iWay

iWay Stored Procedures Reference
Version 5 Release 3.2

DN3501293.0904

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	Introducing Stored Procedures	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 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	Calling a JAVA Class	Describes ways to call a JAVA class using the CALLJAVA command or the EX command.
4	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.
5	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
6	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.
A	Dialogue Manager Quick Reference	Includes all Dialogue Manager commands, with their syntax, in alphabetical order for easy reference.
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
`THIS TYPEFACE` or `this typeface`	Denotes syntax that you must enter exactly as shown.
`this typeface`	Represents a placeholder (or variable) in syntax for a value that you or the system must supply.
underscore	Indicates a default setting.
this typeface	Represents a placeholder (or variable), a cross-reference, or an important term. It may also indicate a button, menu item, or dialog box option you can click or select.
this typeface	Highlights a file name or command.
Key + Key	Indicates keys that you must press simultaneously.
`{ }`	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

To view a current listing of our publications and to place an order, visit our World Wide Web site, http://www.iwaysoftware.com. 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 Services (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 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. You can find your server version and release using the Version option in the Web Console. (Note: the MVS and VM servers do not use the Web Console.)
  - 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 your opinions 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

CHAPTER 1

Introducing Stored Procedures

Topics:

- Calling a Stored Procedure
- Stored Procedure Libraries
- Setting the Execution Order

A stored procedure is a program or procedure that resides on the execution path of a server. The procedure is called by a client application but can also be called or 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 (DBMS) supports and varies by platform. Any limitations will be noted in the documentation.

Stored procedures enable you to:

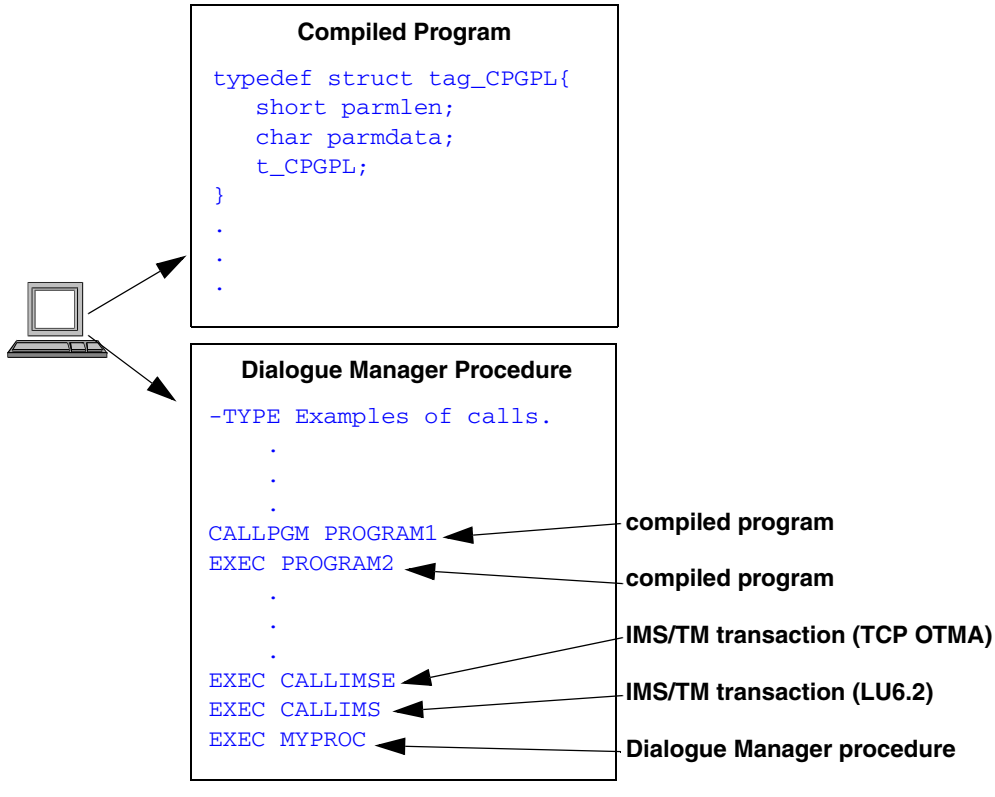
- Embed procedural logic in your server applications. The logic may be modular, eliminating the need to recreate it for each application.
- Update non-relational database management systems.

Calling a Stored Procedure

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, which can call:
  - 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 shows calls to stored procedures made from EDARPC and Dialogue Manager.



Stored Procedure Libraries

A stored procedure must reside in the appropriate library in order for the server to locate it.

Type of Stored Procedure	Library
Dialogue Manager	Server Procedure Library. The external names, EDARPC (MVS) or EDAPATH (all other platforms), are used to locate Dialogue Manager procedures. You can also use the APP PATH feature to locate and manage application code. This process is platform dependent. See the *Server Administration* manual for details.
Compiled Program	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 early practice was to place compiled procedures in the installation home bin directory/library, since it was always searched by default. This practice is no longer recommended since service pack installations will delete these types of files.
IMS/TM Transaction	Server Program Library. Underlying routines are part of the server installation home bin directory; no library configuration is required.

Note: An *external name* is a generic name for a variable that is 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.

Setting the Execution Order

In this section:

How to:

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

The server has a default search order. To change this order:

- Add 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.
- Run 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.

  If you set the execution order in a Dialogue Manager procedure before you run the procedure, make sure the execution order in effect includes a search of the Procedure Library.

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

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
`SET EXORDER=FEX`	Procedure Library only.	This setting is the default.
`SET EXORDER=PGM`	Program Library only.	
`SET EXORDER=FEX/PGM`	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 executing the program.
`SET EXORDER=PGM/FEX`	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.

Syntax: How to Query 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 explicit CALLPGM syntax 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 with no need to set EXORDER.

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 or EXEC
- Passing Parameters
- Program Communication

The following are ways to call a compiled program.

- Directly, using the EDARPC function call.
- Indirectly, using either the CALLPGM command or the EXEC command.

Any of these methods enables you 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.

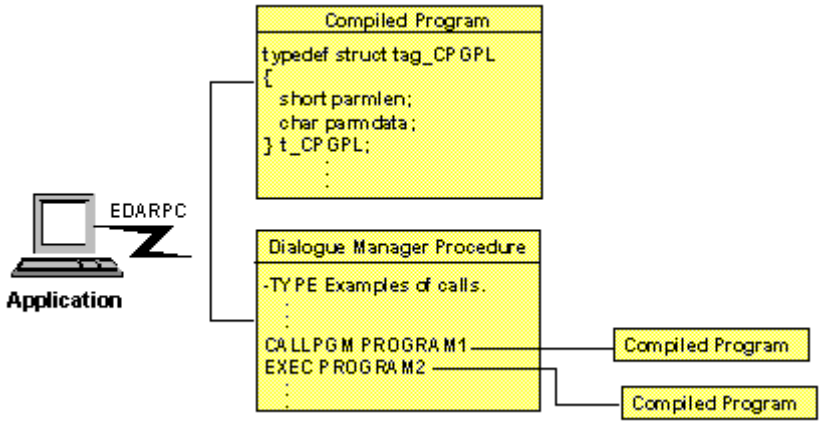
The program is called in the following ways:

- 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
  - SQL EX

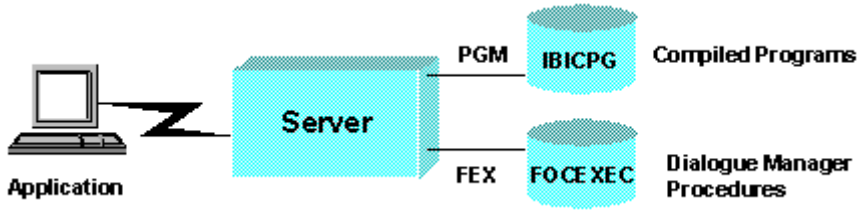
  The command EXEC functions the same way as CALLPGM, except for the difference in execution order requirements as described in *Execution Order of Stored Procedures From Dialogue Manager* in Chapter 1, *Introducing Stored Procedures*. For simplicity, this chapter refers only to CALLPGM when both CALLPGM and EXEC apply. The SQL EX method has the advantage of being able to also apply intermediate processing to the initial results set before passing the final answer set to the calling request.

The term *program* is also used to refer to a compiled program.

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

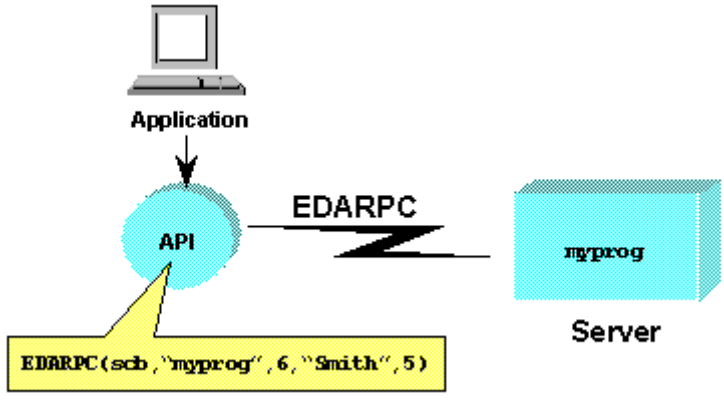


The following figure illustrates the libraries in which compiled programs and Dialogue Manager procedures reside. See Chapter 1, *Introducing Stored Procedures*, for details on stored procedure libraries and stored procedure execution order.



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 or EXEC

In this section:

Switching Plans in DB2 (MVS Only)

How to:

Call a Program From Dialogue Manager

Switch Plans in DB2

Example:

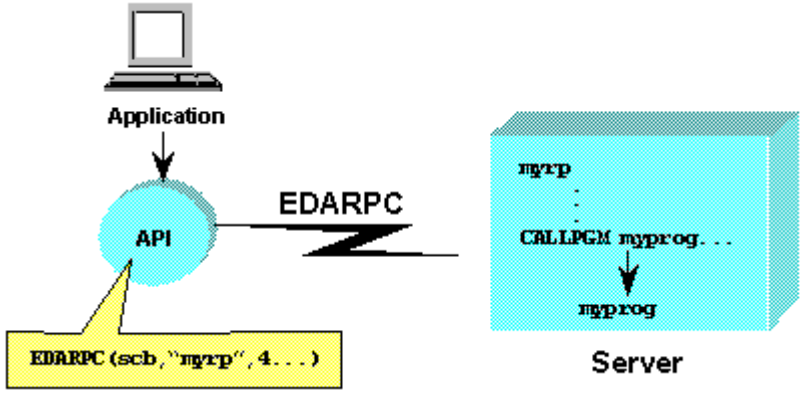
Processing an Answer Set on the Server

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 2-10 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 4, *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.

   The command CALLPGM and EXEC operates the same except EXEC has the advantage of being able to let the EXORDER setting control if FOCEXECs by the same name will be also searched for and which is considered first found (the compiled program or the FOCEXEC.)

Syntax: How to Call a Program From Dialogue Manager

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

or

```
SET EXORDER=PGM/FEX
EX[EC] progname[parmval1][,...]
END
```

or

```
SET EXORDER=PGM/FEX
SET SQLENGINE=CPGFOC
SQL EX PROGRAM [parmval1][,...] TABLE FILE SQLOUT
PRINT field [ON TABLE PCHOLD]
END
SET SQLENGINE=OFF
```

where:

progname

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 *progname*. A Dialogue Manager parameter is an alphanumeric value. See *Passing Parameters* on page 2-10 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.

Calling a Program With SQL EX

Using SQL EX is similar to using EXEC, the difference is that the output from SQL EX is stored into a HOLD file called SQLOUT. The resulting SQLOUT file can then be processed with additional SELECT or TABLE statements which may (or may not) contain additional selection criteria, and possibly return less fields or create a virtual field that is derived from the data.

Syntax: How to Call a Program From Dialogue Manager

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

or

```
SET EXORDER=PGM/FEX
EX[EC] progname[parmval1][,...]
END
```

or

```
SET EXORDER=PGM/FEX
SET SQLENGINE=CPGFOC
SQL EX PROGRAM [parmval1][,...] TABLE FILE SQLOUT
PRINT field [ON TABLE PCHOLD]
END
SET SQLENGINE=OFF
```

where:

progname

Is the name of the program to be run.

parmval1

Is an optional positional Dialogue Manager parameter passed to *progname*. A Dialogue Manager parameter is an alphanumeric value. See *Passing Parameters* on page 2-10 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

```
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 ' '

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

In this section:

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 quotation marks. The following profile setting controls the stripping of quotation marks from parameters.

Syntax: How to Control the Stripping of Quotes From Parameters

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

where:

ON

Causes the quotation marks to be stripped from the parameters. ON is the default value.

OFF

Prevents the stripping of the quotation marks from the parameters.

Using EDARPC

Example:

Passing Positional API Parameters

Passing Keyword API Parameters

Combining Positional and Keyword API Parameters

Passing Long Parameters

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 passes 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. Default 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 amper variables based on the relative position of the positional keywords to each other.

