is the LU6.2 APPLID for the local APPC node.

*&LMODE*

Is the log mode table entry name.

*&UID*

Is the user ID that invokes the IMS MPP transaction.

*&PW*

Is the password used to invoke the IMS MPP transaction.

*&MSG*

Is the input message for the IMS transaction. This message must exactly match the segment layout expected by the transaction.

*&MSGn*

Is the subsequent message segment for the IMS transaction. This message must exactly match the segment layout expected by the transaction. This feature is used if the IMS/TM transaction is expecting multiple input segments.

## **Example**    **Using ETPCIMS to Execute the IMS/TM Transaction**

The following is a sample Dialogue Manager procedure that uses the command ETPCIMS to execute the IMS/TM transaction. It is contained on the installation media as ETPCIMS1.FEX.

```
-DEFAULT &TP      = 'PART'  
-DEFAULT &PLU     = 'DBT5APPC'  
-DEFAULT &LLU     = 'T29MSAI1'  
-DEFAULT &LMODE   = 'LU62APPC'  
-DEFAULT &UID     = 'EDATP'  
-DEFAULT &PW      = 'TPPSWD'
```

```
EXEC ETPCIMS &TP, &PLU, &LLU, &LMODE, &UID, &PW, &1
```

The command -DEFAULT enables you to specify default values in the Dialogue Manager procedure, but they may be overridden by a client application.

## Installing ETPCIMS

---

Once you have created a Dialogue Manager procedure that includes ETPCIMS, you must place it in the appropriate directory in the server. See Chapter 1, *Introduction*, for details about stored procedure libraries and stored procedure execution order.

- Note:** •  ETPCIMS runs from Windows NT or UNIX only. The following is an example of the Windows NT directories that include the ETPCIMS Adapter as well as the ETPCIMS RPC function within a Version 5 Release 1.0 server:

```
IBI\svr51\ffs\user\ETPCIMS.DLL
```

```
IBI\svr51\ffs\catalog\ETPCIMS.FEX
```

If the RPC is called ETPCIMS.FEX, then EXORDER=FEX/PGM must be specified.

For LU6.2, access from Windows NT or UNIX requires Microsoft's SNA Server to connect to the IMS/TM region on MVS.

- For a Version 4.x server, ETPCIMS.DLL is not delivered by default. Contact your Information Builders representative.

## Server Settings and Environment Variables

---

Configuring the server environment for the ETPCIMS LU6.2 Adapter for IMS requires the setting of some server functions and environment variables. The server settings are consistent on all server platforms. The environment variables vary according to platform, and to which SNA Server you are using for that platform.

### Solaris Platform

In the EDASTART routine, export the LD\_LIBRARY\_PATH environment variable to allow the ETPCIMS LU6.2 Adapter for IMS to use SNA communications. For example,

```
export LD_LIBRARY_PATH=/opt/SUNWappc:$LD_LIBRARY_PATH
```

### HP Platform

In the EDASTART routine, export the SHLIB\_PATH environment variable to allow the ETPCIMS LU6.2 Adapter for IMS to use SNA communications. For example,

```
export SHLIB_PATH=/usr/lib/sna:$SHLIB_PATH
```

## **AIX Platform**

No export is required on this platform, since the SNA libraries are in the system library directory.

---

---

## CHAPTER 6

# Platform-specific Commands

### Topics:

- DYNAM Command (MVS)
- Comparison of TSO Commands, JCL, and DYNAM
- FILEDEF Command Under VM
- FILEDEF Command Under UNIX, Windows, OS/400, OS/390 and z/OS, and OpenVMS

These topics describe platform-specific commands that are included in a Dialogue Manager procedure. They explain the DYNAM command for MVS and the rules that apply to it, as well as the syntax and use of all DYNAM subcommands. They also provide a comparison between TSO commands and JCL to equivalent DYNAM commands.

On platforms other than MVS, native commands are directly available using the -UNIX, -VMS, -WINNT, -AS/400, and -CMS series of commands.

However, file references that are symbolic logical names for the purpose of -READ -WRITE, TABLE (with external files), and HOLD AS require a FILEDEF to create the logical reference.

On VM, the native FILEDEF command of the operating system (for example, CMS FILEDEF ...) is used. On Windows, UNIX, OS/400, and OpenVMS, an internal version of FILEDEF is used. See the FILEDEF sections for details on the use of FILEDEF.

## DYNAM Command (MVS)

---

This section describes the DYNAM command and its subcommands.

### **Syntax**    **How to Use the DYNAM Command**

The DYNAM command manipulates data sets under MVS.

*DYNAM subcommand operand [operand] . . .*

where:

*subcommand*

Is required, and specifies one of the operations (subcommands) in the list below. The abbreviated form of the subcommand's syntax is given under the full form. Details on each subcommand are provided in the following sections as noted.

<b>ALLOCATE</b> ALLOC ALLO	Allocates a data set. See <i>The ALLOCATE Subcommand</i> on page 6-4.
<b>CONCAT</b> CONC	Concatenates data sets. See <i>The CONCAT Subcommand</i> on page 6-12.
<b>FREE</b>	Frees data sets specified by ddnames or dsnames. Names may contain wildcard characters. See <i>The FREE Subcommand</i> on page 6-13.
<b>CLOSE</b> CLO	Closes data sets. Use this subcommand when data sets cannot be freed because of being open. See <i>The CLOSE Subcommand</i> on page 6-14.
<b>COPY</b>	Copies an entire data set or selected partitioned data set (PDS) members. This subcommand provides features such as record format conversion, either automatic or option controlled. See <i>The COPY Subcommand</i> on page 6-15.
<b>COPYDD</b>	Copies a sequential data set or PDS member. COPY handles all the features of COPYDD, and is recommended for use instead of COPYDD. See <i>The COPYDD Subcommand</i> on page 6-17.
<b>DELETE</b> DEL	Deletes an entire data set or selected PDS members. See <i>The DELETE Subcommand</i> on page 6-18.
<b>RENAME</b> REN	Renames an entire data set or selected PDS members. See <i>The RENAME Subcommand</i> on page 6-19.
<b>SUBMIT</b> SUB	Submits MVS jobs. See <i>The SUBMIT Subcommand</i> on page 6-20.

**COMPRESS** Compresses a PDS. See *The COMPRESS Subcommand* on page 6-21.  
**COMP**

*operand*

May be a keyword, a keyword followed by its parameter, or a parameter without a keyword.

The following rules apply to the DYNAM command:

- The subcommand, keywords, and parameters are separated with one or more blanks. Keywords are coded in free format.
- A parameter may be a list of subparameters (for example, VOLUME for a multi-volume data set). Separate subparameters in the list using commas. To include blanks between subparameters (with or without the comma), enclose the entire list in parentheses. For example:

A, B (A, B) (A B) (A, B) (A, B C, D)

- A DYNAM command may span several lines. Enter a hyphen (-) at the end of each line to be continued. When the lines are concatenated, blanks after the hyphen and leading blanks from the next line are removed. Blanks before the hyphen are removed if they are preceded by a comma. The total length of a DYNAM command may not exceed 2,048 characters.
- Most keywords may be truncated up to the shortest unambiguous length. The commonly used abbreviations are fixed. Note that the unique truncation of a keyword may not always be valid as new keywords are added. It is recommended that the full keyword be used in files.
- Fixed abbreviations are given in the following sections on the subcommands. For example, DDNAME may be abbreviated as DD, DDN, DDNA, DDNAM, or DDNAME.
- Certain keywords have synonyms. For example, the keywords FILENAME and DDNAME are synonyms, and so are DATASET and DSNAME.
- As in TSO, a data set name is enclosed in single quotation marks. Prefix substitution is not supported; specify only the fully qualified data set names in JCL.
- Some DYNAM commands accept either the ddname or data set name (dsname) as the same parameter. In such cases, the parameter is considered a ddname if it is not longer than 8 bytes, does not contain periods (.), and is not enclosed in single quotation marks ('). Otherwise, the parameter is considered a data set name. Thus, to specify an unqualified data set name, enclose it in single quotation marks.

## Use of Data Sets

MVS obtains a lock for any allocated data set name; a shared lock for those specified as SHR, and an exclusive lock for OLD, NEW, or MOD.

Although data sets are allocated more than once in a job step, only one type of lock may be obtained. For example, if the data set is initially allocated as SHR and is then allocated as OLD in the same step, the MVS lock changes from shared to exclusive, and the data set is not available for use by other jobs until *all* allocations in the job are freed.

The DYNAM commands that manipulate data sets use an improved locking mechanism, similar to that implemented in ISPF:

- Any output PDS is allocated by DYNAM (or pre-allocated by the user) as SHR. This avoids exclusive MVS locking, which lasts until all data set allocations are freed.
- To protect from simultaneous updating, DYNAM obtains an exclusive lock as used by ISPF (and other programs, including LINKEDIT), but only during the actual update time. This lock controls access to the data set between users, even from within the same job step.

**Note:** The DYNAM locking mechanism protects from simultaneous updating and possible corruption of data, but does not protect from updating and simultaneous reading. For example, it is possible to continue to read a PDS member recently deleted by another user.



## DYNAM Allocation User Exit

The DYNAM allocation user exit is an optional site-supplied routine that may be called for each data set allocation made by DYNAM. The routine may test, alter, or reject the allocation request. For more information, see Information Builders Technical Memo 7860.1, *The DYNAM User Exit*.

## The ALLOCATE Subcommand

The DYNAM ALLOCATE command allocates a data set.

### **Syntax**      **How to Use the ALLOCATE Subcommand**

```
DYNAM ALLOCATE [disposition] [CLOSE]
DDNAME ddname [DEFER] [DSNAME dsname[(memname)] ] [DUMMY]
[EXPDT date]
[HIPER OFF]
[INPT|OUTPT]
[LABEL type]
[MEMBER memname] [status] [MSVGP msvgp]
```

[PARALLEL] [PASSWORD *password*] [PERM] [POSITION *nnnn*]  
 [REFVOL *dsname*] [RETPD *days*] [REUSE]  
 [UNIT *unit*]  
 [VOLUME *volser*]

Space operands are:

[*format*]  
 [*parameter*]  
 [DIR *n*]  
 [PRIMARY *n1*]  
 [RELEASE] [ROUND]  
 [SECONDARY *n2*] [SPACE *space*]

DCB operands are:

[BLKSIZE *n*] [BUFNO *n*]  
 [DEN *n*] [DSORG *dsorg*]  
 [LRECL *n*]  
 [RECFM *recfm*] [REFDD *ddname*] [REFDSN *dsname*]

SMS and VSAM operands are:

[DATACLASS *name*] [DSNTYPE|{LIBRARY|PDS}]  
 [KEYOFF *n*]  
 [LIKE *dsname*]  
 [MGMTCLASS *name*]  
 [RECORG *recorg*]  
 [SECMODEL *name*] [STORCLASS *name*]  
 [BUFND *m*]  
 [BUFNI *n*]

Output printing operands are:

[DEST *dest*[. *user*]]  
 [FCB *name* [ALIGN|VERIFY]] [FORMS *name*]  
 [HOLD]  
 [OUTLIM *n*] [OUTPUT *name*]  
 [SYSOUT *class*]  
 [USER *user*]  
 [WRITER *name*]

where:

ALLOCATE

Can be abbreviated as ALLOC or ALLO.

