Information Builders

WebFOCUS

Stored Procedure and Subroutine Reference for 3GL Languages

WebFOCUS Reporting Server Release 8205
DataMigrator Server Release 7709 and Higher

June 04, 2019

Contents

Preface

This content provides information about the building and integration of compiled and linked 3GL stored procedures and subroutines (DLLs) for use within FOCEXEC applications. A 3GL-based stored procedure allows users to build access to data sources not already supported by the server. Use of a 3GL-based subroutine allows one to do specialized calculations at a column level as part of a DEFINE or a COMPUTE. This content is intended for the API Programmer, the FOCEXEC Programmer, and others who develop and maintain client/server applications.

For up-to-the-minute information, please refer to the release notes.

How This Manual Is Organized

This manual includes the following chapters:

	Chapter/Appendix	**Contents**
1	Introducing Stored Procedures and Subroutines	Describes the types of stored procedures and subroutines, how they are called, and their execution order. Explains why stored procedures and subroutines are used.
2	Calling a Program as a Stored Procedure	Describes ways to call a compiled program using the commands CALLPGM or EXEC in a Dialogue Manager FOCEXEC procedure. Addresses the use of parameters.
3	Calling a JAVA Class as a Stored Procedure	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 CALLPGM in a Dialogue Manager FOCEXEC 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	User Written Routines	Describes how to call user written subroutines.
6	Using the GENCPGM Build Tool	Describes how to use the script for UNIX, Windows, and OpenVMS to assist in simple compilations.

	Chapter/Appendix	Contents
7	Additional 3GL Reference Examples	Describes how write and compile a 3GL user-written subroutine in multiple languages.

Conventions

The following table describes the conventions that are used in 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.
<u>`underscore`</u>	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 that you can click or select.
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 (...).

Convention	Description
. . .	Indicates that there are (or could be) intervening or additional commands.

Related Publications

Visit our Technical Content Library at *http://documentation.informationbuilders.com*. You can also contact the Publications Order Department at (800) 969-4636.

Customer Support

Do you have any questions about this product?

Join the Focal Point community. Focal Point is our online developer center and more than a message board. It is an interactive network of more than 3,000 developers from almost every profession and industry, collaborating on solutions and sharing tips and techniques. Access Focal Point at *http://forums.informationbuilders.com/eve/forums*.

You can also access support services electronically, 24 hours a day, with InfoResponse Online. InfoResponse Online is accessible through our website, *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.

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 questions. Information Builders consultants can also give you general guidance regarding product capabilities. Please be ready to provide your six-digit site code number (*xxxx.xx*) when you call.

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*).
- The server software version and release. You can find your server version and release using the Version option in the Web Console.
- 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?
  - Provide 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 Technical Content Management staff welcomes your opinions regarding this document. You can contact us through our website *http://documentation.informationbuilders.com/connections.asp*.

Thank you, in advance, for your comments.

Software Training and Professional Services

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

For information on course descriptions, locations, and dates, or to register for classes, visit our website (*http://education.informationbuilders.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 website (*http://www.informationbuilders.com/support*).

Chapter 1

Introducing Stored Procedures and Subroutines

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 such as WebFOCUS but can also be called 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 and linked 3GL program, written in a language such as C or COBOL, which is located and called on a server or gateway process.
- A file of 4GL command syntax (known as a FOCEXEC) written in a combination of Dialogue Manager (DM) syntax and/or other commands, such as SET, TABLE FILE, SQL, and DBMS passthru that run on the server.

A subroutine is a specialized 3GL-based routine that is compiled and linked as a DLL and then called in-line within a 4GL command syntax request, thus integrating 4GL syntax with specialized 3GL calculations. It can be thought of as a specialized calculation integrated at the table column level, although the same subroutine might also be applicable for use in a Dialogue Manager calculation.

This manual describes the use of compiled and 3GL programs and subroutines. Full 4GL FOCEXEC syntax is fully covered in other documents, such as the *Developing Reporting Applications*, *Creating Reports With WebFOCUS Language*, and *Server Administration* manuals. They are not covered in this manual.

In this chapter:

Using a Stored Procedure

The ability to use a DBMS 3GL stored procedure is limited to what an underlying product (DBMS) supports and varies by platform. Any limitations will be noted in the DBMS documentation.

Stored procedures allow the execution of other procedures and the inclusion of procedures, thus allowing a very modular environment that enables 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.

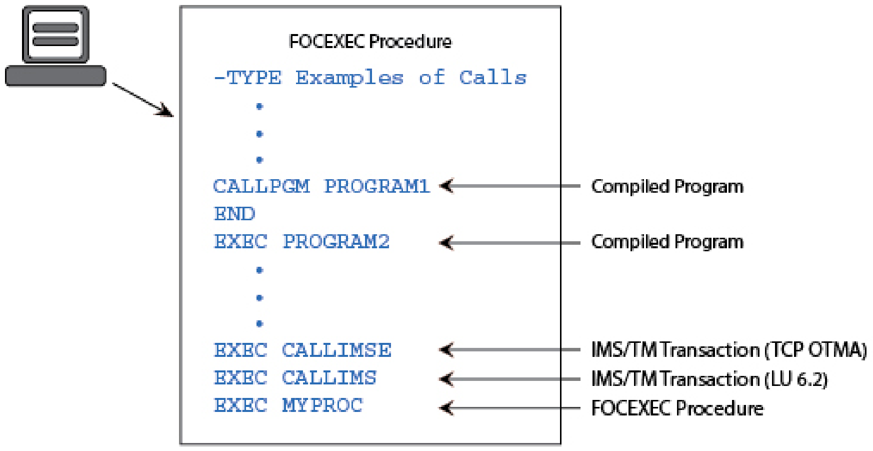
Calling a Stored Procedure

An application typically executes a stored procedure though a front-end tool such as an API call to the server or any number of WebFOCUS mid-tear tools. The initial procedure is usually a 4GL Dialogue Manager procedure that, in turn, calls other 4GL Dialogue Manager procedures, compiled 3GL procedures, or a combination of both.

The following types of procedure calls are supported:

- ❏ A direct call to a compiled 3GL procedure using EDARPC (a sub-feature of EDAAPI, which has been deprecated).
- ❏ A 4GL 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 (which is not covered in this manual).

The following figure shows calls to stored procedures from Dialogue Manager.



```
-TYPE Examples of Calls
   .
   .
   .
CALLPGM PROGRAM1
END
EXEC PROGRAM2
   .
   .
   .
EXEC CALLIMSE
EXEC CALLIMS
EXEC MYPROC
```

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 FOCEXEC Procedure	Server Procedure Library. The external names, EDARPC (MVS) or EDAPATH (all other platforms), are used to locate Dialogue Manager FOCEXEC 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.

Type of Stored Procedure	Library
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

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 FOCEXEC 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 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 FOCEXEC procedure in search path of the server. If it is, the server will find the 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 FOCEXEC procedure, the name of the procedure cannot be the same as the name of a program in the search path of the server. If it is, the server will find the program in the Program Library and execute it, rather than executing the Dialogue Manager FOCEXEC 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

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

Using CALLPGM

If you use explicit CALLPGM syntax in a Dialogue Manager FOCEXEC 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 FOCEXEC 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 FOCEXEC procedure.

Using CALLIMS or CALLITOC

The CALLIMS and CALLITOC programs contain 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 FOCEXEC procedure, the server recognizes that you are calling a compiled program, and IBICPG does not need to be set.

Using a Subroutine

The ability to use a compiled and linked 3GL subroutine is dependent on its being used *in-line* within a COMPUTE or DEFINE of a 4GL request or within a Dialogue Manager calculation. Unlike a 3GL stored procedure that returns data sets of data, a subroutine is used for an individual calculation.

For example, in the following Dialogue Manager procedure, the second -SET command calls the MTHNAME subroutine, which takes a number as an argument and does a lookup call for the corresponding month name:

```
-SET &MTHNUMBER = 1 ;
-SET &MTHNAME = MTHNAME(&MTHNUMBER,'A13') ;
-TYPE Month &MTHNUMBER is &MTHNAME
```

In the following sample, a COMPUTE command in a TABLE request calls the MTHNAME subroutine, which takes a number as an argument and does a lookup call for the corresponding month name:

```
TABLE FILE ...
COMPUTE  MTHNAME = MTHNAME(MTHNUMBER,'A13') ;
END
```

Chapter 2

Calling a Program as a Stored Procedure

The following are ways to call a compiled program. You can use one of the following:

- The CALLPGM command.
- The EXEC command in a procedure.

Either of these methods enables you to pass parameters to programs and Dialogue Manager FOCEXEC procedures.

Once the source language is built, the interface is agnostic of the underlying original language.

In this chapter:

Calling a Compiled Program

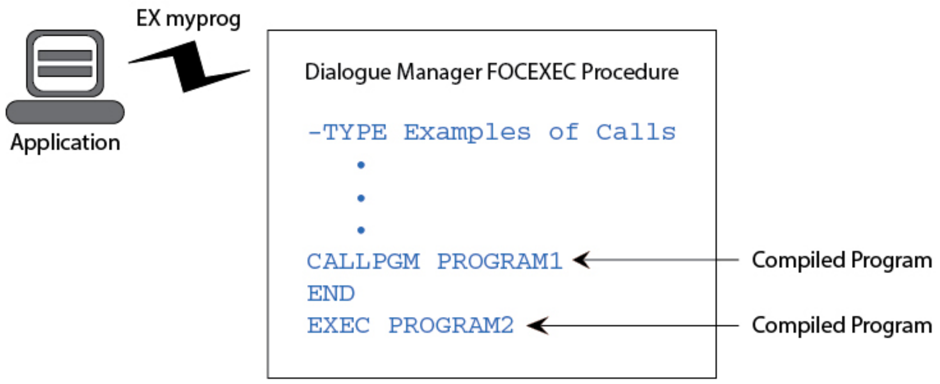
The program is called on the server in the following ways:

- 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* on page 18. 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.

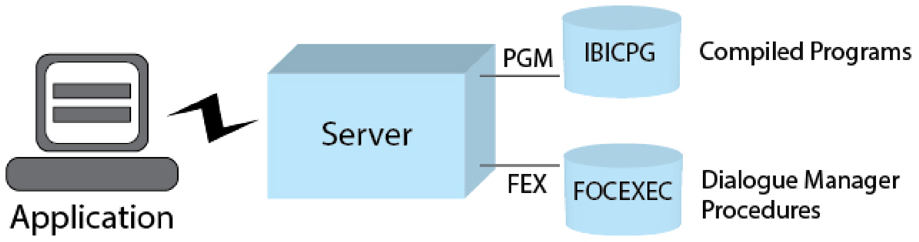
The term *program* is also used to refer to a compiled program.

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



```
-TYPE Examples of Calls
   .
   .
   .
CALLPGM PROGRAM1
END
EXEC PROGRAM2
```

The following figure illustrates the libraries in which compiled programs and Dialogue Manager FOCEXEC procedures reside. See *Introducing Stored Procedures and Subroutines* on page 13, for details on stored procedure libraries and stored procedure execution order.



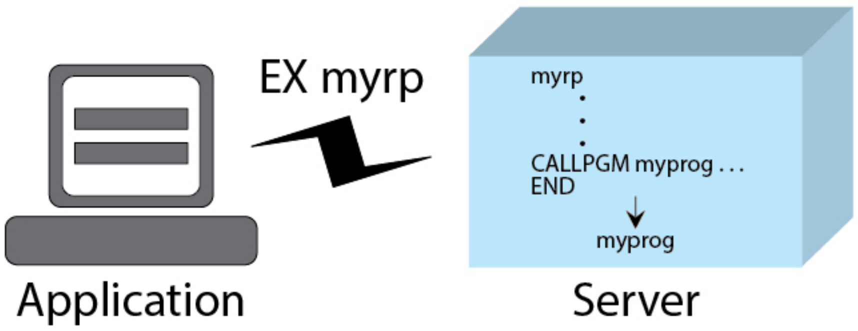
Calling a Program With CALLPGM or EXEC

Application developers use Dialogue Manager FOCEXEC procedures 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 FOCEXEC procedure.



The steps in this process are:

1. The Dialogue Manager FOCEXEC procedure is located and executed by the server. The command CALLPGM myprog within the procedure finds and loads the external procedure and it is run. The END statement after the CALLPGM line is required syntax to end stacking on input to the application.
2. 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 27 for more information.
3. CALLPGM performs one or both of the following actions, which are transparent to the Dialogue Manager FOCEXEC procedure:

   - Passes a message or messages to the client application for processing. The client application issues internal function calls 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 *Writing a 3GL Compiled Stored Procedure Program* on page 55.

Rows of data, which the client application retrieves using internal function calls.

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

The command CALLPGM and EXEC operate 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 Using CALLPGM or EXEC

```
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 FOCEXEC 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 27 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 Using SQL EX

```
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 FOCEXEC 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 27 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.

SQL Procedures and Db2 PLAN (z/OS Db2 CAF Adapter Only)

The z/OS Db2 CAF adapter 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 z/OS Db2 CAF 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 z/OS Db2 CAF

```
ENGINE DB2 SET PLAN progplan
CALLPGM myprog...
END
ENGINE DB2 SET PLAN ' '
```

where:

`progplan`

Is the name of the plan required by the program.

`myprog`

Is the name of the program to be run.

`SET PLAN ' '`

Resets the plan.

An alternative is to use z/OS Db2 CAF 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 PGMSQL. 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: PGMSQL, SPG1, and SPG2. Create three packages, called PGMSQL.PGMSQL, PGMSQL.SPG1, and PGMSQL.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.