```
EDARPC(scb,"myproc",6,"prog=myprog,000001,000002,kparm1=keyparm,000003",47)
```

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.

Passing mixed positional and keyword parameters requires care in assembling the API parameter string so that the positional values match up appropriately with the underlying RPC.

Example: Passing Long Parameters

EDARPC passes a parameter up to 32,000 bytes in length. If a CALLPGM program is being executed directly, the parameter is passed directly to the CALLPGM program.

If a CALLPGM program is being executed from a procedure residing on the server, the -LINES function is used to pass long parameters to the CALLPGM program. The following is an example of a server procedure passing the maximum parameter of 32,000 bytes:

```
"EX -LINES 401 CPG32000 LINE00000000000000000000000000000000000000000001
"
"LINEOFINFORMATION11111111111111111111111111111111111111111111111111111
111111"
"LINEOFINFORMATION22222222222222222222222222222222222222222222222222222
222222"
"LINEOFINFORMATION40040040040040040040040040040040040040040040040040040
004004"
.
.
.
"LASTLINETOTAL32000BYTESTHEENDXX"
```

Note:

- The first line of data ends in column 72. The double quotation marks (“) are not part of the procedure. Quotation marks are used to indicate the beginning and end of lines, some of which may contain leading or trailing spaces.
- If any Dialogue Manager commands, such as -TYPE, are used in the procedure, the parameter is limited to 72 bytes.
- The value after -LINES is the number of lines to read for parameters. In this example, for brevity, several hundred lines are not shown.

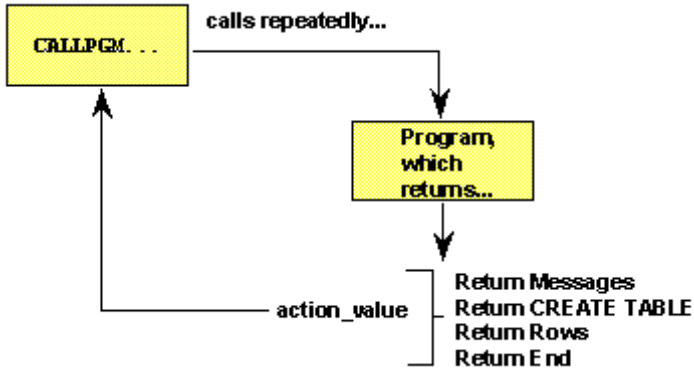
Program Communication

Whether a program is called by EDARPC or CALLPGM, a control block is used for communication between the server and the program.

The program is called repeatedly until it indicates that it is done by supplying the correct value in the field action_value in the control block on return to CALLPGM.

See Chapter 4, *Writing a 3GL Compiled Stored Procedure Program*, for more information, including the specific values to be returned in an action_value.

The process is illustrated below.



CHAPTER 3

Calling a JAVA Class

Topics:

- Using CALLJAVA
- Using EX
- Passing Parameters
- Writing a JAVA Class
- JAVA Class Communication
- Compiling and Running a JAVA Program

You can easily access a JAVA class in your application, much as you would access a program with CALLPGM. There are two ways to call a JAVA class:

- CALLJAVA call.
- EX command.

Either method enables you to pass parameters to the JAVA class.

Using CALLJAVA

You can invoke a user-written JAVA class with the CALLJAVA command.

Syntax: How to Use CALLJAVA to Execute a JAVA Class

```
CALLJAVA class, parameter1, parameter2, ...
```

where:

class

Is the full name of the class to be invoked.

parameter1, parameter2, ...

Are the remaining parameters which must be passed to the JAVA class according to the rules described in *Passing Parameters* on page 3-3.

Example: Calling iway.test.jclass Using CALLJAVA

```
SET EXORDER=PGM/FEX
CALLJAVA iway.test.jclass,parameter1,"subparm1=val1,subparm2=val2",simple
parameter3
-EXIT
```

Using EX

You can invoke a user-written JAVA class with the EX command.

Syntax: How to Use EX to Execute a JAVA Class

```
EX java.class parameter1, parameter2, ...
```

where:

java.class

Is the full name of the class to be invoked and must be preceded by the prefix java.

parameter1, parameter2, ...

Are the parameters which must be passed to the JAVA class according to the rules described in *Passing Parameters* on page 3-3.

Example: Calling iway.test.jclass Using EX

```
SET EXORDER=PGM/FEX
EX iway.test.jclass parameter1,"subparm1=val1,subparm2=val2",simple
parameter3
-EXIT
```

Passing Parameters

The following terminology is used in this section:

- All parameters in either a CALLJAVA call or EX command are separated by commas.
- You must enclose complex parameters containing commas in double quotation marks.
- If a parameter contains a double quote, code it as two consecutive double quotation marks with no spaces.
- Parameter names can have spaces.
- Enclose parameters with leading and/or trailing spaces in double quotes.
- Two consecutive commas do not represent a parameter; use " " to pass a blank parameter.
- One parameter is generated for an unbalanced double quotation mark.

Example: Passing Parameters

The following command invokes the JAVA class java.ibi.bony.bonypoc with three parameters:

```
EX java.ibi.bony.bonypoc Parameter1, " ", ""Parameter3""
```

Writing a JAVA Class

When you write a JAVA class to be invoked by the iWay Server, you use the class with the CALLJAVA interface, as much as you would use a 3GL program with the CALLPGM interface. The CALLJAVA interface defines two methods, **execute** and **fetch**.

- The execute method receives three parameters: user ID, password ID, and the String array of parameters. Any one of those parameters can be a null object reference. Null reference for the parameters array represents invocation with no parameters. The server invokes the JAVA class in the “password pass through” mode. The execute method performs the request and returns the instantiated IBI Answer Set object populated with the answer set description. The fetch method populates this object with data.
- The fetch method is invoked by the server to receive one row of the answer set at a time. IBI_EOD is returned when the answer is finished; IBI_DATA is returned to indicate more data is coming.

You can use **ibtrace.println** to trace execution of the program and **ibtrace.printException** to output all information about a caught interruption. For example:

```
} catch (Exception e) {
ibtrace.printException(e, "Caught execute interruption");
}
```

The output of ibtrace class method calls is forwarded into the standard server trace file.

Interfaces

ibiAnswerSet interface

```
package ibi.callpgm;

public interface ibiAnswerSet {

public static final int IBI_ALPHA = 6;
public static final int IBI_INTEGER = 1;
public static final int IBI_DATE = 10;
public static final String IBI_MISSING = ".";

public int getColsNumb();
public void setColName(int colIndex, String name);
public void setColType(int colIndex, int type);
public void setColSize(int colIndex, int size);
public void setColValue(int colIndex, String value);
}
```

callpgm interface

```
package ibi.callpgm;

public interface callpgm {
/**
* executes the request and returns answer set description
* @param username - the user name or null
* @param password - the user password or null
* @param parms - array of parameters or null
* @param ibianswr - the IBI Answer Set object
* @return ibianswr populated with the meta information
*/
public ibianswr execute(String username, String password, String[] parms)
throws Exception;
/**
* returns one row of the answer
* @param - none. IBI Answer Set object instantiated in "execute" is used
to return data
* @return End-Of-Data indicator
*/
public Integer fetch() throws Exception;

public static final Integer IBI_EOD = null;
public static final Integer IBI_DATA = new Integer(1);
}
```

JAVA Class Communication

When you execute a JAVA class (either with the CALLJAVA call or EX command), the server and the program communicate using an IBI answer set object.

This object has to be instantiated and populated with the answer set description on an "execute" method call. This method is called by the server only once. The server will call a "fetch" method repeatedly until it receives an IBI_EOD indicator. The server expects to receive the answer set row by row in the same instance of the IBI answer set object.

Compiling and Running a JAVA Program

When you compile your JAVA program, the jscom3.jar file located in the EDAHOME etc subdirectory needs to be accessible via the CLASSPATH environment variable or the javac command parameter.

When you execute your JAVA class, you need to place the client jar file containing the JAVA class to be invoked in the CLASSPATH environment variable prior to starting the iWay Server.

Example

```
package ibi.bony;

import ibi.trace.*;
import ibi.callpgm.*;

public class bonypoc implements callpgm{
private ibianswr answr = null;
private int rownum = 0;
private int Rows = 2;
private String[] arrParms = null;

public bonypoc(){}

public ibianswr execute(String username, String password, String[] parms)
throws Exception
{
    ibtrace.println("...BoNY POC Constractor");
    ibtrace.println("...BoNY POC Username: " + username + ", Password: "
+    password);
    int numParms = (parms != null) ? parms.length : 0;
    ibtrace.println("...BoNY POC There is(are) " + numParms + "
parameter(s)");
    for(int i=1; i <= numParms;
    ibtrace.println("...-> Parameter " + i + ": " + parms[i-1]), i++);

    if(numParms == 0)
```

```
    {
        Rows = 4;
        int Cols = 4;
        answr = new ibianswr(Cols);
        ibtrace.println("...BoNY POC number of columns set to " +
answr.getColsNumb());

        answr.setColSize(1, 6);
        answr.setColName(1, "Mes1");
        answr.setColSize(2, 15);
        answr.setColName(2, "Mes2");

        for (int col = 3; col <= Cols; col++)
        {
            answr.setColSize(col, 10);
            answr.setColName(col, "Col" + new Integer(col).toString());
        }
    }
    else
    {
         Rows = 2;
        arrParms = parms;
        answr = new ibianswr(arrParms.length);
        ibtrace.println("...BoNY POC number of columns set to " +
answr.getColsNumb());

        for (int col = 1; col <= answr.getColsNumb(); col++)
        {
            answr.setColSize(col,
                    (col == 1) ? Math.max(5,arrParms[col-1].length())
                               : arrParms[col-1].length());
                    answr.setColName(col, "Col" + col);
        }
    }

    return answr;
}

public Integer fetch() throws Exception
{
    if( ++rownum > Rows ) return IBI_EOD;
    if(arrParms == null)
    {
        switch(rownum)
        {
            case 1:
                answr.setColValue(3, null);
                answr.setColValue(4, null);
```

```
                answsr.setColValue(1, "There");
                answsr.setColValue(2, "is");
            break;
            case 2:
                answsr.setColValue(3, " ");
                answsr.setColValue(4, " ");
                answsr.setColValue(2, "parameters");
                answsr.setColValue(1, "no");
            break;
            default:
                for (int col = 1; col <= answsr.getColsNumb();
                    answsr.setColValue(col++, "Row " + rownum));
            break;
        }
    }
    else
    {
        switch(rownum)
        {
            case 1:
                answsr.setColValue(1, "Echo");
                for (int col = 2; col <= answsr.getColsNumb();
                    answsr.setColValue(col, " "),
                    col++);
                break;
            default:
                for (int col = 1; col <= answsr.getColsNumb();
                    answsr.setColValue(col, arrParms[col-1]),
                    col++);
                break;
        }
    }
    return IBI_DATA;
    }
}
```

CHAPTER 4

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 complied 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 within the program.
- Retain values used by your program.
- Issue the CREATE TABLE command to describe any answer set before returning it.

Theoretically, any 3GL language can be used provided it can be compiled and linked as a loadable library. However, reference examples and tools (GENCPGM) to assist in compilation and linking only exist for a limited set of languages. Thus, any 3GL language is supported, but some are untested and unlikely to be tested. For more information about GENCPGM, see Appendix B, *GENCPGM Usage* for languages supported and the samples in this chapter. If you are using an untested language and are having problems, contact iWay customer support so that a specialist can assist you.

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 files 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

In this section:

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.

CALLPGM supports two styles of control block layouts (old and new) and SET command to control which is used. The default continues to remain the old style for backward compatibility for existing applications. The difference between the two styles is the number of address areas (buffers) and the applicable values for signaling actions. The old style uses two address areas (buffers), one for passing messages and a shared one for creates and answers rows. The new style has a third address area so creates and answer rows each have there own buffer. Which style is used is controlled with the command statement