*disposition*

Is one of the following:

CATALOG is the data set normal disposition.

DELETE is the data set normal disposition.

KEEP is the data set normal disposition.

UNCATALOG is the data set normal disposition.

UNCAT is the data set normal disposition.

By default, for a data set status of NEW, if *dsname* is specified, the disposition is CATALOG; otherwise, the disposition is DELETE. Is incompatible with SYSOUT. Synonyms are CATALOG and CATLG.

CLOSE

Is deallocation of the data set at close, rather than at the end of the step. The JCL analogy is FREE=CLOSE.

DDNAME *ddname*

DD

Is the DDNAME to be associated with an allocation; it must be specified. Synonyms are: DDNAME, FILENAME.

DEFER

Assigns device(s) to the data set but defers mounting of the volume(s) until the data set is opened. The JCL analogy is DEFER in UNIT.

DSNAME *dsname*

[ (*memname*) ]

The member name is specified either in parentheses after *dsname* or using keyword MEMBER (see below). If *dsname* is specified as an asterisk (\*), terminal is allocated. This is used for output only. Synonyms are DSNAME and DATASET.

DUMMY

Allocates a dummy data set.

EXPDT *date*

Is the expiration date in format YYDDD, YYYY/DDD, or YYYYDDD. Is incompatible with RETPD and SYSOUT.

HIPER OFF

Prohibits allocation in a hiperspace. Is equivalent to UNIT NOHIPER, and is used when UNIT is also to be specified. For example, UNIT VIO HIPER OFF.

INPT

Data set is to be processed as input only (INPT) or output only (OUTPUT). The JCL analogy is IN in LABEL. Is incompatible with SYSOUT.

**OUTPT**

Data set is to be processed as input only (INPT) or output only (OUTPUT). JCL analogy: IN in LABEL. Is incompatible with SYSOUT.

**LABEL** *type*

Specifies type of volume labels. Can be one of the following: NL, SL, NSL, SUL, BLP, LTM, AL, or AUL. Is incompatible with SYSOUT.

**MEMBER** *memname*

Is the name of a PDS member to be allocated. See also: DSNAME.

**status**

Is the data set status. Possible values are:

**MOD** is the data set status.

**NEW** is the default data set status. Incompatible with SYSOUT.

**OLD** is the data set status.

**SHR** is the data set status.

**MSVGP** *msvgp*

Is the identification of a group of mass storage system (MSS) virtual volumes. Is incompatible with SYSOUT and VOLUME.

**PARALLEL**

Each volume is to be mounted on a separate device. The JCL analogy is P in UNIT.

**PASSWORD** *password*

Password for a password-protected data set.

**PERM**

The allocation is to be permanent—that is, protected from being freed or concatenated by any DYNAM command issued by an MSO user. The operand is valid only in a MSO server initialization profile.

**POSITION** *nnnn*

Data set sequence number on a tape volume, up to 9999. The JCL analogy is the first subparameter in LABEL.

**REFVOL** *dsname*

Volume serial information is to be obtained from the named cataloged data set. The JCL analogy is VOL=REF=*dsname*. Is incompatible with SYSOUT and VOLUME.

**RETPD** *days*

Is the retention period, up to 9999 days. Is incompatible with EXPDT and SYSOUT.

REUSE

REU

If the ddname to be allocated is already in use, it is to be freed.

UNIT *unit*

Is the device group name, device type, specific unit address, or NOHIPER. NOHIPER prohibits allocation in a hiperspace, and is meaningful for a temporary (NEW, DELETE) data set; see also HIPER OFF.

VOLUME *volser*

VOL

Are volume serial numbers. Are incompatible with REFSVOL and SYSOUT. Synonyms are VOLume and VOLser.

Space operands may be:

*format*

The format of the primary space to be allocated. Possible values are:

ALX is up to five contiguous areas.

CONTIG is one contiguous area.

MXIG is one maximal contiguous area.

JCL analogy: ALX/CONTIG/MXIG in SPACE.

*n*

Represents units of primary and secondary space allocation.

*parameter*

The parameter for space allocation. Possible values are:

BLOCKS [*n*]

CYLINDERS

MEGABYTES

PAGES

TRACKS

*n* represents units of primary and secondary space allocation. If the parameter for BLOCKS is omitted, the average block length is copied from BLKSIZE. If the space unit is omitted but SPACE and BLKSIZE are specified, BLOCKS equal BLKSIZE is used. For PAGES, BLOCKS 4096 is used. BLKSIZE must be specified if the BLOCKS parameter is specified.

Synonyms are: CYLINDERS, CYLs; TRACKS, TRKs.

DIR *n*

The number of 256-byte records for the directory of a PDS.

PRIMARY *n1*

The primary space quantity. See also SPACE.

**RELEASE**

The unused space is to be released when the data set is closed.  
Synonyms are: RELEASE, RLSE.

**ROUND**

If space is requested in BLOCKS, MEGABYTES, or PAGES, it is to be rounded to whole cylinder(s).

**SECONDARY *n2***

Is the secondary space quantity. See also SPACE.

**SPACE *space*****SP**

The primary (n1) and/or secondary (n2) space quantity in one of the following formats:

$n1 / (n1) / n1, n2 / (n1, n2) / n1 \ n2 / (n1 \ n2) / , n2 / (, n2)$

See also PRIMARY and SECONDARY.

DCB operands may be:

**BLKSIZE *n***

The block size, up to 32760. See also BLOCKS.

**BUFNO *n***

The number of buffers, up to 255.

**DEN *n***

*n* represents magnetic tape density: 0, 1, 2, 3, or 4 for 200, 556, 800, 1600, 6250 bpi respectively.

**DSORG *dsorg***

The data set organization. Default, for NEW only: PO if DIR or DSNTYPE specified; PS otherwise. Following values are syntactically correct:

VS	VSAM
PO/POU	PDS or PDS unmovable.
DA/DAU	Direct access or direct access unmovable.
PS/PSU	Physical sequential or physical sequential unmovable.

**LRECL *n***

The logical record length, up to 32760.

**RECFM *rcfm***

The record format. The first letter must be D, F, U, or V, which may be followed by any valid combination of A, B, M, S, or T:

A	Records with ISO/ANSI control characters.
B	Blocked records.
D	Variable-length ISO/ANSI tape records.
F	Fixed-length records.
M	Records with machine code control characters.
S	Standard fixed-length or spanned variable-length records.
T	Track overflow.
U	Undefined-length records.
V	Variable-length records.

**REFDD** *ddname*

DCB attributes are to be copied from the specified *ddname*. Under TSO, EXPDT and INPT/OUTPT specifications are also copied. Any of those can be overridden by the appropriate keyword on the same command. The JCL analogy is DCB=\*.*ddname*. Is incompatible with REFDSN.

**REFDSN** *dsname*

DCB attributes (DSORG, RECFM, OPTCD, BLKSIZE, LRECL, RKP, KEYLEN) and EXPDT are to be copied from the specified cataloged data set. Any of those can be overridden by the appropriate keyword on the same command. The JCL analogy is DCB=*dsname*. Is incompatible with REFDD.

SMS and VSAM operands are:

**DATACLASS** *name*

The name of a data class for an SMS-managed data set.

**DSNTYPE** {*LIBRARY* | *PDS*}

*LIBRARY* is for a new partitioned extended (PDSE), and *PDS* is for a new partitioned data set. A PDSE cannot contain load modules, should be SMS-managed, and allows concurrent updating of different members.

**KEYOFF** *n*

The offset of the key in each logical record for a new VSAM key-sequenced (RECORG KS) data set.

**LIKE** *dsname*

Allocation attributes (DSORG, RECORG, or RECFM, LRECL, KEYLEN, KEYOFF, SPace, DIR) are to be copied from the specified cataloged data set (model). Any of those can be overridden by the appropriate keyword on the same command.

**MGMTCLASS** *name*

The name of a management class for an SMS-managed data set.

**RECORG** *recorg*

The VSAM record organization: KS, ES, RR, or LS for key-sequenced, entry-sequenced, relative record, or linear space data sets, respectively.

**SECMODEL** *name*

The data set RACF profile is to be copied from the named existing RACF profile.

**STORCLASS** *name*

The name of a storage class for an SMS-managed data set.

**BUFND** *m*

The number of VSAM DATA buffers.

**BUFNI** *n*

The number of VSAM INDEX buffers.

Output printing operands may be:

**DEST** *dest[.user]*

The remote destination for a SYSOUT data set. In conjunction with user ID, it is a node and a user at that node; the user ID is coded after the period (.) or using the USER keyword (see below).

**FCB** *name***[ALIGN/VERIFY]**

The name of an FCB (forms control buffer) image to be used for printing of a data set. The operator may be asked to check the printer forms alignment (ALIGN), or to verify the FCB image name displayed on the printer (VERIFY).

**FORMS** *name***FORM**

A SYSOUT form name. JCL analogy: third subparameter in SYSOUT, FORMS in OUTPUT JCL.

**HOLD**

A SYSOUT data set is to be placed on the hold queue.

**OUTLIM** *n*

A limit for the number of logical records in a SYSOUT data set.

**OUTPUT** *name*

The name(s) of OUTPUT JCL statement(s) to be associated with a SYSOUT data set.

**SYSOUT** *class*

A SYSOUT data set is to be allocated and the specified output class (A-Z, 0-9) is to be assigned. If an asterisk (\*) or NULL is coded, the class is copied either from CLASS in OUTPUT JCL if it is specified, or otherwise from MSGCLASS in JOB.

**USER** *user*

A SYSOUT data set is to be routed to the specified user ID. DEST (see above) is required to specify a user's node.