Storing Answer Set Data

When executing a CALLPGM stored procedure, it is sometimes desirable to retain the answer set on the server.

Example: Processing an Answer Set on the Server

The following example illustrates the method used to retain the answer set on the server and assumes the called program is a fex with syntax to call the actual external procedure:

```
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 the *Stored Procedures* chapter of the *Server Administration* manual.

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

Passing Parameters

The following terminology is used in this section:

❑ Amper variables used in a Dialogue Manager FOCEXEC procedure are also called DM variables.

❑ Parameters in a Dialogue Manager FOCEXEC 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 with Embedded Spaces in Parameters

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

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

where:

<u>ON</u>

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 CALLPGM

Parameters are passed to CALLPGM as comma separated values. If a value is a string and contains an embedded space or comma then the string must be quoted. The SET STRIPQUOTE default is to strip the quotes for the underlying program so only the value is seen.

The underlying program then uses the CPM specification to read the individual parameters. The parameter order must match the parameter order that the underlying program expects, but there is also no reason why the underlying program cannot be written to understand parameter pair values (for example, firstname=John,lastname=Doe) so that the program is position independent. It is also permissible to form all values as a single quoted string, if the underlying program has code to parse the string into usable values.

For example assume myproc is a Dialogue Manager FOCEXEC procedure and the procedure contains and sets values for the variables &1, &2, and &3 and issues a CALLPGM command as follows:

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

When the procedure executes, the server substitutes the values for the variables &1, &2, and &3, and the result call might look like:

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

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

Example: **Passing Long Parameters**

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 break up long parameters into more readable strings and internally pass the long parameter 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 LINE000000000000000000000000000000000000000001     "
"LINEOFINFORMATION1111111111111111111111111111111111111111111111111111111"
"LINEOFINFORMATION2222222222222222222222222222222222222222222222222222222"
"LINEOFINFORMATION4004004004004004004004004004004004004004004004004004004"
.
.
.
"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.
- The value after -LINES is the number of lines to read for parameters. In this example, for brevity, several hundred lines are not shown.

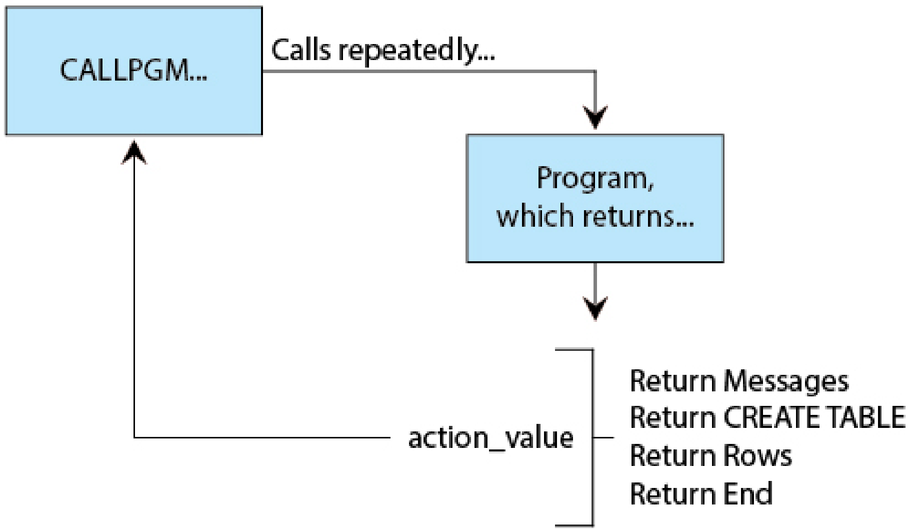
Program Communication

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.

For more information, including the specific values to be returned in an action_value, see *Writing a 3GL Compiled Stored Procedure Program* on page 55.

The process is illustrated below.



Chapter 3

Calling a JAVA Class as a Stored Procedure

You can easily access a JAVA class in your application, much as you would access an external 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 and receive data back.

CALLJAVA is also known as the Call Java Adapter.

In this chapter:

Execute Using CALLJAVA

You can invoke a user-written JAVA class with the CALLJAVA command from your user session. This usage assumes the desired class is in a jar on the CLASSPATH of the running server.

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 34.

Example: **Calling ibi.cjsamples.cjsamp Using CALLJAVA**

```
CALLJAVA ibi.cjsamples.cjsamp,parameter1,
 "subparm1=val1,subparm2=val2",simple parameter3
```

Execute Using EX

You can invoke a user-written JAVA class with the EX command from your user session if SET EXORDER is also used when considering external programs for execution. This usage assumes the desired class is in a jar on the CLASSPATH of the server.

Syntax: **How to Use EX to Execute a JAVA Class**

```
EX java.classparameter1, 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 34.

Example: **Calling ibi.cjsamples.cjsamp Using EX**

```
SET EXORDER=PGM/FEX
EX java.ibi.cjsamples.cjsamp parameter1,
 "subparm1=val1,subparm2=val2",simple parameter3
```

Execute Using SQL EX and SQL CPJAVA EX

Using SQL EX or SQL CPJAVA 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.

You can invoke a user-written JAVA class with the EX command if SET EXORDER is also used when considering external programs for execution.

Syntax: How to Use SQL EX to Execute a JAVA Class

```
SET SQLENGINE=CPJAVA
SQL EX classparameter1,, parameter2, ... ;
TABLE FILE SQLOUT
PRINT * [ON TABLE [PC]HOLD]
SET SQLENGINE=OFF
```

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 34.

Syntax: How to Use SQL CPJAVA EX to Execute a JAVA Class

```
SET CPJAVA EX classparameter1,parameter2, , ... ;
TABLE FILE SQLOUT
PRINT * [ON TABLE [PC]HOLD]
END
```

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 34.

The trailing semi-colon is required syntax. Either syntax is valid and one may be stylistically better for any given application.

Example: Calling ibi.cjsamples.cjsamp Using SQL CPJAVA EX

```
SQL CPJAVA EX java.ibi.cjsamples.cjsamp parameter1,
   "subparm1=val1, subparm2=val2", simple parameter3;
TABLE FILE SQLOUT
PRINT *
END
```

Passing Parameters

You must adhere to the following usage requirements when passing parameters:

- 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 that need to be retained in double quotation marks.
- Two consecutive commas do not represent a null parameter. To pass a blank parameter, use " " or code a keyword, such as null or blank, as an application flag.
- A parameter is generated for an unbalanced double quotation mark; quotes should never be unbalanced.

For information on parameter parsing techniques, see the example under *Compiling and Running a JAVA Program* on page 42.

Example: Passing Parameters

The following command, based on the sample later in this chapter, invokes the JAVA class java.ibi.cjsamples.cjsamp with three parameters:

```
EX java.ibi.cjsamples.cjsamp Parameter1, " ", ""Parameter3""
```

Writing a JAVA Class

When you write a JAVA class to be invoked by the Reporting 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 passthru” mode. The execute method is used to instantiate column attributes (name, data type, and size), and return the instantiated IBI Answer Set object, populated with the answer set description, to the server.

- The fetch method populates the object with data and is invoked by the server to receive one row of the answer set at a time. The IBI_EOD flag is returned when the answer is finished; the IBI_DATA flag is returned to indicate that more data is coming.

Note: The CALLJAVA interface is also used internally, and, as such, requires an additional execute method as a signature returning a null to exist in user application so the two uses can co-exist. The additional method is:

```
public ibianswr
execute(String username, String password, Object obj, String[] parms)
             throws Exception { return null; }
```

Tip: This signature is already included in the sample application, so if the sample is just cloned as a template for another application, the signature will already properly placed.

For applications built prior to 7.6.3 to work with 7.6.3 (or higher) servers, the signature must be added and the application rebuilt. The current sample can be used to determine where to place the signature in older applications (current sample JAVA CALLPGM code is shown in *Compiling and Running a JAVA Program* on page 42).

Please note that while you must include this code, you may neither modify nor use it.

The Java Logging API

A logging api is provided to assist in transaction logging, tracing and debugging. The api follows the industry practices followed by many providers. If you have experience working with other widely accepted Java logging packages (for example, java.logging, org.apache.commons.logging, log4j, slf4j), then you will be familiar with these methods.

Note: This API replaces the deprecated ibtrace methods in earlier releases.

Basic Logging API

Assume you need to add logging capabilities to the following class:

```
public class Foo
{
   public static String concatenate(String s1, String s2)
   {
      return s1 + s2;
   }
}
```

Add static member of type ibi.trace.ILogger (commonly named “log”) to the class, then use it to log actions within the code:

```
import ibi.trace.ILogger;
import ibi.trace.IBILogFactory;

public class Foo
{
   private static final ILogger log = IBILogFactory.getLogger(Foo.class);
   public static String concatenate(String s1, String s2)
   {
      log.debug("About to concatenate ''{0}'' and ''{1}''", s1, s2);
      return s1 + s2;
   }
}
```

Logging Levels

The logging API supports the following levels:

- **Debug.** For detailed logging.
- **Info.** For informational messages.
- **Warn.** For warnings.
- **Error.** For errors.

Each level allows formatted output and exception logging. For example, at warn level:

```
log.warn("About to sleep 100 millis");
try
{
   Thread.sleep(100);
}
catch(InterruptedException ex)
{
   log.warn("Warning: interrupted", ex);
}
```

Java Logging and Server Tracing

When the server is started with the -traceon option, java logging is enabled at the debug level (this level automatically includes all other levels).

The special logger name edaprint.log can be used to duplicate logging messages to the edaprint.log file of the server. Only error, warn, and info levels can go to edaprint.log. Also, edaprint.log logger is always enabled, regardless of whether server was started with traces on or off.

The use edaprint.log method should be done sparingly - only important and critical events should be put to edaprint.

The following code illustrates the edaprint.log logging functionality:

```
public class Foo
{
   private static final log      = IBILogFactory.getLogger(Foo.class);
   private static final edaprint = IBILogFactory.getLogger("edaprint.log");
   public static void concatenateAndWrite(Writer writer, String s1, String s2)
   {
      log.debug("About to concatenate {0} and {1}", s1, s2);
      String s = s1 + s2;
      log.debug("Result is: {0}", s);
      try
      {
         writer.write(s);
      }
      catch(IOException ex)
      {
         // this sends error report to the server's edaprint.log
         // regardless of server tracing setting.
         // Additionally, if traces are on, the error report
         // will be sent to the standard trace facility
         edaprint.error("Failed to write", ex);
      }
   }
}
```

Logging Reference Guide

The following sections describe different logging levels.

Debug Level

```
public boolean isDebugEnabled();
```

Returns true if logging is enabled at the Debug level. Useful to avoid time-consuming computations that are needed only for logging purposes.

```
public void debug(String message);
```

Sends message string to the logging destination, but only if logging is enabled at the debug level.

```
public void debug(String messageFormat, Object arg1);
public void debug(String messageFormat, Object arg1, Object arg2);
public void debug(String messageFormat, Object arg1, Object arg2, Object arg3);
```

Formats message and sends the result to the logging destination, but only if logging is enabled at the debug level.

```
public void debug(String messageFormat, Throwable th);
```

Prints message and exception trace to the logging destination, but only if logging is enabled at the debug level.

Info Level

```
public boolean isInfoEnabled();
```

Returns true if logging is enabled at the Info level. Useful to avoid time-consuming computations that are needed only for logging purposes.

```
public void info(String message);
```

Sends message string to the logging destination, but only if logging is enabled at info or higher level.

```
public void info(String messageFormat, Object arg1);
public void info(String messageFormat, Object arg1, Object arg2);
public void info(String messageFormat, Object arg1, Object arg2, Object arg3);
```

Formats message and sends the result to the logging destination, but only if logging is enabled at info or higher level.

```
public void info(String messageFormat, Throwable th);
```

Prints message and exception trace to the logging destination, but only if logging is enabled at info or higher level.

Warn Level

```
public boolean isWarnEnabled();
```

Returns true if logging is enabled at the Warn level. Useful to avoid time-consuming computations that are needed only for logging purposes.

```
public void warn(String message);
```

Sends message string to the logging destination, but only if logging is enabled at warn or higher level.

```
public void warn(String messageFormat, Object arg1);
public void warn(String messageFormat, Object arg1, Object arg2);
public void warn(String messageFormat, Object arg1, Object arg2, Object arg3);
```

Formats message and sends the result to the logging destination, but only if logging is enabled at warn or higher level.

```
public void warn(String messageFormat, Throwable th);
```

Prints message and exception trace to the logging destination, but only if logging is enabled at warn or higher level.

Error Level

```
public boolean isErrorEnabled();
```

Returns true if logging is enabled at the Error level. Useful to avoid time-consuming computations that are needed only for logging purposes.

```
public void error(String message);
```

Sends message string to the logging destination, but only if logging is enabled at error or higher level.

```
public void error(String messageFormat, Object arg1);
public void error(String messageFormat, Object arg1, Object arg2);
public void error(String messageFormat, Object arg1, Object arg2, Object arg3);
```

Formats message and sends the result to the logging destination, but only if logging is enabled at error or higher level.

```
public void error(String messageFormat, Throwable th);
```

Prints message and exception trace to the logging destination, but only if logging is enabled at error or higher level.

The Java ibtrace Tracing Interface (Deprecated)

The ibi.trace methods described here are considered deprecated as of Version 7 Release 7, but remain (with one exception) for backward compatibility purposes. The normal edastart -traceon facility captures the trace interactions between the user (TSCOM3) process, the CALLJAVA API and any ibtrace.println() statements that may be in the application. The user process interactions are stored in the normal TS###### trace files and the Java traces are stored in the JS###### trace files of the Java service, both in the EDATEMP directory.