```
SQL SPG SET CPGUB style
```

where:

`OLD`

Uses two address area buffers.

`NEW`

Uses three address area buffers.

Applications are allowed to set the style at any time, but if all applications use the new style, then the command should be put in the server profile.

The benefit of the new style is that some new action flags were added for both OLD and NEW but some flags that are strictly for NEW. For example:

- Normally, to send a message, the subroutine would be called twice; first to send the message and a second time so the “all done” exit flag could be set. With the newer flag, a value of 13 (which means message and exit), the routine is only called once.

- Normally, to send multiple records, several calls would be used to set the create statement, get each record, and then the exit flag. Under the newer flags (specifically 18), and with CPGUB NEW, a single call can set the create buffer, load the answer set buffer with multiple records, and set the exit flag.

  Therefore, even a minor recoding of an old application to use the newer exit flags can gain in performance, but applications that can buffer up all data into a single pass will particularly benefit.

The following sections provide the control block specifications and examples of the control block in C, COBOL, and RPG.

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 and RPG.

Control Block Specification

Data layouts used in the control blocks in the following sections are described in the table below. Specific variable names used within an actual program and the samples provided by iWay Software vary based on the limitations of the languages, but closely follow the names below.

Field	Length (in bytes)	Data Type	Description
`input_CB_length`	2	Integer	Specifies the length of the input_CB passed by the server, including any passed parameters. Non-modifiable. The server supplies the value; the called program cannot modify it.
`reserved`	2	Integer	Non-modifiable. Reserved for server use.

Field	Length (in bytes)	Data Type	Description
flag_value	4	Integer	Specifies whether this is the first time the server has called the program for this client application: 1 First time. 0 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 flag_value 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 flag_value.

Field	Length (in bytes)	Data Type	Description
flag_value (*continued*)			The server supplies the value; the called program cannot modify it. 100 Program name invalid. 101 Cannot get main parameter buffer. 200 CS/2 error condition (a communications subsystem error). 300 Cannot get memory. 302 Cannot load program. 305 Bad value from user program. 306 Remote program abend. 307 Client abend. 308 CVT not found. 309 Cxinit call failed (an internal API error). 310 Cxdefault call error (an internal API error). 311 Cxsetuser call error (an internal API error). 312 Cxset call failed (an internal API error). 313 Invalid blocking factor. An action_value of 14, 15, or 18 was specified, but the blocking factor was ≤ 0.

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

Field	Length (in bytes)	Data Type	Description
action_value Initial value on first call: 4	4	Integer	Specifies the type of response from the called program: 1 Program returning a CREATE TABLE statement. 2 Program returning binary data. 3 Program returning character data. 4 Program returning a message. 9 Program done, exit flag … terminate and do not call routine again. 10 Program returning a CREATE TABLE statement plus exit flag. 11 Program returning binary data plus exit flag. 12 Program returning character data plus exit flag. 13 Program returning message plus exit flag. 14 Program returning data as a block of tuples. Row length supplied via message_length. 15 Program returning CREATE TABLE and data as a block of tuples. Row length supplied via message_length. 16 Program returning CREATE TABLE and binary data. 17 Program returning CREATE TABLE and character data.

Field	Length (in bytes)	Data Type	Description
action_value Initial value on first call: 4	4	Integer	18 Program returning CREATE TABLE and data as a block of tuples plus exit flag. Row length supplied via message_length. 19 Program returning CREATE TABLE, binary data plus exit flag. 20 Program returning CREATE TABLE, character data plus exit flag. Action values 15 and higher are only valid for use with SQL SPG SET CPGUB NEW. Undefined action values result in exit flag behavior. The called program supplies the value. Pointer padding filler for OS/400 pointers. Only required to exist for OS/400 applications. Do not declare on other platforms.
filler	4	Any type	Pointer padding filler for OS/400 pointers. Only required to exist for OS/400 applications. Do not declare on other platforms.
answer_area Initial value on first call: 0	4 (32 bit) 8 (64 bit) 16 (OS/400)	Pointer (address)	The address of the data returned by the called program. The called program supplies the value, depending on the action value. See *Storing Program Values* on page 4-20 for more information on the use of this field.
answer_length Initial value on first call: 0	4	Integer	The length of the data returned by the called program. The called program supplies the value, depending on the action value.

Field	Length (in bytes)	Data Type	Description
filler	12	Any type	Pointer padding filler for OS/400 pointers. Only required to exist for OS/400 applications. Do not declare on other platforms.
message_area Initial value on first call: 0	4 (32 bit) 8 (64 bit) 16 (OS/400)	Pointer (address)	The address of a message returned by the called program. The called program supplies the value when action_value is 4. See *Storing Program Values* on page 4-20 for more information on the use of this field.
message_length Initial value on first call: 0	4	Integer	The length of the message returned by the called program. Or The length of an answer row when data is returned as a block used when action_value is 14, 15, or 18. The called program supplies the value.
filler	12	Any type	Pointer padding filler for OS/400 pointers. Only required to exist for OS/400 applications. Do not declare on other platforms. Only existing and applicable when SQL SET CPGUB NEW, do not code when CPUG OLD is used.

Field	Length (in bytes)	Data Type	Description
create_area Initial value on first call: 0	4 (32 bit) 8 (64 bit) 16 (OS/400)	Pointer (address)	The address of a CREATE returned by the called program. The called program supplies the value when action_value is 4. See *Storing Program Values* on page 4-20 for more information on the use of this field. Only existing and applicable when SQL SET CPGUB NEW, do not code when CPUG OLD is used.
filler	12	Any type	Pointer padding filler for OS/400 pointers. Only required to exist for OS/400 applications. Do not declare on other platforms.
parmlen	4	Integer	The length of a parameter passed to the called program. The server supplies the value. This field is paired with parmdata (see next item). Twelve pairs are permitted per program call.
parmdata	Variable	Any type	The value of the parameter passed to the program. The server supplies the value (from EDARPC or a Dialogue Manager procedure). This field is paired with parmlen. Twelve pairs are permitted per program call.

Setting Up a CALLPGM Control Block Structure for C

Example:

Using SQL SPG SET CPGUB OLD in the C Control Block

Using SQL SPG SET CPGUB NEW in the C Control Block

To use CALLPGM with C, a data structure needs to be created. The precise structure depends on whether SQL SPG SET CPGUB is set to NEW or OLD.

The following examples use a static length string to carry size/data pairs for information passed as parameters to the program. It is the responsibility of the developers to place the string into specific variables by reading a given size (length) and moving the correct portion of the string into a variable, then moving down the string to the next size/value pairs until the length of the string is read. It is very important to not read past the end of the total length of the actual data structure (carried in input_CB_length, for example, commonarea_length in the C example) as this memory area may contain excess data depending on how a given operating system initializes memory. The examples included in this manual and the example on disk use one particular style for reading the input area into variables for use in the actual program, but any method can be used.

Example: Using SQL SPG SET CPGUB OLD in the C Control Block

```
typedef struct tag_CPGUB_ext   /* CPGUB structure
*/
{
   short commarea_length;    /* non-modifiable                          */
   short reserved;           /* reserved                                */
   long flag_value;          /* i:flag=1 1st time, =0 all other         */
#define CPGUB_flag_frst 1    /*  First time value for flag              */
#define CPGUB_flag_nfst 0    /*  Non-first time value for flag          */
   long action_value;        /* o:Action to be taken on callback        */
#define CPGUB_action_CT  1   /*  Create Table                           */
#define CPGUB_action_DA  2   /*  Data (Binary)                          */
#define CPGUB_action_CD  3   /*  Character Data                         */
#define CPGUB_action_MS  4   /*  Message                                */
#define CPGUB_action_EX  9   /*  Exit                                   */
#define CPGUB_action_CTE 10  /*  Create Table  & exit                   */
#define CPGUB_action_DAE 11  /*  Data (Binary) & exit                   */
#define CPGUB_action_CDE 12  /*  Character Data & exit                  */
#define CPGUB_action_MSE 13  /*  Message & exit                         */
#define CPGUB_action_DAB 14  /*  Data Block of Tuples                   */
                             /*  Use of any action not define is        */
                             /*  treated as CPGUB_action_EX, ie exit.   */
   char *answer_area;        /* o:answer area address                   */
   long  answer_length;      /* o:answer area length                    */
   char *return_value;       /* o:reply area address                    */
                             /*  for msg on _MS call (sent to client)   */
                             /*  for reply on _EX calls                 */
   long  return_length;      /* o:reply area length                     */
} t_CPGUB;
```

Example: Using SQL SPG SET CPGUB NEW in the C Control Block

```
typedef struct tag_CPGUB_ext   /* CPGUB structure
*/
{
   short commarea_length;    /* non-modifiable                          */
   short reserved;           /* reserved                                */
   long flag_value;          /* i:flag=1 1st time, =0 all other         */
#define CPGUB_flag_frst 1    /*  First time value for flag              */
#define CPGUB_flag_nfst 0    /*  Non-first time value for flag          */
   long action_value;        /* o:Action to be taken on callback        */
#define CPGUB_action_CT  1   /*  Create Table                           */
#define CPGUB_action_DA  2   /*  Data (Binary)                          */
#define CPGUB_action_CD  3   /*  Character Data                         */
#define CPGUB_action_MS  4   /*  Message                                */
#define CPGUB_action_EX  9   /*  Exit                                   */
#define CPGUB_action_CTE 10 /*  Create Table  & exit                    */
#define CPGUB_action_DAE 11 /*  Data (Binary) & exit                    */
#define CPGUB_action_CDE 12 /*  Character Data & exit                   */
#define CPGUB_action_MSE 13 /*  Message & exit                          */
#define CPGUB_action_DAB 14 /*  Data Block of Tuples                    */
#define CPGUB_action_CTB 15 /*  Create Table & Data Block of Tuples     */
#define CPGUB_action_CTD 16 /*  Create Table & Data (Binary)            */
#define CPGUB_action_CTC 17 /*  Create Table & Character Data           */
#define CPGUB_action_CTBE 18/*  Create Table, Block of Tuples & exit    */
#define CPGUB_action_CTDE 19/*  Create Table & Data (Binary) & exit     */
#define CPGUB_action_CTCE 20/*  Create Table & Character Data & exit    */
                           /*  Use of any action not define is         */
                           /*  treated as CPGUB_action_EX, ie exit.    */
   char *answer_area;        /* o:answer area address                   */
   long  answer_length;      /* o:answer area length                    */
   char *return_value;       /* o:reply area address                    */
                           /*  for msg on _MS call (sent to client)    */
                           /*  for reply on _EX calls                  */
   long  return_length;      /* o:reply area length                     */
   char *create_address;     /* o:create table data address             */
   long  create_length;      /* o:create table data length              */
} t_CPGUB_ext;
```

The difference between control blocks SQL SPG SET CPGUB NEW and OLD is that an additional CREATE pointer and length exist for returning separate CREATE information.

Setting Up a CALLPGM LINKAGE SECTION Control Block for Cobol

Example:

Using SQL SPG SET CPGUB OLD in the Cobol Control Block

Using SQL SPG SET CPGUB NEW in the Cobol Control Block

To use CALLPGM with Cobol, an 01 level data structure needs to be created. The precise structure depends on whether SQL SPG SET CPGUB is set to NEW or OLD.

Cobol requires the use of fillers for padding out pointer lengths on OS/400. The padding is a combination of being an OS/400 behavior for pointer alignment and COBOL requiring pointer alignment to be explicitly coded on OS/400. Do not use these OS/400 specific fillers on the platforms.

The following examples use a static length string to carry size/data pairs for information passed as parameters to the program. It is the responsibility of the developers to place the string into specific variables by reading a given size (length) and moving the correct portion of the string into a variable, then moving down the string to the next size/value pairs until the length of the string is read. It is very important to not read past the end of the total length of the actual data structure (carried in input_CB_length, for example, CALLPGM-DATA-LEN in the C example) as this memory area may contain excess data depending on how a given operating system initializes memory. The examples included in this manual and the example on disk use one particular style for reading the input string into variables for use in the actual program, but any method can be used.

Example: Using SQL SPG SET CPGUB OLD in the Cobol Control Block

```
01  CALLPGM-DATA.
    05  FIXED-LENGTH-PART.
        10  CALLPGM-DATA-LEN        PIC S9(4) BINARY.
        10  FILLER                  PIC  X(2).
        10  FLAG-VALUE              PIC S9(8) BINARY.
            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) BINARY.
            88  CREATE-TABLE            VALUE +1.
            88  RETURNING-MIXED-DATA    VALUE +2.
            88  RETURNING-CHAR-DATA     VALUE +3.
            88  RETURNING-MESSAGE       VALUE +4.
            88  PROGRAM-FINISHED        VALUE +9.
**** OS/400 Needs the filler on the next line for alignment.
**** All other platforms should have it commented out
*OS400  10  FILLER                   PIC X(4).
        10  ANSWER-ADDRESS          POINTER.
        10  ANSWER-LENGTH           PIC S9(8) BINARY.
**** OS/400 Needs the filler on the next line for alignment.
**** All other platforms should have it commented out.
*OS400  10  FILLER                   PIC X(12).
        10  MESSAGE-ADDRESS         POINTER.
        10  MESSAGE-LENGTH          PIC S9(8) BINARY.
**** OS/400 Needs the filler on the next line for alignment.
**** All other platforms should have it commented out.
**** The filler also needs to be here vs next section so
**** fix part length test operate correctly.
*OS400  10  FILLER                   PIC X(12).
    05  PARAMETERS-PART.
        Length is arbitrary at 80, could have been longer.
        Should be set the maximum expected length plus extra.
        10  INSTRING                PIC X(80).
```

Example: Using SQL SPG SET CPGUB NEW in the Cobol Control Block