**WRITER** *name*

The name of an installation-written system output printing routine. The JCL analogy is the second subparameter in SYSOUT. Is incompatible with USER.

In addition to the shown fixed abbreviations and synonyms, keywords may be abbreviated up to the unique truncation. Those abbreviations are not fixed and may be changed when new keywords are added. They may be used interactively to save some keystrokes, but when a command is saved in a file, it is recommended that you use unabbreviated keywords.

**Examples:**

Allocate an existing data set:

```
DYNAM ALLOC DD MYDD DS MYID.DATA.SET SHR REU
```

Allocate a new data set. Defaults are NEW, CATALOG (*dsname* present), and DSORG PO (not-zero DIR present):

```
DYNAM ALLOC DD MYDD DS MYID.DATA.SET SPACE 6,2 TRACKS DIR 4 UNIT SYSDA -  
RECFM FB LRECL 80 BLKSIZE 1600
```

Allocate a terminal:

```
DYNAM ALLOC DD MYDD DS *
```

Allocate a SYSOUT data set with default output class. Upon freeing, the data set is sent to the user ID U1234 at node SYSVM:

```
DYNAM ALLOC DD MYDD SYSOUT * DEST SYSVM.U1234
```

## The CONCAT Subcommand

The DYNAM CONCAT command concatenates up to 16 data sets.

### **Syntax**

#### **How to Use the CONCAT Subcommand**

```
DYNAM CONCAT [PERM] DDNAME ddname1 ddname2 [ddname3...]
```

where:

CONCAT

Can be abbreviated as CONC.

PERM

Is optional. This marks the concatenation as permanent—that is, protected from being freed or concatenated again by any DYNAM command issued by a MSO user. Valid only in a MSO server initialization profile.

DDNAME

DDN

DD

Are required; synonym is FILENAME.

*ddname1*

Is the first *ddname* to be concatenated and associated with the resulting concatenated group.

*ddname2*

Is the second *ddname* and any subsequent *ddname* to be concatenated.

For example:

```
DYNAM CONCAT DDN EDARPC MYEX NEWEX
```

## The FREE Subcommand

The DYNAM FREE command deallocates any number of specified data sets.

### **Syntax** How to Use the FREE Subcommand

```
DYNAM FREE {DDNAME ddname [ddname...] | DSNAME dsname [dsname...] }
```

where:

DDNAME

DDN

DD

Are required if there is no *dsname*; synonym is FILENAME.

*ddname*

Is the *ddname* of the data set to be freed.

DSNAME

DSN

DS

Are required if there is no *ddname*; the synonym is DATASET.

*dsname*

Is the name of the data set to be freed. All *ddnames* associated with this *dsname*, except concatenated groups, are deallocated.

While at least one *ddname* or data set name is required, you may specify more than one *ddname* or data set name. Each specified name may contain asterisks (\*) and question marks (?) as wildcards. Wildcards are special characters used to specify a subset of names rather than one name. The wildcards appear anywhere in a name and mean the following:

\*

Represents any number of characters. For example, \*Q\* matches any name containing the character Q.

?

Represents any single character. For example, ?Q? matches any 3-character name containing the character Q in the middle.

If the *ddname* is not found, an error message is issued only if a single *ddname* without wildcards is specified. An error message is not displayed if a data set or more than one *ddname* is not found.

#### Examples:

```
DYNAM FREE DDN SYS0* TEMP?
```

```
DYNAM FREE DSN MYID.DATA.SET
```

## The CLOSE Subcommand

The DYNAM CLOSE command closes data sets that cannot be freed because they are opened.

### **Syntax** How to Use the CLOSE Subcommand

```
DYNAM CLOSE {DDNAME ddname [ddname...] | DSNAME dsname [dsname...] }
```

where:

CLOSE

Can be abbreviated as CLO.

DDNAME

DDN

DD

Are required if there is no *dsname*; the synonym is FILENAME.

*ddname*

Is the *ddname* of the data set to be closed.

DSNAME  
DSN  
DS

Are required if there is no ddname; the synonym is DATASET.

*dsname*

Is the name of the data set to be closed. All ddnames associated with this dsname, except concatenated groups, are closed.

While at least one ddname or data set name is required, more than one ddname or data set name may be specified. Each specified name may contain wildcard characters. The same rules apply to the DYNAM CLOSE command as to the DYNAM FREE command.

## The COPY Subcommand

The DYNAM COPY command copies an entire MVS data set or selected PDS members.

### **Syntax** How to Use the COPY Subcommand

```
DYNAM COPY dname1 {[TO] dname2 [[MEMBER] members] | [MEMBER]
members]} [options]
```

where:

*dname1*

Is the dsname or ddname of the input data set. This is a positional parameter. It must precede all other operands.

TO

May be omitted if *dname2* does not match a reserved word, the MEMBER keyword, an option, or the TO keyword. To avoid confusion, use the TO keyword whenever *dname2* is a ddname.

*dname2*

Is the dsname or ddname of the output data set. If the output data set is not a PDS and the dsname is specified, it is allocated as OLD. If the ddname is specified, and the status is SHR, ensure that other users do not access the data set during COPY. Unlike ISPF, DYNAM locks a non-PDS data set in order to prevent simultaneous updating by different DYNAM users.

MEMBER

May be omitted if members are specified in parentheses.

*members*

Can be a single member specification or a list of member specifications. If the members are enclosed in parentheses, blanks preceding the left parenthesis may be omitted.

*options*

May be one or more of the following options:

**APPEND** adds the input to the end of the existing data, if the output is a sequential data set.

**FORCE** copies input DCB attributes (RECFM, BLKSIZE, LRECL, and KEYLEN) to the output data set. By default, only missing values are assigned.

**KEYMOD** allows key modification according to input/output KEYLEN: truncation or padding with binary zeros.

**REPLACE** replaces all output members matching the selected member names.

**TRUNCATE** allows truncation of input records that are longer than the output record length. Since trailing blanks are truncated automatically when RECFM is different, the keyword is used either to cut records of the same format or to cut non-blank data.

A member specification has the following syntax

*mem* [ , [ *newmem* ] [ , REPLACE ] ]

where:

*mem*

Is the selected member name.

*newmem*

Is the optional new name for the output member.

**REPLACE**

Is optional and specifies an existing member to be replaced in the output PDS.

Since the comma may be used in member specifications, they are separated with one or more blanks when specified in a list. Therefore, a list of member specifications is always enclosed in parentheses. For example:

(MEM MEM, NEWMEM MEM, NEWMEM, R MEM, , R)

- Note:** • All conversions between different DCB attributes (RECFM, BLKSIZE, and LRECL) are performed automatically.
- 
- If the entire PDS is copied or any selected member's directory entry contains a TTRN in user data (for example, a load module), the IBM utility IEBCOPY is invoked. In this case, all options except REPLACE are ignored, format conversion is not possible, and copying members to the same PDS is not supported. Note that IEBCOPY requires APF authorization in order to be performed.
  - If the main member and its alias names are copied, the relationship remains the same on the output PDS.
  - If a specified ddname has been allocated with a member name, the data set is treated as sequential.

### Examples:

Copies the entire data set, whether it is a PDS or not.

```
DYNAM COPY MYDD MYID.DATA.SET
```

All four commands are equivalent. Either input or output may be a sequential data set, or both are PDSs.

```
DYNAM COPY MYDD MYID.DATA.SET MEMBER MEM
DYNAM COPY MYDD MYID.DATA.SET (MEM)
DYNAM COPY MYDD (MEM) MYID.DATA.SET
DYNAM COPY MYDD MEMBER MEM MYID.DATA.SET
```

Copies and renames one member.

```
DYNAM COPY MYID.DATA.LIB TO MYDD (MEM1, MEM2)
```

Copies two members.

```
DYNAM COPY MYID.DATA.LIB TO MYDD (MEM1 MEM2)
```

Copies two members into same PDS with renaming.

```
DYNAM COPY MYDD (OLD1, NEW1, R OLD2, NEW2)
DYNAM COPY MYDD (OLD1, NEW1 OLD2, NEW2) REPL
```

## The COPYDD Subcommand

The DYNAM COPYDD command copies a sequential data set or PDS member.

### **Syntax** How to Use the COPYDD Subcommand

```
DYNAM COPYDD ddname1 [ (mem1) ] ddname2 [ (mem2) ]
```

where:

*ddname1*

Is the ddname of the input data set.

*mem1*

Is optional. It is the input member name.

*ddname2*

Is the ddname of the output data set.

*mem2*

Is optional. It is the output member name.

**Note:** • If the specified ddname has been allocated with a member name, the data set is treated as sequential.



- Identically named members are always replaced on the output PDS.
- All conversions between different DCB attributes (RECFM, BLKSIZE, and LRECL) are performed automatically.
- Since the DYNAM COPY command has more features than COPYDD, it is recommended that you use COPY instead of COPYDD.

## The DELETE Subcommand

The DYNAM DELETE command deletes an entire MVS data set or selected PDS members.

### **Syntax** How to Use the DELETE Subcommand

DYNAM DELETE *dsname*

To delete individual members, use

DYNAM DELETE *dname* [MEMBER] *members*

where:

DELETE

Can be abbreviated as DEL.

*dsname*

Is the data set name to be deleted and uncataloged.

*dname*

Is the dsname or ddname of a PDS containing one or more members to be deleted. The ISPF-like lock is obtained.

MEMBER

May be omitted if the members are specified in parentheses.

*members*

Can be a single member name or a list of members. If the members are enclosed in parentheses, blanks before the left parenthesis can be omitted.

**Examples:**

```
DYNAM DELETE MYID.DATA.OLD
DYNAM DEL MYID.DATA.LIB MEMBER OLD1,OLD2
DYNAM DELETE MYDD(OLD1,OLD2)
DYNAM DEL MYDD(OLD1 OLD2 OLD3)
```

## The RENAME Subcommand

The DYNAM RENAME command renames an entire MVS data set or selected PDS members.

### *Syntax* How to Use the RENAME Subcommand

```
DYNAM RENAME dsname1 dsname2
```

To rename individual members, use

```
DYNAM RENAME dname [MEMBER] members [REPLACE]
```

where:

RENAME

Can be abbreviated as REN.

*dsname1*

Is the data set name to be renamed and uncataloged.

*dsname2*

Is the new name to be assigned to the data set and cataloged.

*dname*

Is the dsname or ddname of a PDS containing one or more members to be renamed. The ISPF-like lock is obtained.

MEMBER

May be omitted if the members are specified in parentheses.