The numbering of the JS###### files is based on the current JAVA execution number. As a reference, in the EDATEMP directory, you will also find JSCOM3 and JSCOM3_J trace files related to the start up of the Java service.

The API also includes ibtrace class methods for the Java application to more precisely control trace activities in which the main traces can be off, yet specific events using ibtrace.println() can be tracked, much like an activity log.

Tracing is typically used when an exception is thrown and a user wants to send special information to the process trace (which may or may not be active) before taking the next step, possibly performing some remediative action.

```
} catch (Exception e)
  {
    ibtrace.println(ClassName + ": Parm One Not Integer: " + e + "\n");
    Rows = 1; /* Reset to 1 if not numeric */
  }
```

Reference: Trace Methods and Uses

The following chart lists a complete set of trace methods and typical uses.

Method	Typical Use
closeTrace( )	Closes an open trace file. This is required before the first initTrace() call in order to close the internal default file. If not supplied, the initTrace() is ineffective and the prior trace file remains active.
initTrace(String fname)	Is used after closeTrace() to initialize and provide an alternate trace file name. If a previously used file is provided, it is opened as new and the contents are over-written.
initTrace(String fname, append flag)	Is used after closeTrace() to initialize and provide an alternate trace file name with a boolean flag (true/false) to indicate appending to an existing file. An append flag of false is effectively the same as initTrace(String fname).
println(String msg)	Prints a text message into a trace file.
println(Exception e + String id)	Prints an exception into a trace file with a string ID.
traceOn( )	Enables Write operations for trace file. This is the default if the server was start with tracing.
traceOff( )	Disables Write operations for trace file. This is the default if the server was started without tracing.
isTraceOn( )	Checks for tracing on/off status to control logic flow. The reply response is "true" or "false."

Note: Prior releases supported a method of println(exception e), however, it was necessary to remove the method and not just deprecate it. It is easily replaced by concatenating with a string (for example, +"\n").

The ibiAnswerSet Interface

```
package ibi.callpgm;

public interface ibiAnswerSet {

public static final int IBI_ALPHA
public static final int IBI_INTEGER
public static final int IBI_FLOAT
public static final int IBI_DOUBLE
public static final int IBI_TIME
public static final int IBI_DATE
public static final int IBI_TIMESTAMP
public static final int IBI_SMALLINT
public static final int IBI_BIGINT
public static final int IBI_DECIMAL
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);
}
```

See *Compiling and Running a JAVA Program* on page 42 for an illustration of how setColName, setColType, setColSize, and setColValue are used.

The 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 regardless of execute method used, 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 ntj2c.jar file (and the ibtrace.jar file when including any ibtrace or log method), located in the EDAHOME etc/java/java16 (7706/81x releases) or etc/java/srvr (7707/82x and higher releases) subdirectory, needs to be accessible. If you are manually building with Javac, you can then compile your JAVA program using the CLASSPATH environment variable or the javac -classpath command parameter. If GENCPGM is used to build your JAVA program, including the jars is automatic. Specific details follow the sample.

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 Reporting Server or configure the JVM environment of the server using the Web Console to include the jars.

When writing your own application, create a .java file containing the application, then build and run as described in the sections that follow.

Note that the following is a working example that needs no customization. Users are expected to build and test this example in order to confirm a properly configured and usable environment for CALLPGM applications before attempting any custom applications.

The example has two modes: echo back the command line parameters sent to it or display data of the various supported data types. The example does not work with or use external data so it has no external dependencies, such as a configured DBMS. However, in practical use, a real application would typically retrieve data from an external source. (It is the users responsibility to do the real coding of the application.)

Example: Compiling and Running a JAVA CALLPGM Program

You can use the following sample code (cjsamp.java) as a model for writing your own JAVA program.

The main mode is to echo parameters back, and, if the first parameter is a number, the application echoes the parameters back as individual columns of that many tuples of data. If the first parameter is not numeric, the exception handler detects and sets the tuple return count to 1. A number greater than 50 also activates a safety (against excessive processing), and resets the tuple return count to 9. (You can change the safety, if that is what you want. For this example, the safety is simply a precaution against excessive records for the demo sample.)

If the first parameter is numeric and the second is the word datatype, the second test mode activates and returns arbitrary data in the various supported data types, for as many tuples as specified. Within this mode, passing a null for string is used for one of the columns, which, in turn, display as MISSING data (dot) within the tuple sent back.

Remember that a real call java program would use the structure of this example to parse the incoming parameters and take some action, such as setting column sizes and returning data from an sql source, but the coding is left to the user.

```
/*
 * Title:        cjsamp.java
 * Description:  Example implementation of a JAVA callpgm program
 * Company:      Information Builders, Inc.
*/
/*
   This example has two modes, echo back the command line parms sent to
   it or display data of the various supported data types. It does not
   work with/use external data so has no external dependencies such as a
   configured DBMS, but in practical use would retrieve data from a
   external source; it is left to the user to do such coding.


   The main mode is to echo parms back and if the first parm is a number,
   the application echoes the parms back as individual columns of that
   many tuples of data. If the first parm is not numeric, the exception
   handler detects and sets the tuple return count to 1. A number greater
   than 50 also activates a safety (for excessive processing) and resets
   the tuple return count to 9 (the user is free to change the safety,
   if that is what they really want, it is simply a precaution for the
   example). If no parms are passed, it is detected and a no parms
   message passed back.


   If the first parm is numeric and the second is the word "datatype",
   the second test mode activates and returns arbitrary data in the
   various supported data types for as many tuples as specified. Within
   this mode, the use of passing a null for string is used for one of
   the columns, these in turn display as MISSING data (dot) within the
   tuple sent back.


   As already stated, a real call java program would use the structure
   of this example to parse the incoming parms and take some action
   such as setting column sizes and returning data from an sql source,
   but is left to the user to code.


   Execution syntax is (see full documentation for parm rules):
     CALLJAVA ibi.cjsamples.cjsamp, parm1, parm2, ...
     END
   or
     SET EXORDER=PGM/FEX
     EX java.ibi.cjsamples.cjsamp parm1,parm2,...
   or
     SET SQLENGINE=CPJAVA
     SQL EX ibi.cjsamples.cjsamp parm1,parm2,... ;
     TABLE FILE SQLOUT
     PRINT field [ON TABLE PCHOLD]
     END
     SET SQLENGINE=OFF
   or
     SQL CPJAVA EX ibi.cjsamples.cjsamp parm1,parm2,... ;
     TABLE FILE SQLOUT
     PRINT field [ON TABLE PCHOLD]
     END
   CALLJAVA syntax uses no leading "java.", has a comma after the
   class and requires the END statement. Where EX syntax requires
   a SET EXORDER, a leading "java." and no comma after the class.
   Where SQL EX uses a SET SQLENGINE, no leading "java.", no comma
   after the class and requires a trailing semi-colon. Where SQL
   CPJAVA EX eliminates the SET SQLENGINE step. Which method
   to use is the users choice.

   The functions ibtrace.traceon and ibtrace.println have been deprecated
   (respectively) in flavor of log.isDebugEnabled() and log.debug, but
   the original functions are still active. This example uses the log.*
   functions.

   The indentation and formatting in this sample may seem odd,
   but is done to control wrapping and readability in the printed
   documentation.
*/

package ibi.cjsamples;
/* Required ibi classes */
import ibi.trace.*;
import ibi.callpgm.*;
import ibi.callpgm.ibianswr;
import ibi.trace.IBILogFactory;
import ibi.trace.ILogger;

/* Needed for time/date usage here, but generally typically needed. */
import java.sql.*;
import java.math.*;
import java.util.*;

public class cjsamp implements callpgm{
  private static final ILogger log = IBILogFactory.getLogger(cjsamp.class);
  private ibianswr answr = null;
  private int rownum = 0;
  private int numOfColumns = 0;
  private int Rows = 0;
  private String[] arrParms = null;
  /* Get class name for error messages */
  private String ClassName = getClass().toString();
  /* Array for no parms passed */
  private String[] NoParms = { "No ","Parameter ","Data ","Supplied!" };
  public cjsamp(){}
  /*
   Applications are required to have the next signature due to
   internal interface requirements for graph, but is not for
   customer use. Attempted customer use is not supported.
  */
```

```
public ibianswr
  execute(String username, String password, Object obj, String[] parms)
             throws Exception { return null; }
/* Actual application ... */
public ibianswr
     execute(String username, String password, String[] parms)
             throws Exception
{
/*
  General init info tracing. Tracing only displays only if server
  tracing on and displays in the stardard server traces jc######.trc.
  log.isDebugEnabled() is used to avoid the overhead of forming strings
  when tracing is not enabled.
*/

int numParms = (parms != null) ? parms.length : 0;
if (log.isDebugEnabled())
   {
   log.debug(" ... " + ClassName + ": Constructor ...");
   log.debug(" ... " + ClassName + ": Username: " +
             username + ", Password: " + password);
   log.debug(" ... " + ClassName +
             ": Number of parameter(s): " + numParms);
   for(int i=1; i <= numParms; i++)
      { log.debug(" ... -> Parameter " + i + ": " + parms[i-1]); }
   }

/* Check parm 1 for numeric and if too high of a repeat value. */
if(numParms == 0)
  { /* No parms defaults to 1 in echo mode */
    arrParms = NoParms;
    numParms = (arrParms != null) ? arrParms.length : 0;
    Rows = 1;
  }
  else
  {
    arrParms = parms;
    /* Example of exception handling and writing to trace */
    try { Rows = Integer.parseInt(arrParms[0],10);
        } catch (Exception e)
        {
        if (log.isDebugEnabled())
           {
           log.debug(" ... " + ClassName +
                     ": Parm 1 Non Integer: " + e + "\n");
           }
        Rows = 1; /* Reset to 1 if not numeric */
        }
    /* This is an example, reduce unrealistic repeats to 9. */
    if(Rows > 50) { Rows = 9 ; }
  }
```

```
/* Based on array contents, set field attributes for SQLOUT master */
/* If parm 2 is "datatype", run in datatype mode else echo mode. */
if(arrParms[1].equalsIgnoreCase("datatype"))
  { /* Create arbitrary number of columns for our datatypes. */
    numOfColumns = 11; /* Actually 10 plus 1 for MISSING example */
    if (log.isDebugEnabled())
       {
       log.debug(" ... " + ClassName +
                 ": Number of Columns: " + numOfColumns);
       }
    answr = new ibianswr(numOfColumns);
    for(int index = 0; index < numOfColumns; index++)
       {
       int colNum = index + 1 ; /* We start at 1 not 0 */
       if (colNum==1 || colNum==2)
       { /* Column 2 for MISSING example */
          answr.setColName(colNum,"FLD_"+colNum+"_IBI_ALPHA");
          answr.setColType(colNum,ibianswr.IBI_ALPHA);
          answr.setColSize(colNum,10);
       }

       else if (colNum == 3)
       {
          answr.setColName(colNum,"FLD_"+colNum+"_IBI_INTEGER");
          answr.setColType(colNum,ibianswr.IBI_INTEGER);
          answr.setColSize(colNum,4);
       }

       else if (colNum == 4)
       {
          answr.setColName(colNum,"FLD_"+colNum+"_IBI_FLOAT");
          answr.setColType(colNum,ibianswr.IBI_FLOAT);
          answr.setColSize(colNum,4);
       }

       else if (colNum == 5)
       {
          answr.setColName(colNum,"FLD_"+colNum+"_IBI_DOUBLE");
          answr.setColType(colNum,ibianswr.IBI_DOUBLE);
          answr.setColSize(colNum,8);
       }

       else if (colNum == 6)
       {
          answr.setColName(colNum,"FLD_"+colNum+"_IBI_TIME");
          answr.setColType(colNum,ibianswr.IBI_TIME);
          answr.setColSize(colNum,10);
       }
```

```
          else if (colNum == 7)
          {
             /*
             IBI_DATE must be java.sql.Date class format.
             Dates like DD-MMM-YYYY (ie OpenVMS format) are not,
             either convert to java.sql.Date or use IBI_ALPHA.
             */
             answr.setColName(colNum,"FLD_"+colNum+"_IBI_DATE");
             answr.setColType(colNum,ibianswr.IBI_DATE);
             answr.setColSize(colNum,10);
          }

          else if (colNum == 8)
          {
             answr.setColName(colNum,"FLD_"+colNum+"_IBI_TIMESTAMP");
             answr.setColType(colNum,ibianswr.IBI_TIMESTAMP);
             answr.setColSize(colNum,21);
          }

          else if (colNum == 9)
          {
             answr.setColName(colNum,"FLD_"+colNum+"_IBI_SMALLINT");
             answr.setColType(colNum,ibianswr.IBI_SMALLINT);
             answr.setColSize(colNum,2);
          }

          else if (colNum == 10)
          {
             answr.setColName(colNum,"FLD_"+colNum+"_IBI_BIGINT");
             answr.setColType(colNum,ibianswr.IBI_BIGINT);
             answr.setColSize(colNum,8);
          }

          else if (colNum == 11)
          {
             answr.setColName(colNum,"FLD_"+colNum+"_IBI_DECIMAL");
             answr.setColType(colNum,ibianswr.IBI_DECIMAL);
             answr.setColSize(colNum,18,7);
          }
          }
     }
```

```
    else
    { /* Create arbitrary number of columns for number of parms */
      answr = new ibianswr(arrParms.length);
      if (log.isDebugEnabled())
         {
         log.debug(" ... " + ClassName +
                   ": Number of columns: " + answr.getColsNumb()); }
         for(int column = 1; column <= answr.getColsNumb(); column++)
            {
            answr.setColSize(column,(column == 1) ?
                  Math.max(1,arrParms[column-1].length()) :
                             arrParms[column-1].length());
            answr.setColName(column, "Column" + column);
            }
    }
    return answr;
  }

  /* Based on array, return data for as many rows passed in parm 1 */
  public Integer fetch() throws Exception
  {
    if( ++rownum > Rows ) return IBI_EOD;
    /* If parm 2 is "datatype" run in test mode else echo mode. */
    if(arrParms[1].equalsIgnoreCase("datatype"))
      { /* For datatype test, set arbitrary data to send back */
      if (log.isDebugEnabled())
         {
          log.debug(" ... " + ClassName + ": fetching row: " + rownum);
         }
      for(int colNum = 1; colNum <= numOfColumns; colNum++)
      {
         if (colNum == 1)
          {  /* IBI_ALPHA */
          String value = new String("Col"+colNum+" Row"+rownum);
          answr.setColValue(colNum, value);
          }

          else if (colNum == 2)
          { /* IBI_ALPHA - Pass null for IBI_MISSING value behavior */
          String value = null;
          answr.setColValue(colNum, value);
          }

          else if (colNum == 3)
          { /* IBI_INTEGER */
          int vali = 2 ; answr.setColValue(colNum, vali);
          }

          else if (colNum == 4)
          { /* IBI_FLOAT */
          Float fl = new Float(7.0);
          float valf = fl.floatValue(); answr.setColValue(colNum, valf);
          }
```

```
else if (colNum == 5)
{ /* IBI_DOUBLE */
double vald = 6.0 ; answr.setColValue(colNum, vald);
}

else if (colNum == 6)
{ /* IBI_TIME */
java.sql.Time DV = new java.sql.Time(10000);
answr.setColValue(colNum, DV);
}

else if (colNum == 7)
{ /* IBI_DATE */
java.sql.Date DV = new java.sql.Date(100000);
answr.setColValue(colNum, DV);
}

else if (colNum == 8)
{ /* IBI_TIMESTAMP */
Timestamp ts = new Timestamp(100000);
answr.setColValue(colNum, ts);
answr.setColSize(colNum,ts.toString().length());
}

else if (colNum == 9)
{ /* IBI_SMALLINT */
short vals= 4 ; answr.setColValue(colNum, vals);
}

else if (colNum == 10)
{ /* IBI_BIGINT */
long vals= 6 ; answr.setColValue(colNum, vals);
}
```

```
            else if (colNum == 11)
            { /* IBI_DECIMAL */
            BigDecimal BD = new BigDecimal("12345678.90876");
            answr.setColValue(colNum, BD);
            }
            }
        }
        else /* For echo test, send back the same parms passed in */
        {
        /* If no parms passed, load array with no parms message */
        if(arrParms == null) { arrParms = NoParms ; }
        /* Return parms received for as many rows passed in parm 1 */
        for(int i = 1; i <= Rows ; i++)
           {
           if (log.isDebugEnabled())
              {
                 log.debug(" ... " + ClassName + ": fetching row: " + i);
              }
           for(int column = 1; column <= answr.getColsNumb(); column++)
              { answr.setColValue(column, arrParms[column-1]) ; };
           }
        }
        return IBI_DATA;
    }
  }
```