```
01  CALLPGM-DATA.
    05  FIXED-LENGTH-PART.
        10  CALLPGM-DATA-LEN        PIC S9(4) BINARY.
        10  FILLER                  PIC  X(2).
        10  FLAG-VALUE              PIC S9(8) BINARY.
            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) BINARY.
            88  CREATE-TABLE            VALUE +1.
            88  RETURNING-MIXED-DATA    VALUE +2.
            88  RETURNING-CHAR-DATA     VALUE +3.
            88  RETURNING-MESSAGE       VALUE +4.
            88  PROGRAM-FINISHED        VALUE +9.
**** OS/400 Needs the filler on the next line for alignment.
**** All other platforms should have it commented out.
*OS400  10  FILLER                   PIC X(4).
        10  ANSWER-ADDRESS          POINTER.
        10  ANSWER-LENGTH           PIC S9(8) BINARY.
**** OS/400 Needs the filler on the next line for alignment.
**** All other platforms should have it commented out.
*OS400  10  FILLER                   PIC X(12).
        10  MESSAGE-ADDRESS         POINTER.
        10  MESSAGE-LENGTH          PIC S9(8) BINARY.
**** OS/400 Needs the filler on the next line for alignment.
**** All other platforms should have it commented out.
*OS400  10  FILLER                   PIC X(12).
        10  CREATE-ADDRESS          POINTER.
        10  CREATE-LENGTH           PIC S9(8) BINARY.
**** OS/400 Needs the filler on the next line for alignment.
**** All other platforms should have it commented out.
**** The filler also needs to be here vs next section so
**** fix part length test operate correctly.
*OS400  10  FILLER                   PIC X(12).
    05  PARAMETERS-PART.
        Length is arbitrary at 80, could have been longer.
        Should be set the maximum expected length plus extra.
        10  INSTRING                PIC X(80).
```

The difference between control blocks SQL SPG SET CPGUB NEW and OLD is that an additional CREATE pointer and length exist for returning separate CREATE information.

Setting Up a CALLPGM Data Structure Control Block for RPG

Example:

Using SQL SPG SET CPGUB OLD in the RPG Control Block

Using SQL SPG SET CPGUB NEW in the RPG Control Block

Note: RPG is an OS/400 only language.

To use CALLPGM with RPG, a data structure needs to be created. The precise structure depends on whether SQL SPG SET CPGUB is set to NEW or OLD.

RPG (like Cobol on OS/400) requires the use of fillers for padding out pointer lengths. This padding is a combination of being an OS/400 behavior for pointer alignment and RPG requiring pointer alignment to be explicitly coded.

The following examples use a static length string to carry size/data pairs for information passed as parameters to the program. It is the responsibility of the developers to place the string into specific variables by read a given size (length) and moving the correct portion of the string into a variable, then moving down the string to the next size/value pairs until the length of the string is read. It is very important to not read past the end of the total length of the actual data structure (carried in CB_LENGTH) as this memory area may contain excess data depending on how the operating system initialized memory. The examples included in this manual and the examples on disk use one particular style for reading the input string into variables for use in the actual program, but any method can be used.

Example: Using SQL SPG SET CPGUB OLD in the RPG Control Block

```
 * Control Block Data Structure ...
D cbds            DS
D  CB_LENGTH                     5I 0
D  FILLER                        2A
D  FLAG_VALUE                    9B 0
D  ACTION_VALUE                 10I 0
D  FILLERA                       4A
D  ANSWER_AREA                    *
D  ANSWER_LEN                   10I 0
D  FILLERM                      12A
D  MESSAGE_AREA                   *
D  MESSAGE_LEN                  10I 0
D  FILLERI                      12A
 * Parm Area (arbitrary minimum length)
D  PARMDATA                   1024A
```

Example: Using SQL SPG SET CPGUB NEW in the RPG Control Block

```
 * Control Block Data Structure ...
D cbds            DS
D  CB_LENGTH                     5I 0
D  FILLER                        2A
D  FLAG_VALUE                    9B 0
D  ACTION_VALUE                 10I 0
D  FILLERA                       4A
D  ANSWER_AREA                    *
D  ANSWER_LEN                   10I 0
D  FILLERM                      12A
D  MESSAGE_AREA                   *
D  MESSAGE_LEN                  10I 0
D  FILLERC                      12A
D  CREATE_AREA                    *
D  CREATE_LEN                   10I 0
D  FILLERI                      12A
 * Parm Area (arbitrary minimum length)
D  PARMDATA                   1024A
```

The difference between control blocks SQL SPG SET CPGUB NEW and OLD is that an additional CREATE pointer and length exist for returning separate CREATE information.

Storing Program Values

Example:

Storing Program Values in C

Storing Program Values in COBOL

Storing Program Values in COBOL II

Linking Program Variables to the Control Block

Checking for First-time Execution

Allocating and Freeing Dynamic Storage

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 in the following subsections.

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 an address (pointer) area.

Address area space allocations are by default 1024-bytes, which may suffice in some applications. An application may acquire dynamic storage on its own using those facilities of the language that are available for use within any given language on any given operating system or by issuing explicit commands to have the calling process (the server) set specific address area allocations for the called program to use.

To have the server set specific address allocations use one or more of the following commands:

```
SQL SPG SET SPGALLOC_CRT n     (SQL SPG SET CPGUB NEW ONLY)
SQL SPG SET SPGALLOC_MSG n
SQL SPG SET SPGALLOC_ANS n
```

Where *n* is a number between 1024 (1K) and 32768 (32K). The allocated address is then placed into the respective control block pointer location for the CALLPGM program to use.

To have the application itself acquire dynamic storage depending upon your environment use features such as:

- malloc in C.
- EXEC CICS GETMAIN in COBOL or C under CICS.
- 'GETCOR' in COBOL under VTAM.
- “LIB$GET_VM” in COBOL under OpenVMS.

It is the program’s responsibility to free such storage at its last invocation.

It may also be necessary for subsequent invocations of a program to retrieve previously stored values, which would also require the use of dynamically acquired storage method.

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-addresses 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 with your software in locations as described below.

Type of Program	Supplied As
C	MVS CPGC370 in *qualif*.EDALIB.DATA All other platforms *cpt.c* in $EDAHOME/etc
CICS COBOL	CPGCICS in *qualif*.EDACICS.DATA for CICS COBOL II usage.
MVS COBOL	CPGVTAM in *qualif*.EDACTL.DATA for MVS COBOL II usage.
COBOL	Portable version for all other platforms (including MVS using Enterprise COBOL.) SPGOLD.CBL in $EDAHOME/etc SPGNEW.CBL in $EDAHOME/etc
RPG	(OS/400 only) SPGOLD.RPG in $EDAHOME/etc SPGNEW.RPG in $EDAHOME/etc

The portable COBOL examples have specifically been tested with IBM Enterprise COBOL V3R2, HP OpenVMS COBOLv2.7, and IBM OS/400 ILE COBOL. Depending on the target platform, minor editing (for example, commenting or uncommenting of lines) is required for use. Specific instructions are contained as comments at the beginning of the file.

The supplied samples work by parsing the parameters passed to the program and passing back information such as a number of records to return. None of the samples use actual database access; they simulate what and how to send data and messages back to the calling process using arbitrary text, therefore they need little in the way of setup for demonstration purposes. The samples all contain comments on requirements for compilation and use.

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.

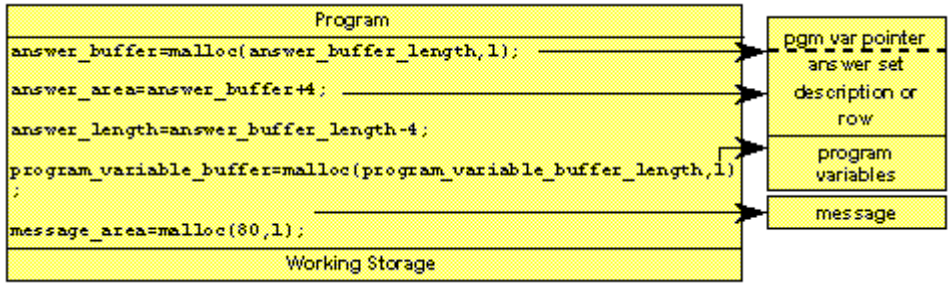
```
typedef struct message_buffer
   { char     message[80] ;
   } message_buffer;

 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.                                                     */
/*                                                                    */
/* 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).

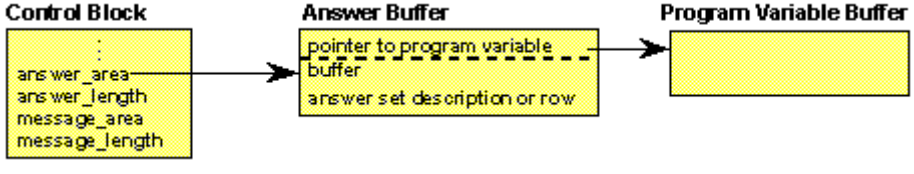
The pointer to the program variable buffer is saved at a fixed location (a known offset), in the first *n* bytes of the buffer, for the answer set description or row (called the answer buffer). This is illustrated in the figure below.



For example, 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 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 in this example using EXEC CICS GETMAIN in COBOL under CICS as the reference platform, but any language supporting the setting of dynamic storage can be used with the syntax specific to that language.

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.

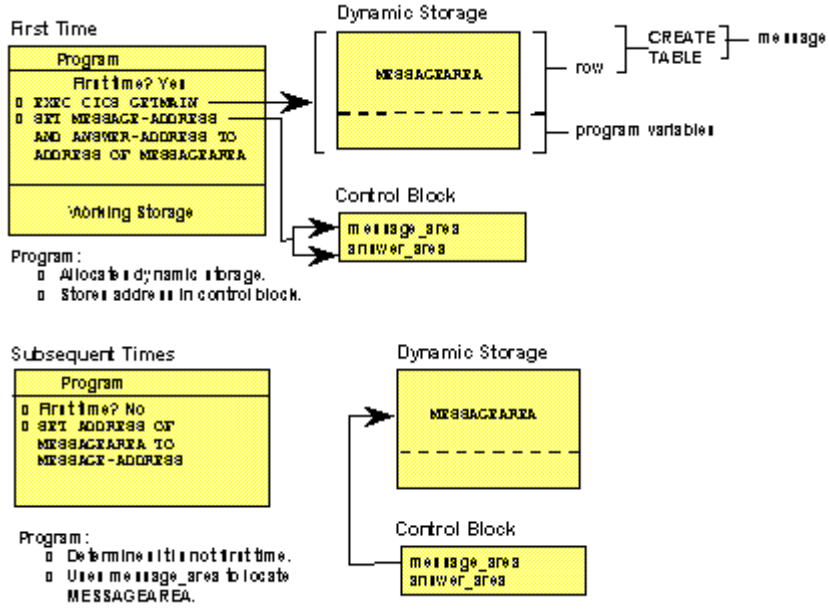
```
01  MESSAGEAREA.
    05  MESSAGE-OUT              PIC X(1000).
    05  NUM-ROWS                 PIC S9(8)  COMP-4.
    05  LAST-REC                 PIC S9(8)  COMP-4.
    05  CLOSE-PENDING-FLAG       PIC X.
        88  CLOSE-PENDING        VALUE "1".
        88  CLOSE-NOT-PENDING    VALUE "0".
 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

The following example uses VTAM MVS COBOL II as the reference platform.

To allocate dynamic storage, 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) BINARY.
    05  MESSAGEAREA-ADDRESS             POINTER.
    05  COR-RESP                        PIC S9(8) BINARY.
                    .
                    .
                    .
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'.

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 Specification* on page 4-4.

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

Note: OpenVMS uses the keywords "TO REFERENCE OF" instead of "TO ADDRESS OF".

Example: Checking for First-time Execution

The following 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.

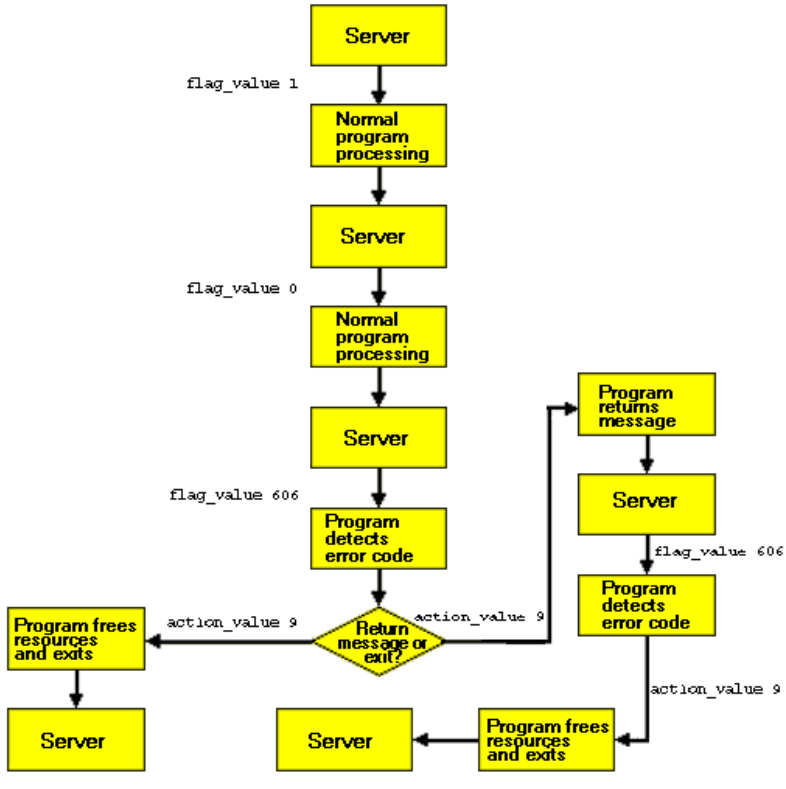
The following figure illustrates the correct error handling sequence. In the figure, the following flag_values are shown:

`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.

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.

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
```

or

```
PIC S9(15)    COMP-3
```

CHAPTER 5

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
- .EVAL Operator
- Creating Expressions
- Using Functions
- Using Commands Specific to an Operating System
- ON TABLE HOLD
- ON TABLE PCHOLD

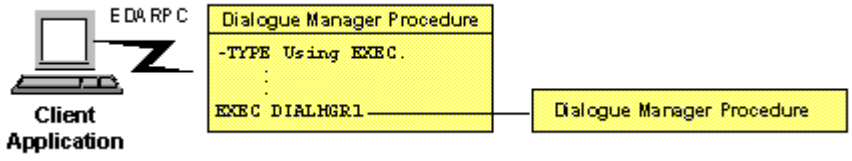
A Dialogue Manager procedure is a file of commands that resides on a server. It typically includes SQL statements that perform tasks such as report generation or file maintenance, or it simply generates messages.