*members*

Can be a single member specification or a list of members. If the members are enclosed in parentheses, blanks before the left parenthesis can be omitted.

REPLACE

Is optional. This replaces all members matching the specified new names.

A member specification has the following syntax

```
oldmem, newmem [, REPLACE]
```

where:

*oldmem*

Is the original member name.

*newmem*

Is the new member name.

REPLACE

Is optional and replaces existing members with the same name as *newmem*.

Since the comma is used in member specifications, each pair of members is separated with one or more blanks when specified in a list; therefore, a list of member specifications is always enclosed in parentheses.

### Examples:

```
DYNAM RENAME MYID.DATA.OLD MYID.DATA.NEW
DYNAM REN MYID.DATA.LIB MEMBER OLD,NEW,R
DYNAM RENAME MYDD(OLD1,NEW1,R OLD2,NEW2)
DYNAM REN MYDD(OLD1,NEW1 OLD2,NEW2) REPL
```

## The SUBMIT Subcommand

The DYNAM SUBMIT command submits jobs to MVS.

### **Syntax** How to Use the SUBMIT Subcommand

```
DYNAM SUBMIT dname [[MEMBER] members]
```

where:

SUBMIT

Can be abbreviated as SUB.

*dname*

Is the dsname or ddname of the input data set(s) containing JCL to be submitted. The ddname specifies a concatenation of data sets.

MEMBER

May be omitted if the members are specified in parentheses.

*members*

May be a single member name or a list of members. When a member list is submitted, the resulting job stream is the concatenation of the members. If the members are enclosed in parentheses, blanks before the left parenthesis can be omitted.

### Examples:

```
DYNAM SUBMIT MYDD MEMBER ASM, PROG, LKED
DYNAM SUB MYDD (ASM, PROG, LKED)
DYNAM SUB MYID.DATA.LIB (CREATE LOAD)
DYNAM SUBMIT MYFILE
```

 **Note:** The DYNAM SUBMIT command provides an interface with the submit user exit IKJEFF10 as described in the *IBM TSO Extensions Version 2 Customization* manual. For details, see Information Builders Technical Memo 7859, *Enabling a Site-Specified Submit Exit Routine*.

## The COMPRESS Subcommand

The DYNAM COMPRESS command compresses the partitioned data sets (PDS).

### Syntax **How to Use the COMPRESS Subcommand**

```
DYNAM COMPRESS dname [dname] ...
```

where:

COMPRESS

Can be abbreviated as COMP.

*dname*

Is the dsname or ddname of a PDS to be compressed. The ISPF-like lock is obtained.

If the dsname is specified, it is allocated as OLD. If the ddname is specified and status is SHR, make sure that another user does not access the PDS during the compress operation.

 **Note:** DYNAM COMPRESS uses the IBM utility IEBCOPY, and therefore are only used when running with APF authorization.

### Examples:

```
DYNAM COMPRESS MYDD
DYNAM COMPRESS MYID.DATA.LIB
DYNAM COMP MYDD MYID.DATA.LIB
```

## Comparison of TSO Commands, JCL, and DYNAM

This section shows examples of TSO commands and JCL, compared to the equivalent DYNAM commands.

### Example **Allocating an Existing File**

<b>TSO:</b>	TSO ALLOC F(EDARPC) DA ('MYUSER.EDARPC.DATA') SHR
<b>JCL:</b>	//EDARPC DD DSN=MYUSER.EDARPC.DATA, DISP=SHR
<b>DYNAM:</b>	DYNAM ALLOC FILE EDARPC DA MYUSER.EDARPC.DATA SHR

**Example**    **Creating a New Data Set**

<b>TSO:</b>	TSO        ALLOC F(EDARPC) DA('MYUSER.EDARPC.DATA') - SPACE(5,3) TRACKS CATALOG DIR(2) - UNIT(SYSDA) USING(NEWDCB) - LRECL(80) RECFM(F B) BLKSIZE(1600)
<b>JCL:</b>	//EDARPC DD DSN=MYUSER.EDARPC.DATA,DISP=(NEW,CATLG), //    SPACE=(TRK,(5,3,2)),UNIT=SYSDA, //    CB=(LRECL=80,RECFM=FB,BLKSIZE=1600)
<b>DYNAM:</b>	DYNAM ALLOC FILE EDARPC DA MYUSER.EDARPC.DATA - SPACE 5,3 TRACKS CATLG DIR 2UNITSYSDA - LRECL 80 RECFM FB BLKSIZE 1600

**Example**    **Freeing Files**

<b>TSO:</b>	TSO FREE F(EDARPC)
<b>DYNAM:</b>	DYNAM FREE FILE EDARPC

**Example**    **Concatenating Files**

<b>TSO:</b>	TSO ALLOC F(EDARPC) DA('MYUSER.EDARPC.DATA' - 'MYUSER.PROGRAMS.DATA') SHR
<b>JCL:</b>	//EDARPC DD DSN=MYUSER.EDARPC.DATA,DISP=SHR //    DD DSN=MYUSER.PROGRAMS.DATA,DISP=SHR
<b>DYNAM:</b>	DYNAM ALLOC FILE EDARPC DA MYUSER.EDARPC.DATA SHR DYNAM ALLOC FILE PROGRAMS DA MYUSER.PROGRAMS.DATA SHR DYNAM CONCAT FILE EDARPC PROGRAMS

**FILEDEF Command Under VM**

---

The VM Server uses the FILEDEF command of the VM/CMS operating system and retrieves the file attribute information from the file. In the case of HOLD, the attributes are adjusted as needed.

**Syntax**    **How to Use the FILEDEF Command in VM**

CMS FILEDEF *dd* DISK *fn ft fm*

where:

*dd*

Is the logical reference name.

*fn*

Is the file name.

*ft*

Is the file type.

*fm*

Is the file mode.

For more information on the specifics of FILEDEF options, see the VM/CMS manuals.

## FILEDEF Command Under UNIX, Windows, OS/400, OS/390 and z/OS, and OpenVMS

---

A logical name (or ddname) is a shorthand name that points to the physical file name as the operating system actually knows the file. Logical names simplify code by allowing short names to be used in place of the longer physical file name.

The FILEDEF command assigns a logical name (or ddname) to a physical file name and specifies file attributes. FILEDEF assignments are in effect for the duration of a connection (except when a server is running in Pool Mode). They are released when the connection to the server is closed or a FILEDEF CLEAR is issued.

### **Syntax** How to Use the FILEDEF Command in UNIX, Windows, OS/400, OS/390 and z/OS, and OpenVMS

```
FILEDEF ddname devicetype fileid [([LRECL n] [RECFM fm] [APPEND])]
```

```
FILEDEF ddname CLEAR
```

where:

*ddname*

Is the logical name. It may contain 1 to 8 alphanumeric characters.

*devicetype*

Identifies the type of device with which to interact. Specifies DISK for a file that resides on disk. Other device types are PRINTER, TRMIN, and TRMOUT, which have special meanings and options. For more information, see *Other FILEDEF Features* on page 6-24.

CLEAR

Clears the specified ddname.

*fileid*

Is the physical name of the file as it is known on the particular operating system using the native style of the operating system. For instance, c:\mydir\myfile.dat (Windows), \\mymachine\mydir\myfile.dat (Windows), /home/myhome/mydir/mtfile.dat (UNIX, OS/390 and z/OS, OS/400 IFS), DISK\$MYDISK:[MYHOME.MYDIR]MYFILE.DAT (OpenVMS), and MYLIB/MYFILE(MYMEMB) (OS/400 QSYS). UNIX, OS/390 and z/OS, and OS/400 IFS are case sensitive file systems where lower case file names are the norm, so appropriate names should be used when coding the fileid option of the FILEDEF. To support directory names with embedded blanks on Windows, the complete fileid needs to be enclosed in single quotation marks.

There is an additional mode of operation when APP ENABLE is active (the default as of 5.2.0). In this mode of operation UNIX like relative path name is used and the path refers to which application directory under the APPROOT directory. For instance, abc/mydata.ftm is the abc directory, which found under the directory pointed to by APPROOT.

**Note:**



APP usage is limited to one directory below APPROOT, any other usage is illegal.

*LRECL n*

Specifies the record length, n, in bytes. This parameter is optional. If you omit it, the default is 80 bytes. Note that the right parenthesis preceding the optional parameters is required.

*RECFM fm*

Describes the record format. Specifies F for fixed format, V for variable format. This parameter is optional. If you omit it, the default is fixed format. Note that the right parenthesis preceding the optional parameters is required.

*APPEND*

Enables you to open the specified file and add new material at the end of the file. This parameter is optional. If you omit it and the specified file exists, it will be overwritten. Note that the right parenthesis preceding the optional parameters is required.

Note that FOCUS data sources (files with the .foc extension) that do not conform to the default naming conventions are identified using the USE command, not FILEDEF.

## Other FILEDEF Features

PRINTER as a device type is used to change the default output file for the OFFLINE print file.

FILEDEF TRMIN TERM LOWER is used to change the uppercasing behavior of an interactive session (edastart -t) into case sensitive mode. FILEDEF TRMIN TERM UP is used to restore default behavior. Interactive session mode is typically used for testing and is not considered a production feature for general use.

FILEDEF TRMOUT DISK fileid is used to capture session output into a file during an interactive session (edastart -t).

FILEDEF TRMOUT TERM is used to restore default behavior. Interactive session mode is typically used for testing and is not considered a production feature for general use.

## **OFFLINE Printing**

Server side printing of formatted reports is accomplished using the OFFLINE command, which sets up and issues a default OFFLINE FILEDEF to receive the formatted outputs after an OFFLINE CLOSE is issued.

There may be one or more outputs buffered to the same output file for printing, but they are not released to the file until an OFFLINE CLOSE is issued. If a system level variable for FOCPRINT is available at OFFLINE CLOSE time, it will be used to attempt printing of the actual file.

The FOCPRINT variable may be any operating system command (typically a print command) using "\$1" as a wildcard for the actual offline file. If a system variable for FOCPRINT has not been configured, printing may also be accomplished by issuing a system command immediately after the OFFLINE CLOSE to take action on the OFFLINE file.



---

---

## APPENDIX A

# Dialogue Manager Quick Reference

**Topic:**

- Dialogue Manager Commands

This topic describes all the Dialogue Manager commands in alphabetical order. It also provides the syntax and explains the functions.