Building a JAVA Program and Starting the Server

The following are examples that can be used when building a JAVA program and starting a server. Note that the etc/java/srvr 7707/82x and higher jar location is used in the following examples and may need to be adjusted for older releases.

Example: Building a JAVA Application Manually on UNIX or IBM i

The following example assumes javac compiler is on $PATH.

One might typically build an application manually using code such as the following:

```
JARS=$HOME/ibi/srv77/home/etc/java/srvr
javac cjsamp.java -classpath $JARS/ntj2c.jar:$JARS/ibtrace.jar
mkdir ibi ibi/cjsamples 2> /dev/null
cp cjsamp.class ibi/cjsamples
jar cvf cjsamp.jar ibi/cjsamples/cjsamp.class
```

The result is a cjsamp.jar file in the current directory. In order to use your application, you must copy the resulting .jar file from the build directory to a production location, such as /usr/local/jars, and place it on the classpath before server start up.

Example: Building a JAVA Application Manually on Windows

The following example assumes javac compiler is on PATH.

One might typically build an application manually using code such as the following:

```
set JARS=C:\ibi\srv77\home\etc\java\srvr
javac cjsamp.java -classpath %JARS%\ntj2c.jar;%JARS%\ibtrace.jar
mkdir ibi ibi\cjsamples 2> nul:
copy cjsamp.class ibi\cjsamples
jar cvf cjsamp.jar ibi\cjsamples\cjsamp.class
```

The result is a cjsamp.jar file in the current directory. In order to use your application, you must copy the resulting .jar file from the build directory to a production location, such as C:\jars, and place it on the classpath before server start up.

Example: Building a JAVA Application Manually on OpenVMS

The following example assumes that the HP-supplied script for activating Java has been executed so the javac compiler can be found on the system path.

One might typically build an application manually using code such as the following:

```
$ DEFINE /USER JAVA$CLASSPATH-
  IADMIN:[IADMIN.IBI.SRV77.HOME.ETC.JAVA.SRVR]NTJ2C.JAR,-
  IADMIN:[IADMIN.IBI.SRV77.HOME.ETC.JAVA.SRVR]IBTRACE.JAR
$ JAVAC -verbose -d . cjsamp.java
$ JAR -cvf CJSAMP.jar [.ibi.cjsamples]cjsamp.class
```

The result is a cjsamp.jar file in the current directory. In order to use your application, you must copy the resulting .jar file from the build directory to a production location, such as IADMIN:[IADMIN.JARS], and place it on the classpath before server start up.

Example: Building a JAVA Application Using GENCPGM

This example assumes that the javac compiler is on $PATH (UNIX and IBM i), PATH (Windows) or system path (OpenVMS) and that GENCPGM has been copied to the current directory (otherwise, the full path must be used).

Use the following notations to build a class inside a jar (cjsamp in this example).

Platform	**Syntax**
Windows	`gencpgm –m cjava cjsamp.java`
UNIX	`gencpgm.sh –m cjava cjsamp.java`
IBM i (formerly known as i5/OS)	`gencpgm.sh –m cjava cjsamp.java`
OpenVMS	`@gencpgm –m cjava cjsamp.java`

For details about setting specific JAR names, see *Using the GENCPGM Build Tool* on page 89. GENCPGM supports building classes in a jar on all platforms.

Example: Starting a Server on UNIX or IBM i

In order for a server to recognize a new or updated CALLJAVA application, the CLASSPATH variable must be set with all required jars (including ntjzc.jar and ibtrace.jar) before you start the server. You must also restart the server if a .jar file is updated. For example, on UNIX you can start the server using the following code:

```
CLASSPATH=/usr/iadmin/ibi/srv77/home/etc/java/srvr/ntj2c.jar:
   /usr/iadmin/ibi/srv77/home/etc/java/srvr/ibtrace.jar:
   /usr/local/jars/cjsamp.jar
export CLASSPATH
edastart -start
```

Alternately, the properties page from the Web Console allows the configuration of CLASSPATH with user built jars as part of the servers configuration. To access the properties page from the Web Console, click *Workspace*, then *Configuration/Monitor*, and then *Java Services*.

For an example of how an application executes a Java class, see the various Execute methods at the start of the chapter.

Example: Starting a Server on Windows

In order for a server to recognize a new or updated CALLJAVA application, you must set the CLASSPATH variable with all required jars (including ntjzc.jar and ibtrace.jar) before you start the server. You must also restart the server if a .jar file is updated.

For example, on Windows you can start the server by selecting the following options:

My Computer > *Properties* > *Advanced Tab* > *Environment Variables*

Next, either add or edit the CLASSPATH variable and include ntjzc.jar, ibtrace.jar, and the new jar in the value. For example:

```
CLASSPATH C:\ibi\srv77\home\etc\java\srvr\ntj2c.jar;c:\ibi\srv77\home\etc
\java\srvr\ibtrace.jar;c:\jars\cjsamp.jar;
```

Then, start the server as a service or use any of the standard Windows start menu options of the server.

For an example of how an application executes a Java class, see the various Execute methods at the start of the chapter.

Example: Starting a Server on OpenVMS

In order for a server to recognize a new or updated CALLJAVA application, you must set the CLASSPATH or JAVA$CLASSPATH variable with all required jars (including ntjzc.jar and ibtrace.jar) before you start the server. You must also restart the server if a .jar file is updated.

If CLASSPATH is used for setting classpath, then UNIX file notation must be used, as described in the Java documentation for HP. Otherwise, JAVA$CLASSPATH may be used with OpenVMS file notation as illustrated next.

On OpenVMS you can start the server using the following code:

```
DEFINE JAVA$CLASSPATH JAVA$CLASSPATH-
 IADMIN:[IADMIN.IBI.SRV76.HOME.ETC.JAVA.SRVR]NTJ2C.JAR, -
 IADMIN:[IADMIN.IBI.SRV76.HOME.ETC.JAVA.SRVR]IBTRACE.JAR, -
 IADMIN:[IADMIN.JARS]CPSAMP.JAR
@EDASTART -START
```

For an example of how an application executes a Java class, see the various Execute methods at the start of the chapter.

Chapter 4

Writing a 3GL Compiled Stored Procedure Program

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.

In this chapter:

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 *Additional 3GL Reference Examples* on page 103 for languages supported and the samples in this chapter. If you are using an untested language and are having problems, contact Information Builders customer support so that a specialist can assist you.

For details on calling a compiled program with CALLPGM, see *Calling a Program as a Stored Procedure* on page 21.

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, IBM i, 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 IBM i, and shared library on OpenVMS). For more information, see *Additional 3GL Reference Examples* on page 103.

Setting Up the Control Block

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

- Under MVS, OpenVMS, UNIX, IBM i, or Windows, 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 their 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 improve 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 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.
			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.

Field	Length (in bytes)	Data Type	Description

`flag_value` (*continued*)	4	Integer	`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. Action_value of 14, 15, or 18 was specified, but the blocking factor was $\leq$ 0.
			`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.

Field	Length (in bytes)	Data Type	Description

`flag_value` (*continued*)	4	Integer	`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.

Field	Length (in bytes)	Data Type	Description

`action_value`
 	
Initial value on first call: 4
 	
(continued)
	4	Integer	`17`
			Program returning CREATE TABLE and character data.
			
			`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 IBM i pointers. Only required to exist for IBM i applications. Do not declare on other platforms.

`filler`	4	Any type	Pointer padding filler for IBM i pointers. Only required to exist for IBM i applications. Do not declare on other platforms.
`answer_area` Initial value on first call: 0	4 (32 bit) 8 (64 bit) 16 (IBM i)	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 70 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 IBM i pointers. Only required to exist for IBM i applications. Do not declare on other platforms.
`message_area` Initial value on first call: 0	4 (32 bit) 8 (64 bit) 16 (IBM i)	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 70 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 IBM i pointers. Only required to exist for IBM i applications. Do not declare on other platforms. Only existing and applicable when SQL SET CPGUB NEW, do not code when CPUG OLD is used.
`create_area` Initial value on first call: 0	4 (32 bit) 8 (64 bit) 16 (IBM i)	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 70 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 IBM i pointers. Only required to exist for IBM i applications. Do not declare on other platforms.

Field	Length (in bytes)	Data Type	Description
`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 a Dialogue Manager FOCEXEC procedure). This field is paired with parmlen. Twelve pairs are permitted per program call.

Setting Up a CALLPGM Control Block Structure for C

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

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 IBM i. The padding is a combination of being an IBM i behavior for pointer alignment and COBOL requiring pointer alignment to be explicitly coded on IBM i. Do not use these IBM i 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.
**** IBM i 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.
****  IBM i 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.
****  IBM i 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.
****  IBM i 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.
****  IBM i 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.
****  IBM i 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.
****  IBM i 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

Note: RPG is an IBM i 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 IBM i) requires the use of fillers for padding out pointer lengths. This padding is a combination of being an IBM i 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

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	The sample is stored in hlq.HOME.ETC (CPT) for PDS deployment. All other platforms are: *cpt.c* in the etc/src3gl directory of EDAHOME
COBOL	The samples are stored in hlq.HOME.ETC (SPGOLD) and hlq.HOME.ETC (SPGNEW) for PDS deployment. All other platforms are, respectively: ❏ SPGOLD.CBL in the etc/src3gl directory of EDAHOME ❏ SPGNEW.CBL in the etc/src3gl directory of EDAHOME
RPG	(IBM i only) ❏ SPGOLD.RPG in the etc/src3gl directory of EDAHOME ❏ SPGNEW.RPG in the etc/src3gl directory of EDAHOME