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, *Introducing Stored Procedures*, 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 5-48 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 5-79 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 5-80 for details on the syntax and use.
- The CALLIMS function.
- Platform-specific commands (for example, DYNAM in MVS). See Chapter 6, *Platform-specific Commands*, for details.

Commands and Processing

In this section:

Dialogue Manager Processing

Example:

Issuing an API Function Call (EDARPC)

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.

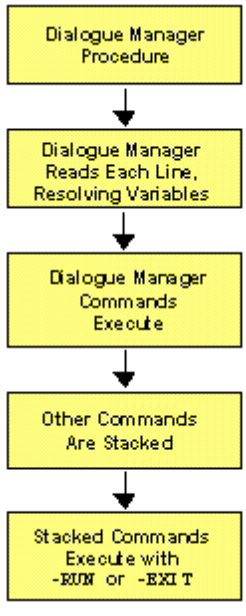
Command	Function
-*	Signals a comment.
-?	Displays the value of local variables.
-CLOSE	Closes an external file opened for reading or writing (an external file is a sequential file in the platform's file system).
-AS/400	Executes an OS/400 operating system command, ignored on other operating systems.
-CMS	Executes a CMS operating system command, ignored on other operating systems.
-DEFAULT -DEFAULTS	Sets a variable to an initial value.
-DOS	Executes a DOS operating system command, ignored on other operating systems.
-EXIT	Executes stacked commands and terminates the procedure. See *Dialogue Manager Processing* on page 5-5 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.

Command	Function
-PASS	Directly issues and controls passwords.
-PROMPT	Types a message to the terminal (if edastart -t is in use) or creates an input window with the message in a browser if the connection type is HTTP and reads the reply from the user. This reply assigns a value to the variable named.
-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.
-RUN	Executes stacked commands and closes any external files opened with -READ or -WRITE.
-SET	Sets a variable to a literal value or to a value computed in an expression.
-SYSTEM	Executes an operating system command regardless of actual operating system type.
-TSO RUN	Executes an MVS operating system command, ignored on other operating systems.
-TYPE	Sends a message to a client application.
-UNIX	Executes a UNIX operating system command, ignored on other operating systems.
-VMS	Executes a VMS operating system command, ignored on other operating systems.
-WINNT	Executes a Windows NT operating system command, ignored on other operating systems.
-WRITE	Writes data to an external file.
-	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  07/28/04  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 quotation marks display 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  07/28/04  SLRPT
-* Component of Retail Sales Reporting Module
-TYPE This report calculates percentage of returns.
SQL
    .
    .
    .
```

Controlling Execution

In this section:

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.
2. Assume that the value of &CODE is B11.

   The command -IF tests the value and passes control to -QUIT.

   The command END is a delimiter.
3. The command -QUIT cancels execution of the stacked commands and exits the procedure.

Using Variables

In this section:

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 5-22 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

How to:

Use Variables in a Procedure

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:

Invalid	Reason
`&CORPORATECITY`	Too long (exceeds 12 characters).
`&WHICH CITY`	Contains embedded blank.
`&WHICH-CITY`	Contains a hyphen (-).
`WHICHCITY`	Leading ampersand(s) is missing.

Local Variables

Example:

Using Local Variables

Creating an Indexed Variable

Displaying the Value of 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 5-52 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

Example:

Using Global Variables

Displaying the Value of 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

Example:

Using System Variables

The table in this section describes system variables that you can use 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 5-23 for information.

System Variable	Description	Format or Value
&APPROOT	Physical location of the APPROOT directory.	Directory name
&DATE	Current date.	MM/DD/YY
&DATE*fmt*	Current date.	*fmt* is any combination of YYMD, MDYY, etc.
&MDY	Current date. Useful for numerical comparisons.	MMDDYY
&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

System Variable	Description	Format or Value
&FOCFOCEXEC	Current running procedure.	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	Current included procedure.	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.
&ECHO	Current echo tracing value.	ON, OFF, or ALL
&FOCMODE	Operating environment.	AS/400 CMS CRJE MSO DOS TSO UNIX VMS WINNT
&FOCPRINT	Current print setting.	ONLINE OFFLINE
&FOCREL	Source code release number.	Release number (for example, R720530B).
&IORETURN	Value returned after the last Dialogue Manager -READ or -WRITE operation.	0 Successful operation 1 End or failure

System Variable	Description	Format or Value
`&RETCODE`	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).
`&TOD`	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.	`HH.MM.SS`
`&FOCNET`	Environment.	`CLIENT, SERVER` 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 < '08/08/2004'
-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

In this section:

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

How to:

Pass Keyword Parameters

Pass Positional Parameters

Pass Long Parameters

Example:

Supplying Values With the EXEC Command

Combining Positional and Keyword Parameters

Passing Long Parameters

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'
   .
   .
   .
```

The calling procedure would issue:

```
EX SLRPT B10,B20,STAMFORD
```

Example: Combining Positional and Keyword Parameters

Consider the following procedure named PPARM1:

```
SQL
SELECT &1, &2, &field1, &3 FROM CAR;
```

It is called by another procedure using EXEC, with the values MODEL, MPG, field1=CAR, and COUNTRY supplied on the command line as parameters:

```
EXEC PPARM1 MODEL,MPG,field1=CAR,COUNTRY
```

Syntax: How to Pass Long Parameters

Note: As of release 5.3, the line length limit has been increased to 32K, so large parameters can be directly passed. The method described below continues to be supported, but there is little reason to use this method in coding new applications.

Individual parameter keyword value pairs that are longer than 80 characters (the limit of a physical line) must use special syntax to describe how many lines to read.

```
EX[EC] - LINES number procedure_name=value[,...]
```

where:

number

Is the number of additional lines to read as a continuation of the initial line. Break variables into 80 character lengths per line.

procedure_name

Is the name of a Dialogue Manager procedure.

value

Is the actual string being passed.

Example: Passing Long Parameters

The following is an example of a server procedure passing the maximum parameter of 32,000 bytes:

```
"EX -LINES 401 CPG32000=LINE00000000000000000000000000000000000000000001      "
"LINEOFINFORMATION1111111111111111111111111111111111111111111111111111111111"
"LINEOFINFORMATION2222222222222222222222222222222222222222222222222222222222"
"LINEOFINFORMATION4004004004004004004004004004004004004004004004004004004004"
.
.
.
"LASTLINETOTAL32000BYTESTHEENDXX"
```

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. OFF is the default value.

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.

-DEFAULT[S] 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 -DEFAULT[S] Command

```
-DEFAULT[S] &[&]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 -DEFAULT[S] Command

In the following example, -DEFAULT[S] 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 -DEFAULT[S].

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
```

-SET Command

How to:

Supply Values With the -SET Command

Example:

Assigning Values With 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 (;).

You can assign a literal value to a variable. Single quotation marks around the literal value are optional unless it contains embedded blanks or commas, in which case you must include single 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 the line length limit, which was 80 but is now 32K as of release 5.3. Using concatenation, several lines of separate values may be joined together to form longer variables, which is the method used for older releases. For example,

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

The maximum length of a variable is 32K, however, this is further limited if the variable is to participate in a calculation or concatentation.

Through the technique of building strings above, 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 but the total maximum is 415.

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
  .
  .
  .
```

-READ Command

How to:

Supply Values With the -READ Command

Example:

Specifying Length

Using 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 5-31.

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 example shown in *Using the -READ Command* on page 5-31.

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 DSNAME EDAUSER.EXTFILE.DATA SHR
```

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

```
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

In this section:

Screening Values With -IF Tests

How to:

Use the -GOTO Command for Unconditional Branching

Use the -IF...GOTO Command for Conditional Branching

Example:

Using the -GOTO Command for Unconditional Branching

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

Using Compound -IF Tests

Reference:

Processing a -GOTO Command

Operators and Functions in -IF Tests

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 a 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 5-36.

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 a -IF test include arithmetic and logical operators, as well as available functions. See *Creating Expressions* on page 5-52 and *Using Functions* on page 5-59 for details.

Screening Values With -IF Tests

How to:

Test for the Presence of a Value

Test for the Length of a Value

Test for the Type of a Value

Example:

Testing for the Presence of a Variable

Testing for Variable Length

Testing for Variable Type

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 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.

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 the remainder of a valid expression, such as GT 8.

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.
```

Example Storing the Length of a Variable for Later Use

The following example sets the variable &WORDLEN to the length of the string contained in the variable &WORD.

```
-PROMPT &WORD.ENTER WORD.
-SET &WORDLEN = &WORD.LENGTH;
```

You can use this technique when you want to use one variable to populate another.

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 109–1 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 the remainder of a valid expression, such as EQ A.

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

In this section:

Ending a Loop

How to:

Use the -REPEAT Command

Example:

Using the -REPEAT Command

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. 1 is the default value.

`TO toval`

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

`STEP s`

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

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

In this section:

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

Reference:

Other Uses of the -INCLUDE Command

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

You can also use the -INCLUDE command 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 5-23 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. A -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 writes information to it. This section describes the command -WRITE. For information on -READ, see *Supplying Values for Variables* on page 5-23.

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. For example, on UNIX:

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

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

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

```
-WRITE CAR &ALINE
```

.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 amper 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 writes information to it. This section describes the command -WRITE. For information on -READ, see *Supplying Values for Variables* on page 5-23.

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

Example:

Using Arithmetic Expressions

Reference:

Guidelines for Using 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 5-59 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

How to:

Concatenate Alphanumeric Variables and Literals

Use Date Functions

Example:

Concatenating Alphanumeric Variables and Literals

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 *Concatenate Alphanumeric Variables and Literals* on page 5-54 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.

alphaexp1, alphaexp2

Are 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 Functions

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 040101:

```
-SET &LOSRV = YMD(&HIRE_DATE,040101);
```

Logical Expressions

Example:

Forming a Logical Expression

Reference:

Guidelines for Alphanumeric and 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.

Operator	Description
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.

Operator	Description
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 a -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

In this section:

This section describes system-supplied functions you can use in expressions. Write your own functions to solve specific application problems.

System-supplied Function Examples

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

Function	Description
ABS	Returns the absolute value of a number. Computes on one argument: `-SET &PRICE = (ABS(&AMOUNT-&OLDAMOUNT))/100;`
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	*value*	Returns the absolute value of a number.
ARGLEN	*inlength,infield,outfield*	Calculates the non-blank length of an alphanumeric field.
ASIS	*value*	Distinguishes between a blank and zero.
ATODBL	*number,inlength,outfield*	Converts an alphanumeric field containing numeric data to a double-precision decimal field.
AYM	*indate,months,outfield*	Adds or subtracts a number of months from a given date.
AYMD	*indate,days,outfield*	Adds or subtracts a number of days from a given date.
BAR	*barlength,infield,maxvalue, char,outfield*	Includes a bar graph in a tabular report. Available for MVS and VM only.
BITSON	*bitnumber,infield,outfield*	Interprets multi-punch data (data that cannot be represented alphanumerically).
BITVAL	*infield,startbit,number, outfield*	Obtains the decimal value of a string of bits.
BYTVAL	*character,outfield*	Obtains the decimal equivalent of an alphanumeric character.
CHGDAT	*oldformat,newformat,indate, outfield*	Changes the format of a date.
CHKFMT	*numchar,infield,mask, outfield*	Checks character strings for incorrect character types.
CHKPCK	*inlength,infield,error, outfield*	Validates packed field format.