## Dialogue Manager Commands

<b>Command:</b>	<b>-*</b>
<b>Syntax:</b>	<p><i>-* text</i></p> <p>where:</p> <p><i>text</i></p> <p>Is a comment. A space is not required between <i>-*</i> and <i>text</i>.</p>
<b>Function:</b>	<p>The command <i>-*</i> signals the beginning of a comment line.</p> <p>Any number of comment lines follows one another, but each must begin with <i>-*</i>. A comment line may be placed at the beginning or end of a procedure, or in between commands. However, it cannot be on the same line as a command.</p> <p>Use comment lines liberally to document a stored procedure so that its purpose and history are clear to others.</p>

<b>Command:</b>	<b>-?</b>
<b>Syntax:</b>	<p><i>-? &amp;[string]</i></p> <p>where:</p> <p><i>string</i></p> <p>Is an optional variable name of up to 12 characters. If this parameter is not specified, the current values of all local, global, and defined system and statistical variables are displayed.</p>
<b>Function:</b>	The command <i>-?</i> displays the current value of a local variable.

<b>Command:</b>	<b>-CLOSE</b>
<b>Syntax:</b>	<p><i>-CLOSE filename</i></p> <p>where:</p> <p><i>filename</i></p> <p>Is a symbolic name associated with a physical file known to the operating system.</p>
<b>Function:</b>	<p><i>-CLOSE</i> closes an external file opened with the <i>-READ</i> or <i>-WRITE NOCLOSE</i> option. The <i>NOCLOSE</i> option keeps a file open until the <i>-READ</i> or <i>-WRITE</i> operation is complete.</p> <p>The external file must be defined to the operating system.</p>

<b>Command:</b>	<b>-AS/400</b>
<b>Syntax:</b>	<i>AS/400 command</i> where: <i>command</i> Is an OS/400 command.
<b>Function:</b>	-AS/400 executes an OS/400 operating system command from a procedure.

<b>Command:</b>	<b>-CMS</b>
<b>Syntax:</b>	<i>CMS command</i> where: <i>command</i> Is a CMS command.
<b>Function:</b>	-CMS executes a CMS operating system command from a procedure.

<b>Command:</b>	<b>-DEFAULTS</b>
<b>Syntax:</b>	<i>-DEFAULTS &amp;[&amp;]name=value [...]</i> where: <i>&amp;name</i> Is a name of a variable. <i>value</i> Is the default value assigned to the variable.
<b>Function:</b>	-DEFAULTS supplies an initial (default) value for a variable that had no value before the command was processed.  Override values set with -DEFAULTS by supplying new values: <ul style="list-style-type: none"> <li>• On the function call EDARPC.</li> <li>• On the command line EXEC.</li> <li>• With the command -SET subsequent to the command -DEFAULTS.</li> </ul> By supplying values to variables in a stored procedure, -DEFAULTS helps ensure that it runs correctly.

<b>Command:</b>	<b>-DOS</b>
<b>Syntax:</b>	<p><i>-DOS command</i></p> <p>where:</p> <p><i>command</i></p> <p>Is a Windows or DOS command.</p>
<b>Function:</b>	-DOS executes a Windows or DOS operating system command from a procedure.

<b>Command:</b>	<b>-EXIT</b>
<b>Syntax:</b>	<i>-EXIT</i>
<b>Function:</b>	<p>-EXIT forces a stored procedure to end. All stacked commands are executed and the stored procedure exits (if the stored procedure was called by another one, the calling procedure continues processing).</p> <p>Use -EXIT for terminating a stored procedure after processing a final branch that completes the desired task.</p> <p>The last line of a stored procedure is an implicit -EXIT. In other words, the procedure ends after the last line is read.</p>

<b>Command:</b>	<b>-GOTO</b>
<b>Syntax:</b>	<p><i>-GOTO label</i></p> <p>·</p> <p>·</p> <p>·</p> <p><i>-label [TYPE text]</i></p> <p>where:</p> <p><i>label</i></p> <p>Is a user-defined name of up to 12 characters that specifies the target of the -GOTO action.</p> <p>Do not use embedded blanks or the name of any other Dialogue Manager command except -QUIT or -EXIT. Do not use words that may be confused with functions, arithmetic and logical operations, and so on.</p> <p><i>TYPE text</i></p> <p>Optionally sends a message to the client application.</p>

<b>Command:</b>	<b>-GOTO</b>
<b>Function:</b>	<p>-GOTO forces an unconditional branch to the specified label.</p> <p>If Dialogue Manager finds the label, processing continues with the line following it.</p> <p>If Dialogue Manager does not find the label, processing ends and an error message is displayed.</p>

<b>Command:</b>	<b>-IF</b>
<b>Syntax:</b>	<pre>-IF <i>expression</i> [THEN] GOTO <i>label1</i> [;]</pre> <pre>-[ELSE GOTO <i>label2</i> [;]]</pre> <pre>-[ELSE IF... [;]]</pre> <p>where:</p> <p><i>label</i></p> <p>Is a user-defined name of up to 12 characters that specifies the target of the GOTO action.</p> <p>Do not use embedded blanks or the name of any other Dialogue Manager command except -QUIT or -EXIT. Do not use words that may be confused with functions, arithmetic or logical operations, and so on.</p> <p><i>expression</i></p> <p>Is a valid expression. Literals need not be enclosed in single quotation marks unless they contain embedded blanks or commas.</p> <p>THEN</p> <p>Is an optional keyword that increases readability of the command.</p> <p>ELSE GOTO</p> <p>Optionally passes control to <i>label2</i> when the -IF test fails.</p> <p>ELSE IF</p> <p>Optionally specifies a compound -IF test.</p> <p>;</p> <p>Is required at the end of the command.</p> <p>-</p> <p>Must begin continuation lines.</p>

<b>Command:</b>	<b>-IF</b>
<b>Function:</b>	<p>-IF routes execution of a stored procedure based on the evaluation of the specified expression.</p> <p>An -IF without an explicitly specified ELSE whose expression is false continues processing with the line immediately following it.</p>

<b>Command:</b>	<b>-INCLUDE</b>
<b>Syntax:</b>	<p><i>-INCLUDE filename</i></p> <p>where:</p> <p><i>filename</i></p> <p>Is the name of the called stored procedure.</p>
<b>Function:</b>	<p>-INCLUDE enables one stored procedure to call another one.</p> <p>A stored procedure calls any number of other procedures. Up to four -INCLUDE commands are nested.</p> <p>The called procedure contains fully executable or partial code.</p> <p>The calling procedure cannot branch to a label in the called procedure and vice versa.</p>

<b>Command:</b>	<b>-label</b>
<b>Syntax:</b>	<p><i>-label [TYPE message]</i></p> <p>where:</p> <p><i>label</i></p> <p>Is a user-supplied name of up to 12 characters that identifies the target for a branch.</p> <p>Do not use embedded blanks or the name of any other Dialogue Manager command except -QUIT or -EXIT. Do not use words that may be confused with functions, arithmetic or logical operations, and so on.</p> <p><i>TYPE message</i></p> <p>Optionally sends a message to the client application.</p>
<b>Function:</b>	A label is the target of a -GOTO or -IF command.

<b>Command:</b>	<b>-QUIT</b>
<b>Syntax:</b>	<code>-QUIT</code>
<b>Function:</b>	<p>-QUIT forces an immediate exit from a stored procedure. Stacked commands are not executed. In this respect, -QUIT is different from -EXIT, which executes stacked commands.</p> <p>If the procedure was called by another one, control returns directly to the client application, not to the calling procedure.</p> <p>-QUIT is the target of a branch.</p>

<b>Command:</b>	<b>-READ</b>
<b>Syntax:</b>	<p><code>-READ filename[,] [NOCLOSE] &amp;name[.format.][,]...</code></p> <p>where:</p> <p><code>filename[,]</code></p> <p>Is the name of an external file to read, which must be defined to the operating system. A space after <i>filename</i> denotes a fixed-format file, while a comma after <i>filename</i> denotes a free-format file.</p> <p><b>NOCLOSE</b></p> <p>Optionally keeps the external file open until the -READ operation is complete. Files kept open with NOCLOSE are closed by using the command -CLOSE file name.</p> <p><code>&amp;name[,]...</code></p> <p>Is a list of variables. For free-format files, you may separate the variable names with commas, but it is not necessary.</p> <p><code>.format.</code></p> <p>Is the format of the variable. For free-format files, you do not have to define the length of the variable, but you may. For fixed-format files, the format specifies the length or the length and type of the variable (A is the default type). The value of format must be delimited by periods.</p>

<b>Command:</b>	<b>-READ</b>
<b>Function:</b>	<p>-READ enables the reading of data from an external file that is defined to the operating system.</p> <p>The length of the variable list must be known before the -READ command is encountered. Use a -DEFAULTS command to establish the number of characters expected for each variable.</p> <p>If the list of variables is longer than one line, end the first line with a comma and begin the next line with a hyphen if you are reading a free-format file.</p> <pre>-READ EXTFILE, &amp;CITY, &amp;CODE1, - &amp;CODE2</pre> <p>If you are reading a fixed-format file, begin the next line with a hyphen and comma.</p> <pre>-READ EXTFILE &amp;CITY.A8. &amp;CODE1.A3., -, &amp;CODE2.A3.</pre>

<b>Command:</b>	<b>-REMOTE</b>
<b>Syntax:</b>	<pre>-REMOTE [BEGIN END]</pre> <p>where:</p> <p><b>BEGIN</b></p> <p>Specifies the start of commands on an originating server to be sent to a target server.</p> <p><b>END</b></p> <p>Specifies the end of commands from the originating server.</p>
<b>Function:</b>	<p>-REMOTE commands are the initial form of stored procedure routing, and are available with Hub Services only.</p> <p>Dialogue Manager commands within the delimiters are executed, and variable substitution takes place before the stack is sent to the target server. An -INCLUDE command takes a Dialogue Manager procedure residing on the originating server and includes the procedure commands in the stack.</p> <p>The commands within the delimiters must make up a complete server request. Any command valid on the target server is included.</p> <p>The command EXEC may be included within the delimiters to execute a stored procedure on the target server.</p> <p>-REMOTE commands cannot be nested.</p>