The portable COBOL examples have specifically been tested with IBM Enterprise COBOL V3R2, HP OpenVMS COBOLv2.7, and IBM i ILE COBOL. Depending on the target platform, minor editing (for example, commenting or un-commenting 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);
 }
                      .
                      .
                      .
```

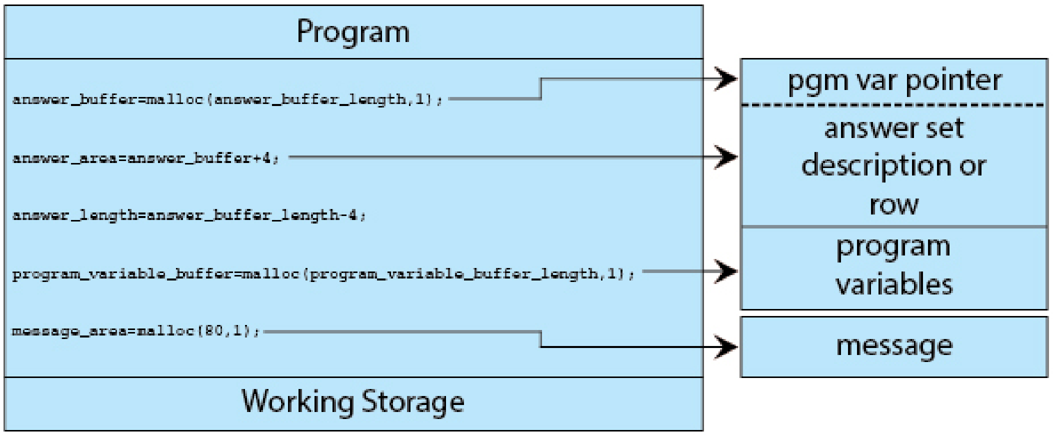
On the first call, the sample code allocates dynamic storage for:

❑ 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 image below.

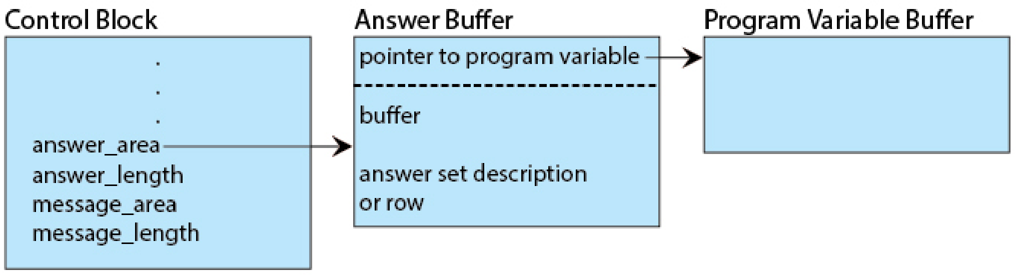


```
answer_buffer=malloc(answer_buffer_length,1);
answer_area=answer_buffer+4;
answer_length=answer_buffer_length-4;
program_variable_buffer=malloc(program_variable_buffer_length,1);
message_area=malloc(80,1);
```

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.

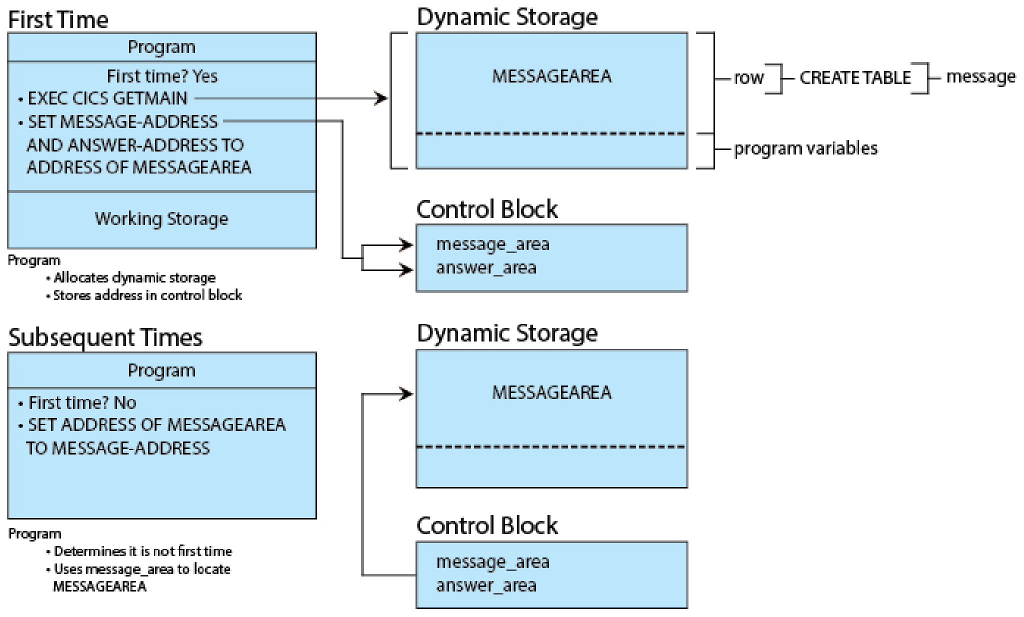
```
01  MESSAGEAREA.
    05  MESSAGE-OUT                PIC X(1000).
    05  NUM-ROWS                   PIC S9(8)  COMP-4.
    05  LAST-REC                   PIC S9(8)  COMP-4.
    05  CLOSE-PENDING-FLAG         PIC X.
        88  CLOSE-PENDING          VALUE "1".
        88  CLOSE-NOT-PENDING      VALUE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

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

To allocate dynamic storage, use the 'GETCOR' function, supplied on your distribution media in the HOME.DATA(GETCOR) library member.

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 in the HOME.DATA(GETCOR) library member.

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 HOME.DATA(CPGVTAM) on your distribution media), use the statements below. Use the HOME.DATA(GENCPGM) JCL as a reference, or your own JCL:

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

GETCOR is an assembler source program that provides functions '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 57.

```
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.

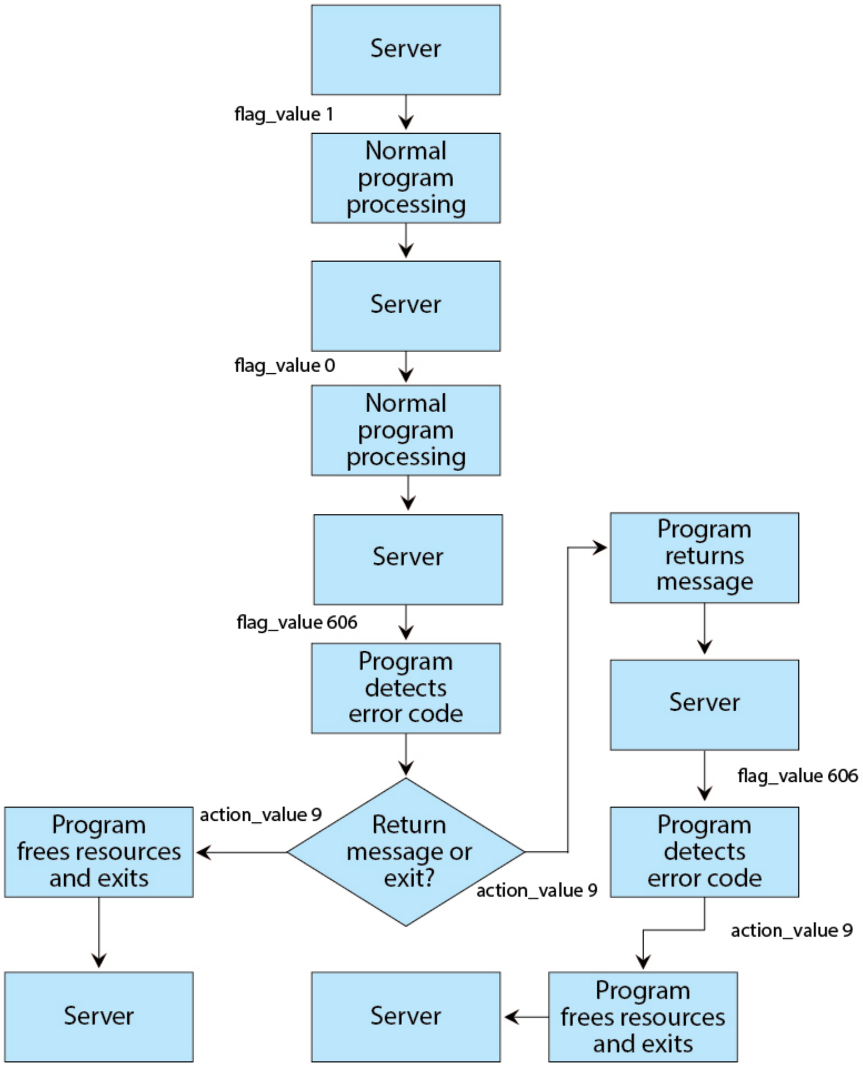
  Messages are retrieved by the client application before the processing of an answer set, or after the completion of answer set processing.

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:

Flag Value	Description
`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:

Data Type	Description
`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.

Data Type	Description
`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 User Written Routines

In addition to direct calls (that is, EX or CALLPGM) to compiled procedures, compiled procedures can be created for use in COMPUTEs, DEFINEs, and Dialogue Manager calculations. This allows one to take what might be a complex task and simply it into a reusable function. A classic example of this is a lookup function for strings or statistical model that requires many inputs and yields a single result.

The User Written Routines are also known as User Written Subroutines, UWSR functions, FUSE functions, and FUSELIB functions in some documentation. They are all the same feature.

In this chapter:

Calling a User Written Routine

Additional 3GL Reference Examples on page 103 contains reference examples primarily centered on writing and compiling a MTHNAME routine (a number-to-name lookup example) in a number of 3GL languages. Here are three simple ways to use the function:

❑ In Dialogue Manager

```
-SET &NAME = MTHNAME(1,'A13') ;
-TYPE The month is &NAME
```

❑ In TABLE

```
...
COMPUTE NAME/A13 = MTHNAME(1,'A13') ;
...
```

❑ In a DEFINE

```
...
NAME/A13 = MTHNAME(1,'A13') ;
...
```

The compiled and linked user written routine must reside in one of the following locations: the user directory under EDACONF, the location pointed to by the IBICPG environmental variable, or in an app directory. The z/OS and IBM i environments are limited to routine name lengths that are respectively 8 and 10 characters due to platform limitations. The limits for other platforms are generally much higher, but still must live within the limits of the operating system.

The GENCPGM build tool is generally used to build the routines. See *Additional 3GL Reference Examples* on page 103 for reference examples.

Chapter 6

Using the GENCPGM Build Tool

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 create and add a user written routine to the functions of the product (also known as a FUSELIB).
- To create and customize user exits that provide special functions.
- To create CALLPGM programs that the server executes.

In this chapter:

Using GENCPGM

GENCPGM is the general term for a series of platform specific scripts for compiling and linking 3GL programs (for example, C, COBOL, Fortran, Java, etc.) that interact with Information Builders products.

The scripts and their associated platforms are:

- gencpgm.sh (UNIX, Linux, z/OS and OS400)
- gencpgm.bat (Windows)
- gencpgm.com (OpenVMS)

The script for a given platform is located in the bin directory of the software installation directory (EDAHOME), except on a z/OS PDS deployment, where it is it in the member hlq.HOME.ETC(GENCPGM), and must be copied to and given execute privileges to be used under HFS.

Examples of the types of programs that can be built are:

- HLI applications to talk directly to FOCUS databases or servers using FDS access to FOCUS databases
- Call Java Adapter (CALLJAVA) applications which use Java classes to retrieve row(s) data
- Call Procedural Program Adapter (CALLPGM) applications which use a DDL to retrieve row(s) data.
- Subroutine applications which use a DLL to do specialized inline calculations for Dialogue Manager, DEFINEs or COMPUTEs.

From a technical perspective, the above list breaks down into 3 classes of 3GL programs that GENCPGM builds:

- Dynamic link libraries.
- Executables.
- Java applications as a class in a jar.

A dynamic link library is also known as a DLL and is generally thought of as a Windows specific term, however, there are equivalences on all other platforms. DLL libraries have an extension of .dll on Windows. On UNIX and USS, the term for DLL is shared library with an extension of .so except for some HPUX systems, which use .sl. On OS400, a DLL is a service program (programs marked SRVPGM). On OpenVMS, the term for DLL is also shared library, but the file extension is .exe.

The GENCPGM scripts are solely supplied as an assist tool for building basic applications. The GENCPGM scripts are not intended to support all languages and complex cases, like building several objects all linked into a final program. The GENCPGM scripts actually depend on an appropriate native compiler and linker being installed and accessible. *Native* refers to the compiler of the operating system vendor (for example, Microsoft on Windows, IBM on AIX, and so forth). *Accessible* means that it is known to the registry on Windows or is in the PATH for other operating systems, and that it employs the normal program names used by the originating vendors (for example, cl.exe on Windows, cc on many UNIX systems, gcc on Linux, javac (and jar) for Java, and so forth). The compiler also needs to generate binaries that match the bit requirements of the application software (32-bit servers require 32-bit compilers), although some compilers control this using a switch (for example, -m32 and -m64).

Because of the widespread use of GNU GCC (which is free), the Windows and UNIX versions of GENCPGM also recognize and allow *gcc* as a compiler specification, although they still depend on gcc being in the path or, in the case of Windows, having the variable MINGWROOT set (see the Windows section for more details). In short, GENPGM is a build assistant to access existing compiler tools, but is not itself a compiler/linker and, as such, the user is responsible for having the appropriate compilers and linkers installed and accessible if GENCPGM compilation is needed. Note that many instances are strictly for build-time use, and the resulting binaries may simply be deployed thereafter if the operating system and application bit requirements match (32-bit or 64-bit), and the deployment machines do not have compiler/linker requirements.

The use of GENCPGM as a build tool is not actually required for applications when proper build rules are followed, as implemented in GENCPGM and outlined in the build rules section. Since complex cases that use other languages or multiple sources are a legitimate requirement, it is left to the user to code and maintain their own build scripts for these cases and alternate languages (possibly using GENCPGM as a template and following the rules outlined in the build rules section).

It should also be noted that a subroutine is sometimes referred to by its former terminology of Fuselib or Fuselib Routine. From an application perspective (that is, a focexec) they are one in the same, however, FOCUS products used a single library to implement and store multiple routines where WebFOCUS uses individual libraries for each routine. This means older existing FOCUS libraries are not directly usable with WebFOCUS, but the underlying 3GL sources are usable and simply need to be built using the current methodologies documented here.

While there are a few platform and need specific switch options, most switches and many languages work on most platforms. Concerning specific 3GL languages, C is the officially supported language on all platforms, other languages vary by platform as noted in the samples. Theoretically, any 3GL language that is capable of being compiled and linked into a DLL or executable and is capable of being used with Information Builders products, however, GENCPGM is only coded for certain commonly used languages that we have easy access to and expertise in creating scripts and working samples. Requests for additional languages will be considered on a case by case basis.

GENCPGM is also used dynamically in the server product for the COMPILED DEFINE feature and as such the version in the EDAHOME directory should never be customized to prevent changes from affecting the COMPILED DEFINE feature. If you have customizations that you feel would be useful to others, they may be submitted via Customer Support for consideration as a permanent change.

Reference: USAGE Chart (Typical Syntax Plus Extended Options)

UNIX, Linux, IBM i, USS:

```
{path}gencpgm.sh [-h] [-x] [-q] [-v] [-e] [-n] [-s script]
 [-H EDAHOMELIB] [-p LOADLIB] [-d directory|-w directory] [-g]
 [-c language] [-m application type]  [-b lib/srvprg] [-j jarname]
 {path}{program name}[.{extension}]
```

Windows:

```
{path}gencpgm.bat [-h] [-x] [-q] [-v] [-e] [-n] [-s script]
 [-d directory|-w directory] [-g]  [-c language]
 [-m application type] [-j jarname]
 {path}{program name}[.{extension}]
```

OpenVMS:

```
@{path}gencpgm.com [-h] [-x] [-q] [-d directory|-w directory] [-g]
 [-c language] [-m application type] [-j jarname]
 {path}{program name}[.{extension}]
```

where:

Switch/Option	Description
-h	Outputs this Help text.
-x	Turns on set -x shell tracing to assist in debugging.
-q	Turns on quiet mode to redirect Microsoft compiler/linker output to nul: on Windows. The switch does nothing on other platforms because the compliers and linkers on most other platforms are already "quiet".
-v	Turn on compiler/linker verbose options plus selective informational messages.
-e	Extended trace/compiler/linker info from just before compiler/linker step.
-H EDAHOMELIB	Server for z/OS in a PDS deployment only. Indicates installation home {HLQ}.HOME of ETC.H for picking up standard IBI C include files (needed for some samples).
-C EDACONFHLQ	Server for z/OS in a PDS deployment only. Indicates installation configuration {HLQ} of ETC for picking up standard IBI files (that is, server and communications configuration files).
-A APPROOTHLQ	Server for z/OS in a PDS deployment. Indicates installation configuration {HLQ} of APPS (APPROOT) for picking up application files for HLI applications.

Switch/Option	Description
-S SCRIPTSPDS	Server for z/OS in a PDS deployment only. Indicates PDS to copy application build JCL and run time execution scripts for batch JCL and interactive CLIST and REXX of the application.
-s *SCRIPT*	For z/OS PDS Unified Environment deployment. Indicates generate only (no compile/link) to a fully qualified PDS or dataset name.
-p LOADLIB	Server for z/OS in a PDS deployment only. Indicates JCL type compilation and points to the load lib to use. The load lib must be in run time STEPLIB.
-d *directory*	Work in the given directory. The C file should be in this directory. All resulting files will be generated in this directory. This is for COMPILED DEFINE purposes and not intended for customer use.
-w *directory*	Write final executable (and any helper scripts) in the given target directory.
-i *directory*	Include directory. Multiple uses allowed.
-n	No runner shell creation for api*, hli and odbc programs. Use to prevent overwrite of an existing shell that may have been customized.
-g	Generate a debuggable program by including debug switch in the compilation and link.

The -c option is described in the following chart:

Language	Compiler to use for a given language source.
cc	Use standard C compiler to compile "progname.c". C is the default compiler language.
assembler	Use assembler compiler. Only implemented for z/OS currently. HFS usage requires source to have a .s file extension.
fortran for f	Use default Fortran to compile Fortran with a .fortran extension. Use default Fortran to compile Fortran with a .for extension. Use default Fortran to compile Fortran with a .f extension. Supply explicit extension to override extension. If default compiler is not available, GNU g77 will be checked for availability and used.

Language	Compiler to use for a given language source.
f77 f90 f95 old_f77	Use a specific Fortran compiler to compile. Fortran implementation on UNIX is limited to Sun SUNWspro f95 and GNU (g77/f77) as most UNIX OS vendors do not supply Fortran compilers. OpenVMS supports F90 (default), F77 (Compaq Specification) and OLD_F77 (DEC Specification).
g77	Use the GNU Fortran (g77/f77) compiler to compile "progname.f". GNU is only selectively supported as we do not have it installed on all platforms, but should work because GNU is GNU.
gcc	Use the GNU C (gcc) compiler to compile "progname.c". GNU is only selectively supported as we do not have it installed on all platforms, but should work because GNU is GNU.
CC CXX cpp	Use the "C++" compiler to compile "progname.cpp" programs. C++ programs are expected to have a .cpp suffix on all platforms except on MVS OE, which requires .C as an explicit extension.
rpg	IBM i Only: Use RPG compiler to compile IFS "progname". Default extension is .rpg. Source may alternately exist as member in *CURLIB/QRPGLESRC.
pl1	z/OS Only: PL/1
basic bas	OpenVMS Only: Basic
pascal pas	OpenVMS Only: Pascal
cobol cob cbl	Use COBOL compiler to compile *progname*.cobol, *progname*.cob, and *progname*.cbl. Supply explicit extension to override.
java	Dummy placeholder, -m cjava is the driving factor for Java source compilation.

The -m option is described in the following chart:

Application Type	Type of Application to Build
hli	Generate an HLI program linked to the EDA HLI library that opens and modifies FOCUS data files.
odbc	Generate an ODBC API client program linked to the ODBC API driver w/o Visigenics Driver Manager (deprecated).
cpgm dll	Generate a "callpgm" server program library or sub routine (also known as a Fuselib routine). Default is a C source with .c extension unless specified by explicit (known) extension or specific -c compiler flag.
cjava	Generate a class in a jar for "calljava" server program usage. Multiple .java sources are allowed under this feature. Default jar file name is the same as the first named java source (use -j to create specific jar file names). Not supported on OpenVMS.
cl	IBM i only: Compile CL command file. Default extension of .cl; LIB/FILE(MBR) type of file specification allowed if quote enclosed to prevent sub shell interpretation by the command line parser.
cmd	IBM i Only: Compile CMD command file. Default extension of .cmd; LIB/FILE(MBR) type of file specification allowed if quote enclosed to prevent sub shell interpretation by the command line parser.
dds	IBM i Only: Compile DDS screen file. Default extension is .dds. Source may alternately exist as member in *CURLIB/QDDSSRC. DDS is an IBM i only extension for compiling screen handling files for RPG and other IBM i languages that use DDS.
-b lib/srvprg	IBM i Only: Bind in additional IBM i service programs during the link phase.
-ansi	OpenVMS Only: Indicates /ANSI switch should be added to compiler string. For COBOL this allows line numbers. For C this uses ANSI C aliasing rules.
-ieee	OpenVMS Only: Indicates /FLOAT=IEEE should be added to compile string. The G_FLOAT version of the product automatically does /FLOAT=G_FLOAT, this switch allows IEEE to be forced by the G_FLOAT version.

Application Type	Type of Application to Build
-gfloat	OpenVMS Only: Indicates /FLOAT=G_FLOAT should be added to compile string. The IEEE version of the product automatically does /FLOAT=IEEE, this switch allows G_FLOAT to be forced by the IEEE version.
-j *jarname*	Used solely in conjunction with -m cjava to specify a specific jar to create. A .jar extension may be supplied, but extensions other than .jar are ignored and automatically switched to .jar. If the switch is not included, the first java source will be used to form the jar name (that is, myapp.java yields myapp.jar).
[{*path*}]{*program name*}[[.*extension*}]	Name of source program to build. Must be last argument. All arguments after *program name* are ignored. Extension is optional, but can serve to override a language default for a -c language specification. A path to a source is allowed (that is, source/foo.c), but non-system includes must be in current directory or the -i option must be used. On Server for z/OS in a PDS deployment, a dataset name or PDS name may be specified if -p option is in use, however, the use of parenthesis characters in the specification also requires the name to be quoted to prevent sub shell interpretation by the command line parser.

Procedure: How to Compile and Link a Procedure

This section outlines the steps required to compile and link a sample procedure provided with the product:

1. Copy GENCPGM from the EDAHOME bin directory to your working directory, or use the full path name to the location, and:

   ❑ 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/src3gl directory of EDAHOME to your working directory.

   ❑ For user exits, copy the desired sample exit from the etc/src3gl directory of EDAHOME to your working directory.

- ❏ For user routines, write the routine or copy and modify an existing routine to your working directory. (This document provides MTHNAME samples for C, COBOL, RPG, and Fortran, which you can use for reference.)

2. Issue an EDAHOME environment variable pointing to the EDAHOME directory. For example:

Windows	SET EDAHOME=C:\ibi\srv77\home
IBM i (formerly known as i5/OS)	export EDAHOME=/home/iadmin/ibi/srv77/home
UNIX	export EDAHOME=/home/iadmin/ibi/srv77/home
USS	export EDAHOME=/home/iadmin/ibi/srv77/home
OpenVMS	DEFINE EDAHOME IADMIN:[IADMIN.IBI.SRV77.HOME]

3. If building an API program, also issue an EDACONF environment variable pointing to the EDACONF directory. For example:

Windows	SET EDACONF=C:\ibi\srv77\ffs
IBM i	export EDACONF=/home/iadmin/ibi/srv77/ffs
UNIX	export EDACONF=/home/iadmin/ibi/srv77/ffs
USS	export EDACONF=/home/iadmin/ibi/srv77/ffs
OpenVMS	DEFINE EDACONF IADMIN:[IADMIN.IBI.SRV77.FFS]

4. Run GENCPGM. For example, on UNIX:

```
gencpgm.sh -m cpgm mysub.c
```

Reference: GENCPGM Usage Notes

While there may not be a sample in every language for every application type, the first step is to confirm that there is a working environment by building one of the standard samples for the desired application type and confirming that it runs. If the samples do not work, there is little hope that a custom program will work.

Switches function similarly on all implementations, although, some are platform/need specific.

Programs generated for HLI will also have command file shell wrappers created with the system variables used for run time execution (that is, EDAHOME, EDACONF, EDACS3 and the library path needed for HLI to load) producing a self contained environment for runtime execution. At runtime, all application setup needs are self-contained within the wrapper so that the application simply runs. The explicit use of a GENCPGM-generated application wrapper is not required if the settings within a given wrapper are issued elsewhere (such as in a system or login profile) and the executable is directly used.

On IBM i CL and CMD, wrappers are also created in *CURLIB so applications can also be called directly on the IBM i command line. On z/OS PDS deployment, JCL, CLIST, and REXX wrappers are created as appnameJ, appnameC and appnameR (respectively), if the application is 7 characters (or less) and the -S switch has been used to indicate a save PDS name. Since the running interactive or batch and selecting a preferred language are strictly run time choices, PDS mode creates all three scripts to be prepared for all situations.

Due to the numerous third party vendors of COBOL, Fortran and other languages, inconsistency of switches between third party vendors and across platforms, GENCPGM has only limited testing of third party compilers. The actual supplied COBOL and Fortran sample programs themselves are known to work on several platforms where we do have compilers so if GENCPGM for you platform doesn't support a particular language the sole question is of figuring out how to compile and link them in order to work. Please also note that some samples (particularly COBOL) have comments of specific platform related changes that must be made for to accomplish proper compilation such as changing the PROGRAM-ID to a quoted lowercase string to achieve a properly created program entry point.

For CALLJAVA applications (-m cjava) more than one source to compile is allowed and the resulting classes are created into a single jar is supported. Java sources must have file extension of .java and specifying the actual extension on the GENCPGM command line is optional. If there are multiple source and no -j jar switch is supplied, the first source will be used to form the jar name.

The language parameter value for -c drives the default extension for a given language (except for Java), but supplying a full program name (ie mthname.cbl) will override a default.

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.

Reference: Language and Platform Notes

Theoretically any compiled 3GL language can be used to create an HLI, Call Procedural Program, or Subroutine programs. C is generally considered the standard language and is universally tested and implemented on all supported platforms with samples for all application types. Other languages are more selective in terms of applications for which samples exist and platforms in which they can be tested (usually due to complier availability on a given platform).

Java and JavaScript do not have options for generating Dynamic Load Libraries (DLLs), and, as such, cannot be used for creating HLI, Call Procedural Program, or Subroutine programs. However, in a language like C, it is possible to create a wrapper that loads and passes parameters to Java and receives parameters back. Thus, while Java is feasible, it is not direct and would present performance issues if done in this context and thus can not be recommended or officially supported.

Fortran: It is possible to build DLLs and programs using Fortran on any platform, however, at this time GENCPGM is only coded for Fortran on z/OS, OpenVMS, UNIX GNU g77 and SunOS SUNWspro. Additionally, there is only a sample for SubrRoutine usage. The -c fortran switch on SunOS defaults to f77 usage on AMD64 and f90 on Sparc9, use the -c f77/f90/f95 switch to force other specific levels. The -c fortran switch on UNIX's will attempt to use g77, if found on the path.

COBOL: It is possible to build routines using Cobol on any platform. At this time gencpgm is coded to do Cobol only on select platforms and using select Cobol vendors, specifically ... on UNIX with MicroFOCUS Cobol (using the mf* switches), OpenVMS using HP's Cobol, on IBM i using IBM ILE Cobol and IBM z/OS using Enterprise Cobol in -p mode. MicroFOCUS Cobol use has some additional use restrictions as described in the -c mfcobol section of the gencpgm chart.

On IBM i: Only ILE compilers are supported. Only the IBM i C compiler can directly compile files on the IFS file system. GENCPGM on IBM i does this feat for other languages by checking the default library location (for a given source type) for the existence of the desired file and if it does not exist it does a CPYFRMSTMF to duplicate the file into the library for the compilation process. If GENCPGM does a source file copy to a library, it will also remove the file afterwards so extra copies aren't floating around. In this way, sources on IBM i can exist as either IFS or library files.

On z/OS PDS Deployment: The script in hlq.HOME.ETC(GENCPGM) is an OMVS shell script and is not JCL, so it cannot be directly run from the PDS. To use under PDS deployment, copy the GENCPGM member to HFS, do a chmod +x to the script, and use as described below with z/OS switches.

Once the script is copied to an HFS directory, it is executed with the -L, -C, -A, -,S and -p options, which creates and then submits a JCL compilation stack that is language- and application-specific. If the JCL is successful, the resulting program will end up in the specified -p PDS. Regardless of build success, the JCL stack is always left behind and is saved in the current directory as the program name with a .jcl extension, as well as in the -S location if a -S switch was used. Additionally the -s switch will allow you to directly generate JCL into an HFS file or DSN, but not execute. The -s switch (lowercase s is required) is useful for sites where standard IBM libraries locations are not used for compiler and link library updates (as is commonly done at sites for add on features and updates), thus allowing a site to generate and then "adjust" the JCL for site specific needs before submission. The -S switch (uppercase S is required) does actual compilation plus saves build and run time scripts into the specified PDS for later use.

On OpenVMS: Special Oracle and Rdb relinking options are supported (-relink oracle, -relink rdb and -relink rdball (rdball is for when Multi Release Rdb is in use) for when library identity mismatches occur and require on-site linking. Do not use these options unless an explicit problem has been identified with customer support and you are requested to do so.

Reference: Build Rules

Should you chose to write a build script instead of using GENCPGM, the rules are fairly simple.

DLLs for Subroutine and CALLPGM Usage: Library name (less extension and any prefix such as lib) and entry point name must match. Some compilers are case sensitive on entry point name usage and some are not or uppercase entry points automatically; thus some require special coding to force lower case names as in mentioned COBOL cases. Specifically entry points must be lowercase.

Executables for HLI Usage: Must link in edahli DLL and create an executable with a main. Various environment variables must be set in order for application to run, the wrapper created by building the appropriate test sample should be used as a template as it contains any general and platform specific coding.

In both cases it is suggested that you use the standard test samples for your language of choice with the -x switch to examine the precise build switches used in any particular environment to assist in any custom built scripts.

Example: Generating a Subroutine Program 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.

Optionally, the explicit API switch could have been used:

```
gencpgm -g -m cpgm myprog
```

Example: Generating an HLI Program From a C Source File

Because the Standard C compiler and HLI mode are default options, the following example will generate a debuggable HLI program from a sample C source file named myprog.c using the standard C compiler.

Optionally, the explicit HLI switch could have been used:

```
gencpgm -g -m hli myprog
```

Example: Generating a CALLPGM Program 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 *Writing a 3GL Compiled Stored Procedure Program* on page 55.

Chapter 7

Additional 3GL Reference Examples

The Java and Stored Procedure chapters directly contain reference examples that annotate the interaction of these types of procedures. The examples in this appendix are in reference to user written routines (*User Written Routines* on page 87) and are primarily the same example written in a number of 3GL languages. This is done to show the flexibility of the user written routine feature in languages with which a developer might be more familiar.

In this chapter:

Subroutine Source Examples and Runtime Testing

This section illustrates select sample subroutines. All 3GL reference examples for subroutines (as well as Exit, RPC and API examples) are delivered within the etc/src3gl sub directory and z/OS PDS locations and names as noted in the reference samples below (and in other examples in this manual). One reference example is actual several different language implementations (C, C++, Fortran, Cobol, BAL, Basic, RPG, PL/1 and Pascal) of a fairly simple task, translate a number into a spelled out month name (mthname). The different language implementations allow one to focus on the implementation issues in a language they may be more familiar with. The other example is a string reversing example that accounts for how to handle Unicode UTF-8 (UREVERSE) which is strictly a C example. Each has been tested and works for its given target environment.

Note that some of the samples have comments within them about portions that need to be adjusted to account for known language implementation differences on some platforms. For example, IBM i COBOL requires a change in the PROGRAM-ID specification to force a lower case entry point name and OpenVMS doesn't support GOBACK. As stated earlier, in theory any compiled and linked languages that can create a DLL can be used to create subroutines. Once a program is built as a DLL, the loading and execution process is generally agnostic of the original language.

Please note that while Microsoft Visual Basic (VB) and C# are popular Windows languages, they do not have options for generating true WIN32 Dynamic Link Libraries (DLLs with .dll extensions) and, as such, cannot be used for building subroutines because the loader process requires that only standard DLL objects be used. This is considered a Microsoft issue. Also note that an internet search for "build dll in vb or C# " yields a number of sites that describe how to force VB and C# to create DLLs. While such techniques seem promising for customers who want to use these languages, and may very well execute properly, Information Builders cannot officially support unsupported techniques. However, we will work with customers to resolve problems within this scope.

Some language samples (Pascal for instance) may not be capable of being built by GENCPGM for a given platform (ie UNIX and Linux), but is still provided on the media for all platforms for reference purposes and for people that decide to create their own build scripts.

The disk locations below, use PDS notation for PDS Deployment and UNIX notation for "all other platforms" for the purpose of being brief. The locations for Windows would be the same except the slashes are back slashes. The locations for OpenVMS would be dots instead of slashes and the directory portion would be enclosed in square braces.

Any of the MTHNAME sample routines can be tested by creating a simple FOCEXEC and using the following sample steps:

Create FOCEXEC **mthname.fex**

```
-SET &MTHNAME = MTHNAME(&MTHNUMBER,'A12') ;
-TYPE Month &MTHNUMBER is &MTHNAME
```

Compile and set IBICPG (this is using the C example on UNIX):

```
export EDAHOME=/home/iadmin/ibi/srv76/home
gencpgm.sh -m cpgm mthname.c
export IBICPG=`pwd`
```

After restarting the server, execute an RPC like:

```
EX MTHNAME MTHNUMBER=4
```

And receive:

Month 4 is March

MTHNAME C Implementation

Note:

- This sample is stored in hlq.HOME.ETC(MTHNAMC) for PDS deployment and as home/etc/src3gl/mthname.c on all other platforms.
- On PDS deployment use "-o mthname" to build using alternate source name of MTHNAMC.

Source:

```
/*                                                      */
/* MTHNAME: Sample User Written Routine in C            */
/*                                                      */
/* iWay/EDA refers to these as User Written Routines    */
/* and WebFOCUS/FOCUS refers to them as FUSELIBs        */
/* Routines. They are written in the same way for all   */
/* platforms and products, but the compilation and      */
/* link steps may differ depending on release and       */
/* product level. See appropriate platform/product      */
/* documentation for compilation and link instructions. */
/*                                                      */

void
mthname(double *mth, char *month)
{
static char *nmonth[13] = {"** Error **",
                           "January    ",
                           "February   ",
                           "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];
return;
}
```

MTHNAME C++ Implementation

Note:

- This sample is stored in hlq.HOME.ETC(MTHNAMCP) for PDS deployment and as home/etc/src3gl/mthname.cpp on all other platforms.
- On PDS deployment use "-o mthname" to build using alternate source name of MTHNAMCP.

Source:

```
// MTHNAME: Sample User Written Routine in C++
// Warning: Use on MVS OE requires extension to be renamed as .C
extern "C" int mthname(double* mth, char* month)
{
const char *nmonth[13] = {"** Error **",
                           "January   ",
                           "February  ",
                           "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];
return 0;
}
```

MTHNAME Fortran Implementation

Note:

- This sample is stored in hlq.HOME.ETC(MTHNAMF) for PDS deployment and as home/etc/src3gl/mthname.f on all other platforms.
- On PDS deployment use "-o mthname" to build using the alternate source name of MTHNAMF.
- This sample is based on the original mainframe Fortran sample with a name of MTHNAM. This has been changed to have a uniquely sourced version that more closely matches the C version and has comments. The samples are otherwise the same.

Source:

```
       SUBROUTINE MTHNAME (MTH,MONTH)
       REAL*8     MTH
       INTEGER*4  MONTH(3),A(13,3),IMTH
       DATA
      +     A( 1,1)/'Janu'/, A( 1,2)/'ary '/, A( 1,3)/'    '/,
      +     A( 2,1)/'Febr'/, A( 2,2)/'uary'/, A( 2,3)/'    '/,
      +     A( 3,1)/'Marc'/, A( 3,2)/'h   '/, A( 3,3)/'    '/,
      +     A( 4,1)/'Apri'/, A( 4,2)/'l   '/, A( 4,3)/'    '/,
      +     A( 5,1)/'May '/, A( 5,2)/'    '/, A( 5,3)/'    '/,
      +     A( 6,1)/'June'/, A( 6,2)/'    '/, A( 6,3)/'    '/,
      +     A( 7,1)/'July'/, A( 7,2)/'    '/, A( 7,3)/'    '/,
      +     A( 8,1)/'Augu'/, A( 8,2)/'st  '/, A( 8,3)/'    '/,
      +     A( 9,1)/'Sept'/, A( 9,2)/'embe'/, A( 9,3)/'r   '/,
      +     A(10,1)/'Octo'/, A(10,2)/'ber '/, A(10,3)/'    '/,
      +     A(11,1)/'Nove'/, A(11,2)/'mber'/, A(11,3)/'    '/,
      +     A(12,1)/'Dece'/, A(12,2)/'mber'/, A(12,3)/'    '/,
      +     A(13,1)/'**ER'/, A(13,2)/'ROR*'/, A(13,3)/'*   '/
       IMTH=MTH+0.000001
       IF (IMTH .LT. 1 .OR. IMTH .GT. 12) IMTH=13
       DO 1 I=1,3
     1 MONTH(I)=A(IMTH,I)
       RETURN
       END
```

Note: Some Fortran compilers support character variables longer than 4 bytes and, in this case, the example's array could be constructed as a CHARACTER*10 with `A(1)/'January'/, ..., A(13)/'***ERROR**'/` syntax, but the split array syntax used in example above is known to work on all Fortran compilers.

MTHNAME COBOL Implementation

Note:

- ❑ This sample is stored in hlq.HOME.ETC(MTHNAMCB) for PDS deployment and as home/etc/src3gl/mthname.cbl on all other platforms.
- ❑ GENCPGM for COBOL is supported on OpenVMS, z/OS and i/OS (not UNIX/Linux) and the sample on disk is only provided for reference for all other platforms.
- ❑ On PDS deployment use "-o mthname" to build using alternate source name of MTHNAMCB.
- ❑ COBOL generally requires small changes depending on platform as outlined in the notes with the sample.
- ❑ Some compilers ignore line numbers and some have compiler switches to ignore. GENCPGM on OpenVMS can use the -ansi switch to ignore the numbers or (alternately) the COBOL command can be redefined to be COBOL /ANSI in the environment.

Source:

```
000100*
000200 IDENTIFICATION DIVISION.
000300*
000400* MTHNAME: Sample User Written Routine in Cobol
000500*
000600* Notes:
000700*
000800*  1. This sample is based on the original mainframe
000900*     sample with a PROGRAM-ID of MTHNAM. This has been
001000*     changed to have a uniquely sourced version that
001100*     more closely matches the C version and has these
001200*     comments. The samples are otherwise the same.
001300*
001400*  2. Original mainframe sample had a GOBACK as the
001500*     last statement. OpenVMS Cobol seems to object
001600*     to this, so commented it out as noted below.
001700*     Unix compiler support for GOBACK may also vary
001800*     by vendor and untested at this time (5/1/2003).
001900*
002000*  3. OpenVMS compiled and was found, but initial
002100*     always returned the error case. This was
002200*     actually a GENCPGM.COM error that the Cobol
002300*     needed the /FLOAT=G_FLOAT switch, so be sure
002400*     that you are using a GENCPGM.COM from 5.2.3
002500*     or higher where this is fixed.
002600*


002700*  4. The PROGRAM-ID name may also needed some
002800*     special handling depending on the platform.
002900*     The reason for this is that iWay routines
003000*     are searched for in lower case and there
003100*     seems to be some case sensitivity problems
003200*     for the platforms tested so far. OpenVMS
003300*     doesn't seem to care if name is lower or
003400*     upper case.  i5/OS Cobol is not only case
003500*     sensitive but requires explicit lower case
003600*     values to be in single quotes, but also
003700*     needs the compiler option *NOMONOPRC to
003800*     respect the coded value. So, depending
003900*     on your platform, the PROGRAM-ID value may
004000*     need editing as per notes below.
004100*
004800*
004900* ID Usage for Mainframe and OpenVMS ...
005000*PROGRAM-ID. MTHNAME.
005100* ID Usage for Unix and Windows ...
005200*PROGRAM-ID. mthname.
005300* ID Usage for i5/OS ...
005400*PROGRAM-ID. 'mthname'.
005500*
005600* ID Usage for this run ...
005700 PROGRAM-ID. mthname.
005800*
```