Function	Arguments	Description
CNCTUSR	*outfield*	Indicates the connected user.
CTRAN	*inlength,infield,incode, outcode,outfield*	Substitutes characters in a string.
CTRFLD	*infield,inlength,outfield*	Centers character strings within fields.
DADMY	*indate,outfield*	Calculates number of days from 1/1/00 with input in day-month-year format.
DADYM	*indate,outfield*	Calculates number of days from 1/1/00 with input in day-year-month format.
DAMDY	*indate,outfield*	Calculates number of days from 1/1/00 with input in month-day-year format.
DAMYD	*indate,outfield*	Calculates number of days from 1/1/00 with input in month-year-day format.
DAYDM	*indate,outfield*	Calculates number of days from 1/1/00 with input in year-day-month format.
DAYMD	*indate,outfield*	Calculates number of days from 1/1/00 with input in year-month-day format.
DATEADD	*YYMDdate,unit,#units*	Adds or subtracts a unit to or from a date format.
DATECVT	*indate,infmt,outfmt*	Converts date formats within applications without requiring intermediate calculations.
DATEDIF	*fromYYMD,toYYMD,unit*	Returns the difference between two dates in units.
DATEMOV	*YYMDdate,move-point*	Moves a date to a significant point on the calendar.

Function	Arguments	Description
DECODE	*instring (invalue outvalue...)*	Value translation (for more information, see *Decoding a Value* on page 5-72).
DMOD	*dividend,divisor,outfield*	Calculates the remainder from a division operation and returns a double-precision value.
DMY	*begin,end*	Calculates the difference between two dates in integer, alphanumeric, or packed format.
DOWK	*indate,outfield*	Provides the day of the week (in 4-character alphanumeric format) based on input date.
DOWKL	*indate,outfield*	Provides the day of the week (in 12-character alphanumeric format) based on input date.
DTDMY	*number,outfield*	Calculates the date from the number of days since 1/1/00 in day-month-year format.
DTDYM	*number,outfield*	Calculates the date from the number of days since 1/1/00 in day-year-month format.
DTMDY	*number,outfield*	Calculates the date from the number of days since 1/1/00 in month-day-year format.
DTMYD	*number,outfield*	Calculates the date from the number of days since 1/1/00 in month-year-day format.
DTYDM	*number,outfield*	Calculates the date from the number of days since 1/1/00 in year-day-month format.
DTYMD	*number,outfield*	Calculates the date from the number of days since 1/1/00 in year-month-day format.

Function	Arguments	Description
EDIT	*infield*[,mask]	Converts numeric to string when both parms are supplied or string to masked string when only the first parm is supplied. For more information, see *Editing a Value* on page 5-71.
EXP	*power,outfield*	Raises "e" to a given power.
EXPN	*n.nn {E|D} {+|-} p*	Evaluates an argument expressed in scientific notation.
FEXERR	*nnnnn*,A72	Retrieves an error message.
FGETENV	*envnamelen,envname, outfieldlen,outfldformat*	Retrieves the value of an environment variable and returns it as an alphanumeric string.
FINDMEM	*ddname,member,outfield*	Determines whether a partitioned data set contains a specified member. Available for MVS only.
FMLINFO	*'forvalue', outfield*	Retrieves FML value for direct use in calculations or a report.
FMOD	*dividend,divisor,outfield*	Calculates the remainder from a division operation and returns a single-precision value.
FORECAST	*fld2,interval,npredict, method*	Uncovers trends in numeric data. Methods are: Simple Moving Average (MOVAVE), Exponential Moving Average (EXPAVE), and Linear Regression Analysis (REGRESS).
FPUTENV	*namelength,name, valuelength,value,outfield*	Assigns a character string to an environment variable.
FTOA	*number,usage,outfield*	Converts a numeric field to alphanumeric format without inserting leading zeros.

Function	Arguments	Description
GETPDS	*ddname,member,outfield*	Determines whether a specific member of a partitioned data set (PDS) exists and returns the PDS name.
GETSECID	*outfield*	Retrieves the security ID. Available for MVS only.
GETTOK	*infield,inlen,toknum,delim, outlen,outfield*	Extracts a token from a data string.
GETUSER	*outfield*	Retrieves the user ID from the system.
GREGDT	*indate,outfield*	Converts a Julian date to a Gregorian date.
HADD	*dtfield,component,increment ,length,Hformat*	Specifies a date-time field by a given number of units.
HCNVRT	*dtfield,Hfmt,rlength,Ann*	Converts a date-time field to alphanumeric format for use with operators such as EDIT, CONTAINS, and LIKE.
HDATE	*dtfield,dateformat*	Extracts the date portion of a date-time field and converts it to a date format.
HDIFF	*dtfield1,dtfield2,component ,Dformat*	Finds the number of boundaries of a given type crossed going from date 2 to date 1.
HDTTM	*datefield,length,Hformat*	Converts a date field to a date-time field.
HEXBYT	*number,outfield*	Obtains the character equivalent of a numeric value.
HGETC	*length,Hformat*	Stores the current date and time in a date-time field.
HHMMSS	*outfield*	Retrieves the current time from the system.

Function	Arguments	Description
HINPUT	*inputlength,inputstring, length,Hfmt*	Converts an alphanumeric string to a date-time value.
HMIDNT	*dtfield,length,Hformat*	Changes the time portion of a date-time field to midnight.
HNAME	*dtfield,component,Aformat*	Extracts a specified component from a date-time field and returns it in alphanumeric format.
HPART	*dtfield,component,Iformat*	Extracts a specified component from a date-time field and returns it in numeric format.
HSETPT	*dtfield,component,value, length,Hformat*	Inserts the numeric value of a specified component into a date-time field.
HTIME	*length,dtfield,Dformat*	Converts the time portion of a date-time field to a numeric number of milliseconds or microseconds.
IMOD	*dividend,divisor,outfield*	Calculates the remainder from a division operation and returns an integer value.
INT	*value*	Returns the integer part of an argument.
ITONUM	*sigbytes,infield,outfield*	Converts large binary integers in non-FOCUS files to double-precision format.
ITOPACK	*sigbytes,infield,outfield*	Converts large binary integers in non-Dialogue Manager files to packed format.
ITOZ	*outlength,number,outfield*	Converts a numeric field to a zoned decimal.
JULDAT	*indate,outfield*	Converts a Gregorian date to a Julian date.

Function	Arguments	Description
LCWORD	*length,instring,outstring*	Translates uppercase characters in alphanumeric fields to lowercase on a word-by-word basis.
LJUST	*inlength,infield,outfield*	Left-justifies a character string within a field.
LOCASE	*length,infield,outfield*	Translates uppercase characters in alphanumeric fields to lowercase characters.
LOG	*value*	Returns the logarithm of a number, base e.
MAX	*value1,value2...*	Returns the maximum value.
MDY	*begin,end*	Calculates the difference between two dates in integer, alphanumeric, or packed format.
MIN	*value1,value2...*	Returns the maximum value.
MVS DYNAM	*command,length,rc*	Transfers a specified FOCUS DYNAM command to the DYNAM command processor.
NORMSDST	*value,outfield*	Calculates percent of data value less than or equal to a normalized value.
NORMSINV	*value,outfield*	Calculates the upper percent of data value boundary for a normalized value.
OVRLAY	*base,baselen,substring, sublen,position,outfield*	Overlays a character string on a character string.
PARAG	*inlength,infield,delimiter, subsize,outfield*	Inserts a delimiter into a character string.
PCKOUT	*infield,outlength,outfield*	Outputs a packed field.
POSIT	*parent,inlength,substring, sublength,outfield*	Finds the position of a character string in another string.

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

Function	Arguments	Description
TRUNCATE	*var1*	Removes trailing blanks from Dialogue Manager amper variables and adjusts the length accordingly.
UFMT	*infield,inlength,outfield,*	Converts characters in alphanumeric fields to HEX representation.
UPCASE	*length,infield,outfield*	Translates lowercase characters in alphanumeric fields to uppercase.
YM	*fromdate,todate,outfield*	Returns the number of months between two dates.
YMD	*begin,end*	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'.

Verifying Function Parameters

How to:

Enable Parameter Verification

Control Function Parameter Verification

The USERFCHK setting controls the level of verification applied to DEFINE FUNCTION and Information-Builders-supplied function arguments. It does not affect verification of the number of parameters; the correct number must always be supplied.

USERFCHK is not supported from Maintain.

Functions typically expect parameters to be a specific type or have a length that depends on the value of another parameter. It is possible in some situations to enforce these rules by truncating the length of a parameter and, therefore, avoid generating an error at run-time.

The level of verification and possible conversion to a valid format performed depends on the specific function. The following two situations can usually be converted satisfactorily:

- If a numeric parameter specifies a maximum size for an alphanumeric parameter, but the alphanumeric string supplied is longer than the specified size, the string can be truncated.
- If a parameter supplied as a numeric literal specifies a value larger than the maximum size for a parameter, it can be reduced to the proper value.

Syntax: How to Enable Parameter Verification

Parameter verification can be enabled only for DEFINE FUNCTIONs and functions supplied by Information Builders. If your site has a locally written function with the same name as an Information-Builders-supplied function, the USERFNS setting determines which function will be used:

```
SET USERFNS= {SYSTEM|LOCAL}
```

where:

SYSTEM

Gives precedence to functions supplied by Information Builders. SYSTEM is the default setting. This setting is required in order to enable parameter verification.

LOCAL

Gives precedence to locally written functions. Parameter verification is not performed with this setting in effect.

Syntax: How to Control Function Parameter Verification

Issue the following command in FOCPARM, FOCPROF, on the command line, in a FOCEXEC, or in an ON TABLE command. Note that the USERFNS=SYSTEM setting must be in effect

```
SET USERFCHK = setting
```

where:

setting

Can be one of the following:

ON is the default value. Verifies parameters in requests, but does not verify parameters for functions used in Master File DEFINEs. If a parameter has an incorrect length, an attempt is made to fix the problem. If such a problem cannot be fixed, a message is generated and the evaluation of the affected expression is terminated.

Note that if a parameter provided is the incorrect type, verification fails and processing terminates.

Because parameters are not verified for functions specified in a Master File, no errors are reported for those functions until the DEFINE field is used in a subsequent request when, if a problem occurs, the following message is generated:

```
(FOC003) THE FIELDNAME IS NOT RECOGNIZED
```

OFF does not verify parameters except in the following cases:

- If a parameter that is too long would overwrite the memory area in which the computational code is stored, the size is automatically reduced without issuing a message.
- If an alphanumeric parameter is too short, it is padded with blanks to the correct length.

FULL is the same as ON, but also verifies parameters for functions used in Master File DEFINEs.

Note that if a parameter provided is the incorrect type, verification fails and processing terminates.

ALERT verifies parameters in a request without halting execution when a problem is detected. It does not verify parameters for functions used in Master File DEFINEs. If a parameter has an incorrect length and an attempt is made to fix the problem behind the scenes, the problem is corrected with no message. If such a problem cannot be fixed, a warning message is generated. Execution then continues as though the setting were OFF, but the results may be incorrect.

Note that if a parameter provided is the incorrect type, verification fails and processing terminates.

Creating Routines

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 most platforms, as well as sample routines, but any method is allowed provided that it compiles and links a program correctly. 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

How to:

Decode a Value

Store Codes and Results in a Separate File

Example:

Using the DECODE Function

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...;
```

Creating an Indexed Variable

You can append the value of one variable to the value of another variable, creating an *indexed variable*. This feature applies to both local and global variables.

If the indexed 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 can be used as a counter; it can be set, incremented, and tested in a procedure.

Syntax How to Create an Indexed Variable

```
-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.

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. For information on the kinds of expressions you can write, see the *Creating Reports* manual.

Example Using an Indexed Variable in a Loop

An indexed variable can be 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 increments the index each time the procedure goes through the 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 do not change.

Indexed variables are included in the system limit of 1024, which includes variables reserved by FOCUS.

Removing Trailing Blanks From Variables With the TRUNCATE Function

The Dialogue Manager TRUNCATE function removes trailing blanks from Dialogue Manager amper variables and adjusts the length accordingly.

The Dialogue Manager TRUNCATE function has only one argument, the string or variable to be truncated. If you attempt to use the Dialogue Manager TRUNCATE function with more than one argument, the following message is generated:

```
(FOC03665) Error loading external function 'TRUNCATE'
```

This function can only be used in Dialogue Manager commands that support function calls, such as -SET and -IF commands. It cannot be used in -TYPE or -CRTFORM commands or in arguments passed to stored procedures.

Note: A user-written function of the same name can exist without conflict.