<b>Command:</b>	<b>-REPEAT</b>
<b>Syntax:</b>	<p><code>-REPEAT <i>label</i> <i>n</i> TIMES</code></p> <p>or</p> <p><code>-REPEAT <i>label</i> WHILE <i>condition</i></code></p> <p>or</p> <p><code>-REPEAT <i>label</i> FOR <i>&amp;variable</i> [FROM <i>fromval</i>] [TO <i>toval</i>] [STEP <i>s</i>]</code></p> <p>where:</p> <p><i>label</i></p> <p>Identifies the code to be repeated (the loop). A label includes another loop if the label for the second loop has a different name from the first.</p> <p><i>n</i> TIMES</p> <p>Specifies the number of times to execute the loop. The value of <i>n</i> is a local variable, a global variable, or a constant. If it is a variable, it is evaluated only once, so the only way to end the loop early is with -QUIT or -EXIT (you cannot change the number of times to execute the loop).</p> <p>WHILE <i>condition</i></p> <p>Specifies the condition under which to execute the loop. The condition is any logical expression that is either true or false. The loop is run if the condition is true.</p> <p>FOR <i>&amp;variable</i></p> <p>Is a variable that is tested at the start of each execution of the loop. It is compared with the value of <i>fromval</i> and <i>toval</i> (if supplied). The loop is executed only if <i>&amp;variable</i> is less than or equal to <i>toval</i> (STEP is positive), or greater than or equal to <i>toval</i> (STEP is negative).</p> <p>FROM <i>fromval</i></p> <p>Is a constant that is compared with <i>&amp;variable</i> at the start of each execution of the loop. The default value is 1.</p> <p>TO <i>toval</i></p> <p>Is a value against which <i>&amp;variable</i> is tested. The default is 1,000,000.</p> <p>STEP <i>s</i></p> <p>Is a constant used to increment <i>&amp;variable</i> at the end of each execution of the loop. It may be positive or negative. The default value is 1.</p>

<b>Command:</b>	<b>-REPEAT</b>
<b>Function:</b>	<p>-REPEAT allows looping in a stored procedure.</p> <p>The parameters FROM, TO, and STEP appear in any order.</p> <p>A loop ends when:</p> <ul style="list-style-type: none"> <li>• It is executed in its entirety.</li> <li>• A -QUIT or -EXIT is issued.</li> <li>• A -GOTO is issued to a label outside of the loop. If a -GOTO is later issued to return to the loop, the loop proceeds from the point it left off.</li> </ul>

<b>Command:</b>	<b>-RUN</b>
<b>Syntax:</b>	<i>-RUN</i>
<b>Function:</b>	<p>-RUN causes immediate execution of all stacked commands.</p> <p>Following execution, processing of the stored procedure continues with the line that follows -RUN.</p> <p>-RUN is commonly used to:</p> <ul style="list-style-type: none"> <li>• Generate results from an SQL request that are then used in testing and branching.</li> <li>• Close an external file opened with -READ or -WRITE. When a file is closed, the line pointer is placed at the beginning of the file for a -READ. The line pointer for a -WRITE is positioned depending on the allocation and definition of the file.</li> </ul>

<b>Command:</b>	<b>-SET</b>
<b>Syntax:</b>	<p><i>-SET &amp;[&amp;]name=expression;</i></p> <p>where:</p> <p><i>&amp;name</i></p> <p>Is the name of a variable whose value is to be set.</p> <p><i>expression</i></p> <p>Is a valid expression. Expressions occupy several lines, so end the command with a semicolon.</p>

<b>Command:</b>	<b>-SET</b>
<b>Function:</b>	<p>-SET assigns a literal value to a variable, or a value that is computed in an arithmetic or logical expression.</p> <p>Single quotation marks around a literal value are optional unless it contains embedded blanks or commas, in which case the quotation marks must be included.</p>

<b>Command:</b>	<b>-TSO RUN</b>
<b>Syntax:</b>	<p><i>-TSO RUN command</i></p> <p>where:</p> <p><i>command</i></p> <p>Is a TSO command.</p>
<b>Function:</b>	-TSO executes a TSO operating system command from a procedure.

<b>Command:</b>	<b>-TYPE</b>
<b>Syntax:</b>	<p><i>-TYPE text</i></p> <p>where:</p> <p><i>text</i></p> <p>Is a message that is sent to a client application, followed by a line feed. Quotation marks will be displayed as part of the message if included around text.</p> <p>The length of text can be up to 256 bytes.</p>
<b>Function:</b>	<p>-TYPE sends a message to a client application.</p> <p>Any number of -TYPE commands can follow one another but each must begin with -TYPE.</p> <p>Variables may be embedded in the message. The values currently assigned to each variable are displayed.</p>

<b>Command:</b>	<b>-UNIX</b>
<b>Syntax:</b>	<p><i>-UNIX command</i></p> <p>where:</p> <p><i>command</i></p> <p>Is a UNIX command.</p>

<b>Command:</b>	<b>-UNIX</b>
<b>Function:</b>	-UNIX executes a UNIX operating system command from a procedure.

<b>Command:</b>	<b>-VMS</b>
<b>Syntax:</b>	<p><i>-VMS command</i></p> <p>where:</p> <p><i>command</i></p> <p>Is a VMS command.</p>
<b>Function:</b>	-VMS executes a VMS operating system command from a procedure.

<b>Command:</b>	<b>-WINNT</b>
<b>Syntax:</b>	<p><i>-WINNT command</i></p> <p>where:</p> <p><i>command</i></p> <p>Is a Windows or DOS command.</p>
<b>Function:</b>	-WINNT executes a Windows or DOS operating system command from a procedure.

<b>Command:</b>	<b>-WRITE</b>
<b>Syntax:</b>	<p><i>-WRITE filename [NOCLOSE] text</i></p> <p>where:</p> <p><i>filename</i></p> <p>Is a symbolic name for a physical external file being written to. The file name must be known to the operating system.</p> <p><b>NOCLOSE</b></p> <p>Keeps the external file open until the -WRITE operation is complete. Files kept open with NOCLOSE are closed with the command -CLOSE filename.</p> <p><i>text</i></p> <p>Is any combination of variables and text.</p>

<b>Command:</b>	<b>-WRITE</b>
<b>Function:</b>	<p>-WRITE writes data to an external file.</p> <p>If the command continues over several lines, put a comma at the end of the line and a hyphen at the beginning of each succeeding line.</p> <p>Unless you specify the NOCLOSE option, an opened file is closed upon termination of the procedure with -RUN, -EXIT, or -QUIT.</p>



---

---

## APPENDIX B

### GENCPGM Usage

**Topic:**

- Using GENCPGM

The building and compilation of 3GL applications is platform-specific and sometimes driven by standards with which a site must conform in terms of programming style or managing programming source. Due to this wide variation, we only make recommendations, test certain languages, and provide limited examples with a script that minimally compiles the test examples.

The specific uses for 3GL programs and examples are documented elsewhere, but the general purposes are:

- To add a user written routine to the functions of the product (also know as a FUSELIB).
- To customize user exits that provide special functions.
- To create CALLPGM programs that the server executes.
- To create an API program to converse with a server.

## Using GENCPGM

---

A script has been created for UNIX, Windows NT/2000, and OpenVMS to assist in simple compilations of the provided C examples. On the respective platforms, the names are gencpgm.sh, gencpgm.bat, and gencpgm.com. The script is located in the bin directory of EDAPATH.

The basic function of GENCPGM is to either:

- Create a program similar to an API application that is run.  
or
- Create a dynamically loadable library program that may be accessed by other programs. Dynamically loadable library program type programs are known as .dll on Windows, .exe shared object images on OpenVMS, .so or .sl shared object libraries on UNIX and Service Programs on OS/400.

There is no specific requirement that GENCPGM be used in program creation, only that a given program be a properly compiled and linked program.

On some platforms, the GENCPGM script supports languages other than C and may have had some level of testing for these other languages in the past. While theoretically any language is used, C is the only officially supported language for these platforms as it is universally and readily available and testable. Support for using other languages is only extended as far as reviewing cases when language compilers or expertise in the language is not available.

The GENCPGM script is written for simple compilation cases. Complex cases such as multiple sources, including library locations, ordering of libraries, special compilers, and linker options are not handled, however, the GENCPGM script may be used as a model.

### **Procedure** How to Compile and Link a Procedure

This section outlines the steps required to compile and link a procedure:

1. Copy GENCPGM from the EDAPATH bin directory to your working directory.
  - For an API program or to build the sample API program (EDAAPP), copy EDA.H, EDASYS.H, and EDAAPP.\* (the sample program) from the etc directory of EDAPATH to your working directory.
  - For a CALLPGM program or to build the CALLPGM sample program (CPT), copy CPT.C (the sample program) and CPGUB.H from the etc directory of EDAPATH to your working directory.
  - For user exits, copy the desired sample exit from the etc directory of EDAPATH to your working directory.

- For user routines, write the routine or copy an existing routine to your working directory.
2. Issue an environment variable for the EDAPHOME directory.
  3. If building an API program, also issue an environment variable for the EDACONF directory.

## **Syntax**    **How to Run GENCPGM**

```
gencpgm [-g] [-q] [-x] [-m [ option ]][-c [ option ]] prog
```

where:

*prog*

Is the name of the procedure to be compiled and linked.

**Note:** OpenVMS also requires a leading “@”:



## **Syntax**    **How to Display GENCPGM Options**

Display the full GENCPGM options available for a given platform by issuing:

```
gencpgm -?
```

## **Reference**    **GENCPGM Parameters**

The following table explains the GENCPGM parameters, parameter options, and usage.

Parameter	Use	Possible Values
-g	An optional parameter, which creates a debug version of the module.	No value needed.
-q	Suppresses output messages from platforms that have messages.	No value needed.
-x	Activates <code>set -x</code> for shell tracing (UNIX Only).	No value needed.

Parameter	Use	Possible Values	
<code>-m</code>	Specifies the type of module to create. If this parameter is omitted, the default is <code>-m cpgm</code> .	<code>api</code>	Creates an API program.
		<code>cpgm</code>	Creates a CALLPGM program, a user exit, or a routine.
<code>-c</code>	Specifies the type of compiler to use. If this parameter is omitted, the default will be <code>-c cc</code> .	<code>cc</code>	Uses the standard C compiler to compile <code>prog.c</code> .
		<code>CC</code>	Uses the C++ compiler to compile <code>prog.cpp</code> . (UNIX)
		<code>cxx or c++</code>	Uses the C++ compiler to compile <code>prog.cpp</code> . (OpenVMS only)
		<code>gcc</code>	GNU C compiler
		<code>for</code>	Uses the Fortran compiler to compile <code>prog.f</code> . (OpenVMS only)
		<code>cob</code>	Uses the Cobol compiler to compile <code>prog.cob</code> . (OpenVMS only)

After running GENCPGM, an executable for the program will be created in the working directory. If the compilation was for an API program, a "helper" script with the same name is also created containing start up variables and program initialization information.

If the compilation was for an API program and it needs to be moved to another directory, then the helper script should also be moved and edited to account for the new location in any of its variables.

If the compilation was for CALLPGM, a user exit, or a routine, the final step is to either copy the resulting routine to the user directory of EDACONF or set the environment variable IBICPG to the name of the actual working directory (and restart the server). This final step puts the resulting routine in a path that the server searches for routines at run time.

### **Example**    **Generating an API Program From a C Source File**

Because the Standard C compiler and API mode are default options, the following example will generate an API program from a C source file named `myprog.c` using the standard C compiler.

```
gencpgm myprog
```

### **Example**    **Generating an API Program From a C++ Source File**

Because the Standard C compiler and API mode are default options, the following example will generate a debuggable API program from a sample C++ source file named `edaapp.cpp` using the C++ compiler.

```
gencpgm -g -c C++ edaapp
```

### **Example**    **Generating a CALLPGM Program Library From a C Source File**

The following example will generate a debuggable `callpgm` program library from a C source code file named `myprog.c` using the standard C compiler.

```
gencpgm -g -m cpgm myprog"
```

### **Example**    **Using GENCPGM With the MTHNAME Routine**

The following example is a routine that is used in Dialogue Manger to translate a month number into a spelled out name.

**mthname.c**

```
mthname(mth,month)
double *mth;
char *month;
{
static char *nmonth[13] = {"** Error **",
                           "January   ",
                           "Febuary  ",
                           "March    ",
                           "April   ",
                           "May     ",
                           "June    ",
                           "July    ",
                           "August  ",
                           "September ",
                           "October  ",
                           "November ",
                           "December ",};

int imth, loop;
imth = (int)*mth;
imth = (imth < 1 || imth > 12 ? 0:imth);
for (loop=0;loop < 12;++loop)
    month[loop] = nmonth[imth][loop];
}
```

### **mthname.fex**

```
-SET &MTHNAME = MTHNAME(&MTHNUMBER,'A12') ;
-TYPE Month &MTHNUMBER is &MTHNAME
```

Compile and set IBICPG (this is an example on UNIX):

```
gencpgm -m cpgm myprog"
export IBICPG=`pwd`
```

After restarting the server, execute an RPC like:

```
EX MTHNAME MTHNUMBER=4
```

And receive:

```
Month 4 is March
```

---

---

# Index

## Symbols

---

! command 4-68  
&&name variable 4-13  
&DATE variable 4-17, 4-19  
&DATEfmt variable 4-17  
&DMY variable 4-17  
&DMYY variable 4-17  
&ECHO variable 4-25  
&FOCFOCEXEC variable 4-17  
&FOCINCLUDE variable 4-17  
&FOCMODE variable 4-17  
&FOCNET variable 4-17  
&FOCPRINT variable 4-17  
&FOCREL variable 4-17  
&IORETURN variable 4-17  
&MDY variable 4-17  
&MDYY variable 4-17  
&name variable 4-13  
&RETCODE variable 4-17  
&TOD variable 4-17  
&YMD variable 4-17  
&YYMD variable 4-17  
-\* command 4-3, A-2  
-? command 4-3, 4-16, A-2  
? EXORDER command 1-5

## Numerics

---

3GL programs 3-1, B-1  
requirements 3-2

## A

---

ABS function 4-56–4-57  
ALLOCATE subcommand 6-2, 6-5  
allocating data sets 6-5  
allocating dynamic storage 3-19  
alphanumeric expressions 4-50–4-51, 4-54  
AND keyword 4-53  
answer sets 2-7  
    returning 3-9, 3-11, 3-14, 4-69–4-70  
API parameters 2-8  
ARGLEN function 4-57  
arithmetic expressions 4-49–4-50  
ASIS function 4-57  
ATODBL function 4-57  
AYM function 4-57  
AYMD function 4-57

## B

---

BAR function 4-57  
BITSON function 4-57  
BITVAL function 4-57  
branching 4-31–4-34  
BYTVAL function 4-57

## C

---

- C programming language B-2
- CALLIMS LU6.2 Adapter for IMS/TM 5-1–5-2, 5-15
  - installing 5-11–5-12
  - returning data 5-5
  - running 5-4, 5-6–5-7, 5-10
  - transaction processing 5-2–5-3, 5-5
- CALLIMS procedure 1-6
- calling procedures 4-41, 4-43
- calling programs 2-2–2-5
- calling stored procedures 1-2
  - Dialogue Manager 1-6
- CALLITOC OTMA Adapter for IMS/TM 5-1–5-2, 5-15
  - installing 5-11, 5-14
  - returning data 5-5
  - running 5-4, 5-6, 5-9, 5-11
  - transaction processing 5-2, 5-4–5-5
- CALLITOC program 1-6
- CALLPGM command 1-6, 2-2, 2-4–2-5, 3-9
  - DB2 plans 2-6
- CHGDAT function 4-57
- CHKFMT function 4-57
- CHKPCK function 4-57
- CLOSE command 4-3, A-2
- CLOSE subcommand 6-2, 6-16
- CMD command 4-3, 4-68, A-3
- CMS command 4-3, 4-68, A-3
- CNTCTUSR function 4-57
- command lines 4-25
- commands 4-2
  - delimiting 4-45
  - variables and 4-20
- comments 4-7
- communicating between the server and the program 2-13
- compiled programs 2-1, 3-1
  - calling 2-2–2-5
  - libraries 1-4
  - requirements 3-2
  - running 3-18–3-19
- compound expressions 4-55
- COMPRESS subcommand 6-2, 6-23
- CONCAT subcommand 6-2, 6-14
- concatenating files 6-24
- concatenation 4-51
- conditional branching 4-33–4-34
- CONTAINS operator 4-53
- control block fields 3-3, 3-8–3-9
- control blocks 2-13, 3-2–3-3, 3-9, 3-18
- COPY subcommand 6-2, 6-17
- COPYDD subcommand 6-19
- CPG parameters 2-8
- CPGVMS.COBI program 3-18
- CREATE TABLE command 3-9, 3-21
- creating data sets 6-24
- creating expressions 4-48
- creating variables 4-15
- CTRAN function 4-57
- CTRFLD function 4-57
- Customer Support Service 1-v
  - How to contact 1-v
  - Information required 1-vi

**D**

- 
- DADMY function 4-57
  - DADYM function 4-57
  - DAMDY function 4-57
  - DAMYD function 4-57
  - data sets 6-2
    - allocating 6-5
    - creating 6-24
    - locking 6-4
  - date fields 4-52
  - DATEADD function 4-57
  - DATECVT function 4-57
  - DATEDIF function 4-57
  - DATEMOV function 4-57
  - DATERPT command 4-44
  - DAYDM function 4-57
  - DAYMD function 4-57
  - DB2 plans 2-6
  - ddnames (logical names) 6-25–6-26
  - DECODE function 4-66–4-67
  - DEFAULTS command 4-3, 4-25–4-26, A-3
  - DELETE subcommand 6-2, 6-20
  - Dialogue Manager commands 4-3, 4-8–4-9, 6-1, A-1
    - ! command 4-68
    - \* command 4-3, A-2
    - ? command 4-3, A-2
    - CLOSE 4-3, A-2
    - CMD 4-3, 4-68, A-3
    - CMS 4-3, 4-68, A-3
    - DEFAULTS 4-3, 4-25–4-26, A-3
    - DOS 4-68, A-4
    - EXIT 4-3, 4-10, 4-39, A-4
    - GOTO 4-3, 4-31–4-32, 4-39, A-4
    - IF 4-3, 4-33–4-38, A-5
    - INCLUDE 4-3, 4-41, 4-43, A-6
    - label 4-3, A-6
    - QUIT 4-3, 4-11, 4-39, A-7
    - READ 4-3, 4-28, 4-30, A-7
    - REMOTE 4-45, A-8
    - REMOTE BEGIN 4-3, 4-44
    - REMOTE END 4-3, 4-44
    - REPEAT 4-3, 4-38, 4-40, A-9
    - RUN 4-3, 4-9, A-10
    - SET 4-3, 4-27–4-28, 5-6–5-7, 5-9–5-11, A-10
    - TSO RUN 4-3, 4-68, A-11
    - TYPE 4-3, 4-8, A-11
    - UNIX 4-3, 4-68, A-12
    - VMS 4-3, 4-68, A-12
    - WINNT 4-68, A-12
    - WRITE 4-3, 4-45, 4-48, A-13
    - command 4-3
  - Dialogue Manager procedures 1-3–1-4, 1-6, 2-2, 2-4, 4-2, 4-8
    - calling 4-6
    - creating 4-7
    - ETPCIMS command 5-21
    - platform-specific commands 6-1
    - running 4-5, 5-12, 5-14
    - variables and 4-12–4-14
  - displaying GENCPGM options B-3
  - displaying variable values 4-16
  - DMOD function 4-57
  - DMY function 4-57
  - DOS command 4-68, A-4
  - DOWK function 4-57
  - DOWKL function 4-57
  - DTDMY function 4-57
  - DTDYM function 4-57

## Index

DTMDY function 4-57  
DTMYD function 4-57  
DTYDM function 4-57  
DTYMD function 4-57  
DYNAM command 6-2, 6-4, 6-15, 6-23–6-24  
DYNAM subcommands 6-2  
    ALLOCATE 6-2, 6-5  
    CLOSE 6-2, 6-16  
    COMPRESS 6-2, 6-23  
    CONCAT 6-2, 6-14  
    COPY 6-2, 6-17  
    COPYDD 6-19  
    DELETE 6-2, 6-20  
    FREE 6-2, 6-15  
    RENAME 6-2, 6-21  
    SUBMIT 6-2, 6-22  
dynamic storage 3-9, 3-11, 3-14, 3-17  
    allocating 3-19