Syntax How to Remove Trailing Blanks From Variables

```
-SET &var2 = TRUNCATE(&var1);
```

where:

&var2

Is the Dialogue Manager variable to which the truncated string is returned. The length of this variable is the length of the original string or variable minus the trailing blanks. If the original string consisted of only blanks, a single blank, with a length of one is returned.

&var1

Is a Dialogue Manager variable or a literal string enclosed in single quotation marks. System variables and statistical variables are allowed as well as user-created local and global variables.

Example Removing Trailing Blanks

The following example shows the result of truncating trailing blanks:

```
-SET &LONG = 'ABC    ' ;
-SET &RESULT =  TRUNCATE(&LONG);
-SET &LL = &LONG.LENGTH;
-SET &RL = &RESULT.LENGTH;
-TYPE LONG    =  &LONG  LENGTH =  &LL
-TYPE RESULT =  &RESULT LENGTH =  &RL
```

The output is:

```
LONG    =  ABC      LENGTH =  06
RESULT =  ABC LENGTH =  03
```

The following example shows the result of truncating a string that consists of all blanks:

```
-SET &LONG = '       ' ;
-SET &RESULT =  TRUNCATE(&LONG);
-SET &LL = &LONG.LENGTH;
-SET &RL = &RESULT.LENGTH;
-TYPE LONG    =  &LONG  LENGTH =  &LL
-TYPE RESULT =  &RESULT LENGTH =  &RL
```

The output is:

```
LONG    =           LENGTH =  06
RESULT =    LENGTH =  01
```

The following example uses the TRUNCATE function as an argument for EDIT:

```
-SET &LONG = 'ABC    ' ;
-SET &RESULT =  EDIT(TRUNCATE(&LONG)|'Z','9999');
-SET &LL = &LONG.LENGTH;
-SET &RL = &RESULT.LENGTH;
-TYPE LONG   =  &LONG  LENGTH =  &LL
-TYPE RESULT =  &RESULT LENGTH =  &RL
```

The output is:

```
LONG   =  ABC      LENGTH =  06
RESULT =  ABCZ LENGTH =  04
```

Using Variables to Alter Commands

A variable can refer to a FOCUS command or to a particular field. Therefore, the command structure of a procedure can be determined by the value of the variable.

Example Using a Variable to Control What the TABLE Command Prints

In this example, the variable &FIELD determines the field to print in the TABLE request.

In the file named SALES, the variable &FIELD can display the values RETURNS, DAMAGED, or UNIT_SOLD.

```
TABLE FILE SALES
 .
 .
 .
PRINT &FIELD
BY PROD_CODE
 .
 .
 .
```

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:

`[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 -! feature.

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 5-80.
- 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 value.

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.

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)

In this section:

How to:

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.

ALLOCATE ALLOC ALLO	Allocates a data set. See *The ALLOCATE Subcommand* on page 6-6.
CONCAT CONC	Concatenates data sets. See *The CONCAT Subcommand* on page 6-15.
FREE	Frees data sets specified by ddnames or dsnames. Names may contain wildcard characters. See *The FREE Subcommand* on page 6-16.
CLOSE CLO	Closes data sets. Use this subcommand when data sets cannot be freed because of being open. See *The CLOSE Subcommand* on page 6-17.
COPY	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 *The COPY Subcommand* on page 6-18.
COPYDD	Copies a sequential data set or PDS member. COPY handles all the features of COPYDD, and is recommended for use instead of COPYDD. See *The COPYDD Subcommand* on page 6-20.
DELETE DEL	Deletes an entire data set or selected PDS members. See *The DELETE Subcommand* on page 6-21.
RENAME REN	Renames an entire data set or selected PDS members. See *The RENAME Subcommand* on page 6-22.
SUBMIT SUB	Submits MVS jobs. See *The SUBMIT Subcommand* on page 6-23.
COMPRESS COMP	Compresses a PDS. See *The COMPRESS Subcommand* on page 6-24.

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`
`DELETE`
`KEEP`
`UNCATALOG`
	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. CATLG and UNCAT are also valid as synonyms for CATALOG and UNCATALOG. DELETE, KEEP, and UNCATLOG follow the standard MVS meanings of delete after free, keep as is after free, and keep uncataloged.

`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:

NEW	is the default data set status. Incompatible with SYSOUT.
MOD	is an extended data set.
OLD	is exclusive control of the data set.
SHR	is shared access to the data set.

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 an 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 REFVOL 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 *recfm*

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 an MSO user. Valid only in an 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, a message is issued only if a single ddname without wildcards is specified. A 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

TSO:	`TSO ALLOC F(EDARPC) DA('MYUSER.EDARPC.DATA') SHR`
JCL:	`//EDARPC DD DSN=MYUSER.EDARPC.DATA,DISP=SHR`
DYNAM:	`DYNAM ALLOC FILE EDARPC DA MYUSER.EDARPC.DATA SHR`

Example: Creating a New Data Set

TSO:	`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)`

JCL:	`//EDARPC DD DSN=MYUSER.EDARPC.DATA,DISP=(NEW,CATLG),`
	`//  SPACE=(TRK,(5,3,2)),UNIT=SYSDA,`
	`//  CB=(LRECL=80,RECFM=FB,BLKSIZE=1600)`

DYNAM:	`DYNAM ALLOC FILE EDARPC DA MYUSER.EDARPC.DATA -`
	`SPACE 5,3 TRACKS CATLG DIR 2UNITSYSDA -`
	`LRECL 80 RECFM FB BLKSIZE 1600`

Example: Freeing Files

TSO:	`TSO FREE F(EDARPC)`
DYNAM:	`DYNAM FREE FILE EDARPC`

Example: Concatenating Files

TSO:	`TSO ALLOC F(EDARPC) DA('MYUSER.EDARPC.DATA'-` `'MYUSER.PROGRAMS.DATA') SHR`
JCL:	`//EDARPC DD DSN=MYUSER.EDARPC.DATA,DISP=SHR` `//  DD DSN=MYUSER.PROGRAMS.DATA,DISP=SHR`

DYNAM:	`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

In this section:

How to:

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 6-29.

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 or set output destinations. For more information, see *OFFLINE Printing* on page 6-29.

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 and is only valid 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 FILEDEF OFFLINE feature within iWay has been improved as of release 5.2 to allow the specification of output destinations and, in some cases, additional operating system print command switches for features such as multiple copies.

Prior behavior was that if the operating system variable FOCPRINT was declared (with an operating system command and a $1) then it would be called to take an action on the file name which also replaced the $1 in the string.

The FILEDEF command has been improved to:

```
FILEDEF OFFLINE PRINTER [filename] [ ( PRINTER printername ].
```

The new printer name option is used with the standard print feature of a given platform. If no printer name is declared or is set to blank then the offline file is created, but the print feature is not called. Since printing is platform specific, each platform must be described here individually, however, there are still some common issues that are sometimes best resolved by creating a shell layer that acts as a proxy between the server and the print system.

UNIX and USS

The printer name is dropped in as the "-d" switch value in the "lp -c -d" command. The "-c" switch is used to avoid over-writing of the offline file before actual printing has occurred. If additional lp switches are desired (like multiple copies with -n switch) they may be stacked into the name by enclosing the string in single quotation marks:

```
FILEDEF OFFLINE PRINTER ( PRINTER '29d1 -n 2'
```

If a site uses lpr instead of l,p then an lp shell script can be created in the $PATH before the standard lp command and can act as a proxy to call lpr instead. The lp script could be as simple as "/usr/bin/lpr $*" to redirect lp to lpr.

Note: On USS, the file is spooled to the system and actual disposition will depend on the configuration of the printer spool.

OpenVMS

The printer name is dropped in as the "/QUEUE=" switch value in the "PRINT/QUEUE=" command. OpenVMS always makes a copy of the file to be printed so it does not have an over-writing the offline file problem. If additional PRINT switches are desired (like multiple copies with /COPIES= switch) they may be stacked into the name by enclosing the string in single quotation marks:

```
FILEDEF OFFLINE PRINTER ( PRINTER '29d1 /COPIES=2'
```

Sites rarely use anything but the standard PRINT command, but can be also proxied if necessary by creating an alternate printer command at the OS level or queue / symbiont that routes to the alternate method.

Windows

The printer name is a shared printer name and is used to set up (and later drop a NET USE for the LPT1device to a shared name (for example, \\nodename\myprinter), which is then used in a PRINT /D:LPT1 command to print the actual file. As such, additional switch options can not be done and use of a PRINT.BAT as a proxy is the only method for further manipulation of the output.

OS/400

The output is always spooled from the offline file to the print spool using the system QPRINT file (with whatever the standard values are) on the server's library list. If the spool is set to directly print, then output will always be routed as directed with no declaration of a printer name via FILEDEF. If output is not automatically routed and a printer name (via FILEDEF) is declared, then a CHGSPLFA command will be issued with the printer name as the OUTQ() value to direct the spool file to a destination. If additional CHGSPLFA parameters are desired (like multiple copies with COPIES() parameter) they may be stacked into the name by enclosing the string in single quotation marks:

```
FILEDEF OFFLINE PRINTER ( PRINTER '29d1 COPIES(2)'
```

If an alternate QPRINT file is desired to control some special aspect (like adjusting page size), this must be done by placing an alternate QPRINT in a library before the standard copy in QGPL. Note that the CRTPRTF command for creating printer attribute files does not create a default QPRINT that matches the IBM delivered standard default that is in QGPL. To create a QPRINT with standard IBM delivered default values use:

```
CRTPRTF FILE(*CURLIB/QPRINT) RPLUNPRT(*NO) CHRID(*CHRIDCTL)
```

Then make any site specific changes and change the owner attribute rights from *ALL to *CHANGE (to prevent over-writing) with:

```
EDTOBJAUT OBJ(*CURLIB/QPRINT) OBJTYPE(*FILE)
```

Also note that FOCUS 6.x allowed specifically named printer files and needed the CTLCHAR(*FCFC) option. iWay does not allow specific alternate names and uses the library path to locate and use the first found QPRINT file (standard OS/400 library path behavior) and that use of a printer file with CTLCHAR(*FCFC) will cause page breaks to fail.

The use of an lp script on OS/400 as a proxy is not effective because lp is not used. The use of alternate QPRINT files is the closest equivalent to an lp proxy.

FILEDEF OFFLINE to DISK versus PRINTER

The "(PRINTER printername" feature is only valid in reference to when the FILEDEF device is PRINTER. There is, however, a difference between the use of DISK versus PRINTER as a device in a FILEDEF for OFFLINE. When the device DISK is used, page breaks are represented by a 1 in the first column of a given line where a page break is to occur; this is the FORTRAN Carriage Control method of page control and is a vestige of the product's original mainframe roots. When the disk device is PRINTER, the more modern, Control L (^L / Decimal 12 / Hex 0C) form feed method is used.

Other Printing Issues

Very often sophisticated laser based printers are "hung" off networks and communicated with various print protocols. While these printers may come from many manufactures, a very common (but not standard) attribute of these types of printers is automatic sensing between a plain clear text file being sent to the printer and a postscript file that contains printer attribute commands as well as the text to print. iWay OFFLINE files are plain text (vs. HOLD FORMAT PS which do not get spooled via OFFLINE and it is up to the user to direct to a printer). Very often these sophisticated printers can be set up or used improperly causing a printer to think a plain text file is postscript when it is not and yielding a page with a postscript error message. This has only been seen so far when printing from Windows, but is in theory possible from any platform.

This problem is not considered to be an iWay issue because the software is not directly manipulating these printers and uses standard commands supplied by the OS vendor for printing. Generally, this problem can also be repro'ed using standard print tools stand-alone from an iWay environment. The systems administrator for these printers should be able to track down why this happens in any given environment and take corrective action. However, this may also be corrected by creating a proxy script to inject a leading character into the output that resets the printer. Generally, this is a control D, but printers may vary. Specific implementation of such proxy scripts is left to the customer since needs may vary greatly.

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

Command:	-*
Syntax:	`-* text` where: *text* Is a comment. A space is not required between -* and *text*.
Function:	The command -* signals the beginning of a comment line. Any number of comment lines follows one another, but each must begin with -*. 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. Use comment lines liberally to document a stored procedure so that its purpose and history are clear to others.

Command:	-?
Syntax:	`-? &[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.
Function:	The command -? displays the current value of a local variable.

Command:	-CLOSE
Syntax:	`-CLOSE filename` where: *filename* Is a symbolic name associated with a physical file known to the operating system.
Function:	-CLOSE closes an external file opened with the -READ or -WRITE NOCLOSE option. The NOCLOSE option keeps a file open until the -READ or -WRITE operation is complete. The external file must be defined to the operating system.

Command:	**-AS/400**

Syntax:	`AS/400 command`
	where:
	`command`
	Is an OS/400 command.

Function:	-AS/400 executes an OS/400 operating system command from a procedure.

Command:	**-CMS**

Syntax:	`CMS command`
	where:
	`command`
	Is a CMS command.