## E

---

EDAACCEPT method call 5-5  
EDAFETCH method call 3-21, 5-18  
EDAINFO method call 3-21  
EDAPATH method call 1-4  
EDARPC command 2-8  
EDARPC method call 1-2, 1-4, 2-2–2-3, 4-6  
EDIT function 4-65–4-66  
environment parameters 5-18  
    IMSDELIMITER 5-18  
    IMSRECORDLENGTH 5-18  
    IOTRACE 5-18  
environment variables 5-21  
    LD\_LIBRARY\_PATH 5-22  
    SHLIB\_PATH 5-22  
EQ operator 4-53

error processing 3-19  
ETPCIMS LU6.2 Adapter for IMS/TM 5-1, 5-16, 5-21  
    configuring 5-21–5-22  
    environment parameters 5-18  
    installing 5-21  
    returning data 5-18  
    running 5-20  
    transaction processing 5-16  
EVAL operator 4-46–4-47  
EXEC command 1-6, 2-2, 2-5, 4-21–4-22, 4-41, 4-44  
execution flow 4-9–4-11, 4-25, 4-31  
    debugging 4-25  
execution order 1-3, 1-6  
    Dialogue Manager 1-6  
    querying 1-5  
    setting 1-5  
-EXIT command 4-3, 4-10, 4-39, A-4  
EXORDER command 1-5  
EXP function 4-57  
EXPN function 4-57  
expressions 4-48–4-55  
external files 4-45–4-46, 4-48

## F

---

FEXERR function 4-57  
FGETENV function 4-57  
file attributes 6-25  
    specifying 6-25–6-26  
FILEDEF command 6-25–6-27  
files  
    concatenating 6-24  
    freeing 6-24

FINDMEM function 4-57  
 flow of execution 4-9–4-11, 4-25, 4-31  
   debugging 4-25  
 FMOD function 4-57  
 FORECAST function 4-57  
 FPUTENV function 4-57  
 FREE command 6-15  
 FREE subcommand 6-2, 6-15  
 freeing dynamic storage 3-19  
 freeing files 6-24  
 FTOA function 4-57  
 functions 4-56–4-57  
   expressions and 4-56

**G**

---

GE operator 4-53  
 GENCPGM parameters B-4  
 GENCPGM script B-1–B-2, B-6–B-7  
   displaying options B-3  
   running B-3  
 generating API programs B-6  
 generating CALLPGM programs B-6  
 GETPDS function 4-57  
 GETSECID function 4-57  
 GETTOK function 4-57  
 GETUSER function 4-57  
 global variables 4-12, 4-16  
   naming 4-12–4-13  
 -GOTO command 4-3, 4-31–4-32, 4-39, A-4  
 GREGDT function 4-57  
 GT operator 4-53

## H

---

HADD function 4-57  
 HCNVRT function 4-57  
 HDATE function 4-57  
 HDIFF function 4-57  
 HDTTM function 4-57  
 HEXBYT function 4-57  
 HGETC function 4-57  
 HHMMSS function 4-57  
 HINPUT function 4-57  
 HMIDNT function 4-57  
 HNAME function 4-57  
 HPART function 4-57  
 HSETPT function 4-57  
 HTIME function 4-57

## I

---

IBICPG library 1-4  
 -IF command 4-3, 4-33–4-38, A-5–A-6  
 IMOD function 4-57  
 IMS transactions 5-18  
 IMS/TM transactions 1-4, 5-1–5-2  
   running 5-21  
 IMSDELIMITER parameter 5-18  
 IMSRECORDLENGTH parameter 5-18  
 -INCLUDE command 4-3, 4-41–4-43, A-6  
 indexed variables 4-15  
 installing CALLIMS Adapter for IMS/TM 5-12  
 installing CALLITOC Adapter for IMS/TM 5-14  
 installing ETPCIMS LU6.2 Adapter for IMS/TM 5-21

## Index

installing OTMA Adapter for IMS/TM 5-14

INT function 4-56–4-57

IOTRACE parameter 5-18

ITONUM function 4-57

ITOPACK function 4-57

ITOZ function 4-57

## J

---

JCL commands 6-24

JULDAT function 4-57

## K

---

keyword parameters 2-8, 4-23  
    passing 2-10–2-11, 4-22

## L

---

-label command 4-3, A-6

LCWORD function 4-57

LD\_LIBRARY\_PATH variable 5-22

LE operator 4-53

libraries 1-4

LJUST function 4-57

local variables 4-12–4-14, 4-16  
    naming 4-12–4-13

LOCASE function 4-57

locking data sets 6-4

LOG function 4-56–4-57

logical expressions 4-52–4-54

logical names 6-25–6-26

logical operators 4-53

long parameters 2-12  
    passing 4-24

looping 4-38–4-40

## M

---

MAX function 4-56–4-57

MDY function 4-57

message codes 5-5  
    viewing 5-6

message delivery 4-8

messages 3-9, 5-5  
    converting 5-15  
    retrieving 5-5  
    returning 3-11, 3-14

MIN function 4-56–4-57

MTHNAME routine B-7

multiple messages 5-15

multi-threaded programs 3-9

MVS commands 6-2

MVS DYNAM function 4-57

## N

---

naming variables 4-12–4-13

NE operator 4-53

nesting 4-43

NOT operator 4-53

## O

---

OFFLINE command 6-27

OMITS operator 4-53

ON TABLE HOLD 4-69

ON TABLE PCHOLD 4-69–4-70

operating system commands 4-68

operators 4-53  
    CONTAINS 4-53

EQ 4-53  
 EVAL 4-46–4-47  
 GE 4-53  
 LE 4-53  
 NOT 4-53  
 OMIT 4-53  
 OR 4-53  
 OR operator 4-53  
 order of execution 1-3, 1-6  
     Dialogue Manager 1-6  
     querying 1-5  
     setting 1-5  
 OTMA Adapter for IMS/TM 5-1–5-2, 5-15  
     installing 5-11, 5-14  
     returning data 5-5  
     running 5-4, 5-6, 5-9, 5-11  
     transaction processing 5-2, 5-4–5-5  
 overriding default variable values 4-26  
 OVLAY function 4-57  
**P**  


---

 PARAG function 4-57  
 parameters 2-7, 4-23  
     passing 2-8–2-10, 2-12, 4-22–4-24  
 passing parameters 2-7–2-12, 4-22–4-24  
 PCKOUT function 4-57  
 plans for DB2 2-6  
 platform-specific commands 6-1  
 POSIT function 4-57  
 positional parameters 2-8, 4-23  
     passing 2-9, 2-11, 4-23  
 PRDNOR function 4-57  
 PRDUNI function 4-57  
 procedure libraries 1-4

procedures 1-1  
     calling 4-41, 4-43  
     compiling B-3  
     creating 4-46  
     exiting 4-10  
     linking B-3  
     testing 4-25  
 program libraries 1-4  
 program values 3-9, 3-11, 3-14, 3-17  
     storing 3-19  
 program variables 3-18  
 program-to-server communication 2-13

## Q

---

-QUIT command 4-3, 4-11, 4-39, A-7  
 quotes 2-8

## R

---

RDNORM function 4-57  
 RDUNIF function 4-57  
 -READ command 4-3, 4-28, 4-30, A-7  
 reading variable values 4-28, 4-30  
 -REMOTE BEGIN command 4-3, 4-44  
 -REMOTE command 4-45, A-8  
 -REMOTE END command 4-3, 4-44  
 RENAME subcommand 6-2, 6-21  
 -REPEAT command 4-3, 4-38, 4-40, A-9  
 retrieving messages 5-5  
 REVERSE function 4-57  
 RJUST function 4-57  
 -RUN command 4-3, 4-9, A-10

## S

---

server settings for ETPCIMS LU6.2 Adapter for IMS/  
TM 5-21

- SET command 4-3, 4-27–4-29, 5-6–5-7, 5-9–  
5-11, A-11
- SET EXORDER command 1-3
- SHLIB\_PATH environment variable 5-22
- SOUNDEX function 4-57
- specifying variable length 4-29
- SPELLNUM function 4-57
- SQRT function 4-56–4-57
- stacked commands 4-5, 4-10
  - canceling 4-11
  - running 4-9
- statistical variables 4-12
- stored procedure libraries 1-4, 2-2
- stored procedures 1-1–1-2, 3-1, 5-15
  - calling 1-2, 2-1
  - canceling 4-11
  - exiting 4-10
  - requirements 3-2
  - running 1-3, 1-5–1-6, 2-7
- stored values 3-9, 3-11, 3-14, 3-17
- storing program values 3-9, 3-11, 3-14, 3-17, 3-  
19
- STRIP function 4-57
- stripping quotes from parameters 2-8
- SUBMIT subcommand 6-2, 6-22
- SUBSTR function 4-57
- supplying variable values 4-20–4-22, 4-25–4-30
- system variables 4-12, 4-17, 4-19
- system-supplied functions 4-56–4-57

## T

---

TEMPPATH function 4-57

- testing for variable values 4-35–4-38

TODAY function 4-57

- transaction adapters 5-2
- transaction processing 5-2, 5-5
  - CALLIMS Adapter 5-2–5-3
  - CALLITOC Adapter 5-2, 5-4
  - ETPCIMS LU6.2 Adapter 5-16–5-17
  - OTMA Adapter 5-4

TRIM function 4-57

TRUNCATE function 4-57

TSO commands 6-23–6-24

- TSO RUN command 4-3, 4-68, A-11
- TYPE command 4-3, 4-8, A-11

## U

---

UFMT function 4-57

- unconditional branching 4-31–4-32

- UNIX command 4-3, 4-68, A-12

UPCASE function 4-57

- user exit routines 6-4

## V

---

values 4-65

- decoding 4-66–4-67

- variable values 4-20–4-22
  - displaying 4-16
  - overriding 4-26
  - setting 4-25–4-30
  - testing 4-35–4-38
- variables 4-12–4-14, 4-16–4-17
  - commands and 4-20
  - creating 4-15

naming 4-12–4-13

-VMS command 4-3, 4-68, A-12

## **W**

---

-WINNT command 4-68, A-12

-WRITE command 4-3, 4-45–4-46, 4-48, A-13

## **Y**

---

YM function 4-57

YMD function 4-57

## **Z**

---

command 4-3



---

---

## Reader Comments

In an ongoing effort to produce effective documentation, the Documentation Services staff at Information Builders welcomes any opinion you can offer regarding this manual.

Please use this form to relay suggestions for improving this publication or to alert us to corrections. Identify specific pages where applicable. You can contact us through the following methods:

**Mail:** Documentation Services - Customer Support  
Information Builders, Inc.  
Two Penn Plaza  
New York, NY 10121-2898

**Fax:** (212) 967-0460

**E-mail:** [books\\_info@ibi.com](mailto:books_info@ibi.com)

**Web form:** <http://www.informationbuilders.com/bookstore/derf.html>

Name: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_

Telephone: \_\_\_\_\_ Date: \_\_\_\_\_

E-mail: \_\_\_\_\_

Comments:

---

---

## **Reader Comments**