Function:	-CMS executes a CMS operating system command from a procedure.

Command:	**-DEFAULTS**

Syntax:	`-DEFAULTS &[&]name=value [...]`
	where:
	`&name`
	Is a name of a variable.
	`value`
	Is the default value assigned to the variable.

Function:	-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: • On the function call EDARPC. • On the command line EXEC. • With the command -SET subsequent to the command -DEFAULTS. By supplying values to variables in a stored procedure, -DEFAULTS helps ensure that it runs correctly.

Command:	-DOS
Syntax:	`-DOS command` where: *command* Is a Windows or DOS command.
Function:	-DOS executes a Windows or DOS operating system command from a procedure.

Command:	-EXIT
Syntax:	`-EXIT`
Function:	-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). Use -EXIT for terminating a stored procedure after processing a final branch that completes the desired task. The last line of a stored procedure is an implicit -EXIT. In other words, the procedure ends after the last line is read.

Command:	-GOTO

Syntax:	`-GOTO label`
	.
	.
	.
	`-label [TYPE text]`
	where:
	label
	Is a user-defined name of up to 12 characters that specifies the target of the -GOTO action.
	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.
	`TYPE text`
	Optionally sends a message to the client application.

Command:	**-GOTO**
Function:	-GOTO forces an unconditional branch to the specified label. If Dialogue Manager finds the label, processing continues with the line following it. If Dialogue Manager does not find the label, processing ends and a message is displayed.

Command:	**-IF**

Syntax:	`-IF expression [THEN] GOTO label1[;]`
	`-[ELSE GOTO label2 [;]]`
	`-[ELSE IF...[;]]`
	where:
	label
	Is a user-defined name of up to 12 characters that specifies the target of the GOTO action.
	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.
	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.
	`;`
	Is required at the end of the command.
	`-`
	Must begin continuation lines.

Command:	**-IF**
Function:	-IF routes execution of a stored procedure based on the evaluation of the specified expression. A -IF without an explicitly specified ELSE whose expression is false continues processing with the line immediately following it.

Command:	**-INCLUDE**
Syntax:	`-INCLUDE filename` where: *filename* Is the name of the called stored procedure.
Function:	-INCLUDE enables one stored procedure to call another one. A stored procedure calls any number of other procedures. Up to four -INCLUDE commands are nested. The called procedure contains fully executable or partial code. The calling procedure cannot branch to a label in the called procedure and vice versa.

Command:	**-label**
Syntax:	`-label [TYPE message]` where: *label* Is a user-supplied name of up to 12 characters that identifies the target for a branch. 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. TYPE *message* Optionally sends a message to the client application.
Function:	A label is the target of a -GOTO or -IF command.

Command:	**-PASS**
Syntax:	`-PASS password` where: *password* Is a literal FOCUS password or a variable containing a password.
Function:	Directly issues and controls passwords. This feature is especially useful for specifying a particular file or set of files that a given user can read or write. Passwords have detailed sets of functions associated with them through the DBA module. Procedures that set passwords should be encrypted so that it and the passwords that it sets cannot be typed and made known. A variable can be associated with -PASS so that you can prompt for and assign a password value using -PROMPT. The PASS command provides the same function at the command level, as does the PASS parameter of the SET command.

Command:	**-PROMPT**
Syntax:	`-PROMPT &name [[.format|.(list)] [.text].]` where: *&name* Is a user-defined variable. *format* Optionally specifies alphanumeric or integer data type and length. *list* Optionally specifies a range of acceptable responses. Must be a comma separated list enclosed in parentheses. *text* Optionally specifies prompting text that appears on the screen. Must be delimited by periods.
Function:	Types a message to the terminal (if edastart -t is in use) or creates an input window with the message in a browser if the connection type is HTTP and reads the reply from the user. This reply assigns a value to the variable named. In edastart -t mode, if a format is specified and the supplied value does not conform, FOCUS displays a message and prompts the user again for the value. In HTTP mode, only a message is displayed. In edastart -t mode, if a (list) is specified and the user does not reply with a value on the list, FOCUS reprompts and prints the list of acceptable values. For HTTP type connections, the list is interpreted as a pull-down list, so only a valid value may be selected. **Note:** You cannot use format and list together.

Command:	**-QUIT**
Syntax:	`-QUIT`
Function:	-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. If the procedure was called by another one, control returns directly to the client application, not to the calling procedure. -QUIT is the target of a branch.

Command:	**-READ**

Syntax:	`-READ filename[,] [NOCLOSE] &name[.format.][,]...`
	
	where:
	
	`filename[,]`
	Is the name of an external file to read, which must be defined to the operating system. A space after *filename* denotes a fixed-format file, while a comma after *filename* denotes a free-format file.
	
	`NOCLOSE`
	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.
	
	`&name[,]...`
	Is a list of variables. For free-format files, you may separate the variable names with commas, but it is not necessary.
	
	`.format.`
	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.

Command:	**-READ**

Function:	-READ enables the reading of data from an external file that is defined to the operating system.
	
	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.
	
	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.
	
	`-READ EXTFILE, &CITY, &CODE1,`
	`- &CODE2`
	
	If you are reading a fixed-format file, begin the next line with a hyphen and comma.
	
	`-READ EXTFILE &CITY.A8. &CODE1.A3.,`
	`-, &CODE2.A3.`

Command:	**-REMOTE**

Syntax:	`-REMOTE [BEGIN|END]`
	
	where:
	
	`BEGIN`
	Specifies the start of commands on an originating server to be sent to a target server.
	
	`END`
	Specifies the end of commands from the originating server.

Function:	-REMOTE commands are the initial form of stored procedure routing, and are available with Hub Services only. Dialogue Manager commands within the delimiters are executed, and variable substitution takes place before the stack is sent to the target server. A -INCLUDE command takes a Dialogue Manager procedure residing on the originating server and includes the procedure commands in the stack. The commands within the delimiters must make up a complete server request. Any command valid on the target server is included. The command EXEC may be included within the delimiters to execute a stored procedure on the target server. -REMOTE commands cannot be nested.

Command:	**-REPEAT**

Syntax:	`-REPEAT label n TIMES`
	or
	`-REPEAT label WHILE condition`
	or
	`-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 (you cannot change the number of times to execute the loop).
	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. 1 is the default value.
	TO *toval*
	Is a value against which *&variable* is tested. 1,000,000 is the default value.
	STEP *s*
	Is a constant used to increment *&variable* at the end of each execution of the loop. It may be positive or negative. 1 is the default value.

Command:	-REPEAT
Function:	-REPEAT allows looping in a stored procedure. The parameters FROM, TO, and STEP appear in any order. A loop ends when: • It is executed in its entirety. • A -QUIT or -EXIT is issued. • 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.

Command:	-RUN
Syntax:	`-RUN`
Function:	-RUN causes immediate execution of all stacked commands. Following execution, processing of the stored procedure continues with the line that follows -RUN. -RUN is commonly used to: • Generate results from an SQL request that are then used in testing and branching. • 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.

Command:	-SET
Syntax:	`-SET &[&]name=expression;` where: *`&name`* Is the name of a variable whose value is to be set. *`expression`* Is a valid expression. Expressions occupy several lines, so end the command with a semicolon.

Command:	**-SET**
Function:	-SET assigns a literal value to a variable, or a value that is computed in an arithmetic or logical expression. 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.

Command:	**-TSO RUN**
Syntax:	`-TSO RUN command` where: *command* Is a TSO command.
Function:	-TSO executes a TSO operating system command from a procedure.

Command:	**-TYPE**
Syntax:	`-TYPE text` where: *text* 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. The length of text can be up to 256 bytes.
Function:	-TYPE sends a message to a client application. Any number of -TYPE commands can follow one another but each must begin with -TYPE. Variables may be embedded in the message. The values currently assigned to each variable are displayed.

Command:	**-UNIX**
Syntax:	`-UNIX command` where: *command* Is a UNIX command.

Command:	**-UNIX**
Function:	-UNIX executes a UNIX operating system command from a procedure.

Command:	**-VMS**

Syntax:	`-VMS command`
	where:
	`command`
	Is a VMS command.

Function:	-VMS executes a VMS operating system command from a procedure.

Command:	**-WINNT**

Syntax:	`-WINNT command`
	where:
	`command`
	Is a Windows or DOS command.

Function:	-WINNT executes a Windows or DOS operating system command from a procedure.

Command:	**-WRITE**

Syntax:	`-WRITE filename [NOCLOSE] text`
	where:
	`filename`
	Is a symbolic name for a physical external file being written to. The file name must be known to the operating system.
	`NOCLOSE`
	Keeps the external file open until the -WRITE operation is complete. Files kept open with NOCLOSE are closed with the command -CLOSE filename.
	`text`
	Is any combination of variables and text.

Command:	**-WRITE**
Function:	-WRITE writes data to an external file. 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. Unless you specify the NOCLOSE option, an opened file is closed upon termination of the procedure with -RUN, -EXIT, or -QUIT.

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 API programs to converse with a server.

Using GENCPGM

How to:

Compile and Link a Procedure

Run GENCPGM

Display GENCPGM Options

Example:

Generating an API Program From a C Source File

Generating an API Program From a C++ Source File

Generating a CALLPGM Program Library From a C Source File

Generating a Routine with the MTHNAME Sample Routine

Reference:

GENCPGM Parameters

A script has been created for UNIX, OS/400, Windows NT/2000, and OpenVMS to assist in simple compilations of the provided C examples. On the respective platforms, the names are gencpgm.sh (also for OS/400), gencpgm.bat, and gencpgm.com. The script is located in the bin directory of EDAHOME.

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. GENCPGM is provided as a tool to expedite program creation in many, but not all cases.

On some platforms, the GENCPGM script supports languages other than C and has had testing for these other languages. While theoretically any language can be used to create a program, 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 and are up to the developer to create their own build scripts. In complex cases, the GENCPGM script may be used as a model for forming an application-specific script.

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 EDAHOME bin directory to your working directory or use the full path name to the location.
   - 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 EDAHOME to your working directory.
   - For a CALLPGM program or to build the CALLPGM sample program (CPT, SPG*.CBL, or SPG*.RPG), copy the sample program and any required include files from the etc directory of EDAHOME to your working directory.
   - For user exits, copy the desired sample exit from the etc directory of EDAHOME to your working directory. There are MTHNAME samples for C, COBOL, RPG, and Fortran.
   - For user routines, write the routine or copy an existing routine to your working directory.
2. Issue an environment variable for the EDAHOME directory.
3. If building an API program, also issue an environment variable for the EDACONF directory.

Syntax: How to Run GENCPGM

```
gencpgm[.sh] [-g] [-q] [-x] [-m [ option ]][-c [ option ]] prog
```

where:

prog

Is the name of the procedure to be compiled and linked. Generally, a file extension does not need to be supplied, but may be if nontypical extensions are used.

Note: OpenVMS requires use of a leading "@" for DCC execution, and UNIX and OS/400 require the .sh extension to be explicitly used.

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 useage.

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 set -x for shell tracing.	No value needed.	
-m	Specifies the type of module to create. If this parameter is omitted, the default is -m cpgm.	api	Creates an API program.
		api m	Same as API, but uses XXX dll linkage (OS/390 only).
		cpgm	Creates a CALLPGM program, a user exit, or a routine.

Parameter	Use	Possible Values	
-c	Specifies the type of compiler to use. If this parameter is omitted, the default will be -c cc.	cc	Uses the standard C compiler to compile prog.c.
		CC	Uses the C++ compiler to compile prog.cpp. (UNIX)
		cxx or c++ cpp	Uses the C++ compiler to compile prog.cpp. (OpenVMS, Windows, and UNIX)
		gcc	GNU C compiler
		fortran for f	Uses the Fortran compiler. (OpenVMS only) Parameter values drive the default file extension unless the full file name is used to specify the file.
		cobol cob cbl	Uses the COBOL compiler. (OpenVMS and OS/400) Parameter values drive the default file extension unless the full file name is used to specify the file.
		RPG	Uses RPG compiler to compile pro.rpg. (OS/400 only)
		DDS	Use to compile OS/400 DDS files for use with RPG and other OS/400 languages.

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 the files need 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. User exits are not explicitly covered in this manual, but follow the same rules as a routine.

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
```

For actual CALLPGM code samples, see Chapter 4, *Writing a 3GL Compiled Stored Procedure Program*.

Example: Generating a Routine with the MTHNAME Sample Routine

The following example is a routine that is used in Dialogue Manger to translate a month number into a spelled out name. This sample uses C version, but this sample and matching mthname samples in COBOL, RPG, and Fortran are included with the product in the EDAHOME etc subdirectory.

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

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

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

Web form: http://www.informationbuilders.com/bookstore/derf.html

Name:_____

Company:_____

Address:_____

Telephone:_____Date:_____

E-mail:_____

Comments:

Reader Comments

Information Builders, Two Penn Plaza, New York, NY 10121-2898 **(212) 736-4433**