```
005900 ENVIRONMENT DIVISION.
006000 CONFIGURATION SECTION.
006100 DATA DIVISION.
006200 WORKING-STORAGE SECTION.
006300    01 MONTH-TABLE.
006400      05 FILLER PIC X(9) VALUE 'January  '.
006500      05 FILLER PIC X(9) VALUE 'February '.
006600      05 FILLER PIC X(9) VALUE 'March    '.
006700      05 FILLER PIC X(9) VALUE 'April    '.
006800      05 FILLER PIC X(9) VALUE 'May      '.
006900      05 FILLER PIC X(9) VALUE 'June     '.
007000      05 FILLER PIC X(9) VALUE 'July     '.
007100      05 FILLER PIC X(9) VALUE 'August   '.
007200      05 FILLER PIC X(9) VALUE 'September'.
007300      05 FILLER PIC X(9) VALUE 'October  '.
007400      05 FILLER PIC X(9) VALUE 'November '.
007500      05 FILLER PIC X(9) VALUE 'December '.
007600      05 FILLER PIC X(9) VALUE '**ERROR**'.
007700    01  MLIST REDEFINES MONTH-TABLE.
007800      05  MLINE OCCURS 13 TIMES INDEXED BY IX.
007900          10 A  PIC X(9).
008000    01  IMTH    PIC S9(5) COMP.
008100 LINKAGE SECTION.
008200    01  MTH     COMP-2.
008300    01  MONTH   PIC X(9).
008400 PROCEDURE DIVISION USING MTH, MONTH.
008500 BEG-1.
008600       ADD 0.000001 TO MTH.
008700       MOVE MTH TO IMTH.
008800       IF IMTH < +1 OR > 12
008900         SET IX TO +13
009000       ELSE
009100         SET IX TO IMTH.
009200       MOVE A (IX) TO MONTH.
009300*
009400* On OpenVMS ... Comment out the GOBACK.
009500*
009600       GOBACK.
```

MTHNAME z/OS BAL Assembler Implementation

Note:

- This sample is stored in hlq.HOME.ETC(MTHNAMAS) for PDS deployment and as home/etc/src3gl/mthname.x on all other platforms.
- GENCPGM for BAL is only supported on z/OS and the sample on disk is only provided for reference for all other platforms.
- On PDS deployment use "-o mthname" to build using alternate source name of MTHNAMAS.

Source:

```
*
* MTHNAME: Sample User Written Routine in z/OS BAL Assembler
*
* If this is used as a source read directly from an HFS file
* system the extension must be .x for assembler files.
*
MTHNAME  CSECT
MTHNAME  AMODE 31
MTHNAME  RMODE ANY
         STM   14,12,12(13)        save registers
         BALR  12,0                load base reg
         USING *,12
*
         L     3,0(0,1)            load addr of first arg into R3
         LD    4,=D'0.0'           clear out FPR4 and FPR5
         LE    6,0(0,3)            FP number in FPR6
         LPER  4,6                 abs value in FPR4
         AW    4,=D'0.00001'       add rounding constant
         AW    4,DZERO             shift out fraction
         STD   4,FPNUM             move to memory
         L     2,FPNUM+4           integer part in R2
         TM    0(3),B'10000000'    check sign of original no
         BNO   POS                 branch if positive
         LCR   2,2                 complement if negative
*
POS      LR    3,2                 copy month number into R3
         C     2,=F'0'             is it zero or less?
         BNP   INVALID             yes. so invalid
         C     2,=F'12'            is it greater than 12?
         BNP   VALID               no. so valid
INVALID  LA    3,13(0,0)           set R3 to point to item @13 (error)
*
```

```
VALID    SR    2,2                 clear out R2
         M     2,=F'9'             multiply by shift in table
*
         LA    6,MTH(3)            get addr of item in R6
         L     4,4(0,1)            get addr of second arg in R4
         MVC   0(9,4),0(6)         move in text
*
         LM    14,12,12(13)        recover regs
         BR    14 return
*
         DS    0D                  alignment
FPNUM    DS    D                   floating point number
DZERO    DC    X'4E00000000000000' shift constant
MTH      DC    CL9'dummyitem'      month table
         DC    CL9'January'
         DC    CL9'February'
         DC    CL9'March'
         DC    CL9'April'
         DC    CL9'May'
         DC    CL9'June'
         DC    CL9'July'
         DC    CL9'August'
         DC    CL9'September'
         DC    CL9'October'
         DC    CL9'November'
         DC    CL9'December'
         DC    CL9'**ERROR**'
         END   MTHNAME
```

MTHNAME Basic Implementation (Based on HP OpenVMS Basic 1.4)

Note:

- This sample is stored in hlq.HOME.ETC(MTHNAMBS) for PDS deployment and as home/etc/src3gl/mthname.bas on all other platforms.
- GENCPGM for Basic is only supported on OpenVMS and the sample on disk is only provided for reference for all other platforms.

Source:

```
1000 SUB mthname BY REF(REAL MTH, STRING MONTH = 12)
1001 REM
1002 REM MTHNAME: Sample User Written Routine in Basic
1003 REM This sample is based on FOCUS/VMS 6.x sample.
1004 REM
1005 REM Only changes were to make it more like the standard
1006 REM sample (entry point of lowercase mthname (vs. MTHNAM)
1007 REM datatype of REAL (vs. DOUBLE) and use mixed case
1008 REM month names.
1009 REM
2000 ON INTEGER(MTH) GOTO 2001,2002,2003,2004,2005,2006, &
                          2007,2008,2009,2010,2011,2012 &
                          OTHERWISE 2013
2001 MONTH = "January" \ EXIT SUB
2002 MONTH = "February" \ EXIT SUB
2003 MONTH = "March" \ EXIT SUB
2004 MONTH = "April" \ EXIT SUB
2005 MONTH = "May" \ EXIT SUB
2006 MONTH = "June" \ EXIT SUB
2007 MONTH = "July" \ EXIT SUB
2008 MONTH = "August" \ EXIT SUB
2009 MONTH = "September" \ EXIT SUB
2010 MONTH = "October" \ EXIT SUB
2011 MONTH = "November" \ EXIT SUB
2012 MONTH = "December" \ EXIT SUB
2013 MONTH = "** Error **" \ EXIT SUB
3000 END SUB
```

MTHNAME RPG IBM i ILE Implementation

Note:

- This sample is stored in hlq.HOME.ETC(MTHNAMRP) for PDS deployment and as home/etc/src3gl/mthname.rpg on all other platforms.
- GENCPGM for RPG is only supported on i/OS and the sample on disk is only provided for reference for all other platforms.
- Code must start in column 7 of file.

Source:

```
HNOMAIN

* MTHNAME: Sample User Written Routine in RPG
*          Converts month number to month name

* This is an IBM i RPG version of the standard mthname.c
* sub routine supplied with IBI products.
* This a no main dll service type program with a lowercase
* exported symbol ... which is what is needed to integrate
* with programs that typically use lower or mixed case
* symbols in there dlls (ie. C).

* This routine is stored for z/OS PDS Deployment purposes
* as MTHNAMRP so it does not conflict with any of the
* other MTHNAME samples. Gencpgm on z/OS doesn't support
* RPG so building there is a non issue.

* Declare procedure parameter prototype.
* EXTPROC needed for lower case symbol ... very important!

D mthname         PR                    EXTPROC('mthname')
D   MTH                          8F
D   MTHNAME                     11A

* Procedure begin with external symbol export declaration.
P mthname         B                     EXPORT

* Declare procedure parameter interface.
D mthname         PI
D   MTH                          8F
D   MTHNAME                     11A
* Error Cases ... check if below 1 or above 12
C                   IF        MTH < 1 OR MTH > 12
C                   MOVE      '** Error **' MTHNAME
C                   ENDIF
```

```
* Look up by month ...
* (Using LOOKUP would be better, but lets keep it simple)
C                    IF        MTH =  1
C                    MOVE      'January     ' MTHNAME
C                    ENDIF
C                    IF        MTH =  2
C                    MOVE      'February    ' MTHNAME
C                    ENDIF
C                    IF        MTH =  3
C                    MOVE      'March       ' MTHNAME
C                    ENDIF
C                    IF        MTH =  4
C                    MOVE      'April       ' MTHNAME
C                    ENDIF
C                    IF        MTH =  5
C                    MOVE      'May         ' MTHNAME
C                    ENDIF
C                    IF        MTH =  6
C                    MOVE      'June        ' MTHNAME
C                    ENDIF
C                    IF        MTH =  7
C                    MOVE      'July        ' MTHNAME
C                    ENDIF
C                    IF        MTH =  8
C                    MOVE      'August      ' MTHNAME
C                    ENDIF
C                    IF        MTH =  9
C                    MOVE      'September   ' MTHNAME
C                    ENDIF
C                    IF        MTH =  10
C                    MOVE      'October     ' MTHNAME
C                    ENDIF
C                    IF        MTH =  11
C                    MOVE      'November    ' MTHNAME
C                    ENDIF
C                    IF        MTH =  12
C                    MOVE      'December    ' MTHNAME
C                    ENDIF

* Done; return to caller.
C                    RETURN
* Procedure End
P                  E
```

MTHNAME PL/1 Implementation

Note:

- ❏ This sample is stored in hlq.HOME.ETC(MTHNAMPL) for PDS deployment and as home/etc/src3gl/mthname.pl1 on all other platforms.
- ❏ PL/1 is only supported on z/OS and the sample on disk is only provided for reference for all other platforms.

- On PDS deployment use "-o mthname" to build using alternate source name of MTHNAMAS.
- PL/1 coding must start in column 2 of file.

Source:

```
/* MTHNAME: Sample User Written Routine in PL/1 */

MTHNAME: PROC(MTHNUM,FULLMTH) OPTIONS(COBOL);
DECLARE MTHNUM DECIMAL FLOAT (16) ;
DECLARE FULLMTH CHARACTER (9) ;
DECLARE MONTHNUM FIXED BIN (15,0) STATIC ;
DECLARE MONTH_TABLE(13) CHARACTER (9) STATIC
                      INIT ('January',
                            'February',
                            'March',
                            'April',
                            'May',
                            'June',
                            'July',
                            'August',
                            'September',
                            'October',
                            'November',
                            'December',
                            '**ERROR**') ;
MONTHNUM = MTHNUM + 0.00001 ;
IF MONTHNUM < 1 | MONTHNUM > 12 THEN MONTHNUM = 13 ;
FULLMTH = MONTH_TABLE(MONTHNUM) ;
RETURN ;
END MTHNAME ;
```

MTHNAME Pascal Implementation (Based on HP OpenVMS Pascal 5.8)

Note:

- This sample is stored in hlq.HOME.ETC(MTHNAMPS) for PDS deployment and ashome/etc/src3gl/mthname.pas on all other platforms.
- GENCPGM for Pascal is only supported on OpenVMS and the sample on disk is only provided for reference for all other platforms.

Source:

```
{
  MTHNAME: Sample User Written Routine in Pascal
  This sample is based on FOCUS/VMS 6.x sample.
  Only changes were to make it more like the standard
  C sample (entry point of lowercase mthname (vs. MTHNAM)
  and use mixed case month names).
}
MODULE MTH;
TYPE
  monthstring = packed array [1..12] OF CHAR;
[GLOBAL] PROCEDURE mthname(MTH:double ; var month : monthstring);
  VAR
IMONTH :INTEGER;
 BEGIN
   IMONTH:= ROUND(MTH);
   IF IMONTH IN [1..12] THEN
     CASE IMONTH OF
       1 : MONTH := 'January';
       2 : MONTH := 'February';
       3 : MONTH := 'March';
       4 : MONTH := 'April';
       5 : MONTH := 'May';
       6 : MONTH := 'June';
       7 : MONTH := 'July';
       8 : MONTH := 'August';
       9 : MONTH := 'September';
      10 : MONTH := 'October';
      11 : MONTH := 'November';
      12 : MONTH := 'December';
    END
  ELSE
    MONTH := '** Error **'
 END;
END.
```

UREVERSE C Implementation

Note:

- This sample is stored in hlq.HOME.ETC(UREVERSE) for PDS deployment and as home/etc/src3gl/ureverse.c on all other platforms.
- Sample reverses a string, but checks server codepage for UTF-8 (65001) condition and handles the string accordingly.
- The example is not only an example of how to handle UTF-8, but also how an existing routine can be updated and use getenv() to obtain codepage information instead of using a passed parameter, which would in turn lead to having to also update FOCEXEC application code for the extra parameter (which may or may not be frequent within an application).

❑ Comments within example describe typical usage and how to test.

```
/*                                                                               */
/* Sample User Written Routine in C ...                                          */
/* UREVERSE: Unicode UTF-8 capable string reversing routine                      */
/*                                                                               */
/* Typical usage:                                                                */
/* -SET &STRING = 'abcd' ;                                                       */
/* -SET &RSTRING = UREVERSE(&STRING,&STRING.LENGTH,&FOCCODEPAGE,A&STRING.LENGTH) ; */
/* -TYPE Reverse of &STRING is &RSTRING                                          */
/* Note: &FOCCODEPAGE is standard amper variable for server code page            */
/*                                                                               */

/* Servers using the Unicode 65002 page are effectively UTF-EBCDIC and beyond      */
/* the scope of this simple sample. Customer implementations should follow the     */
/* information at http://www.unicode.org/reports/tr16 when using the 65002         */
/* UTF-EBCDIC code page.                                                           */

#include <stdio.h>
#include <stdlib.h>

void ureverse( char *instr, double *charsize, double *codepage, char *outstr )
{
  unsigned short codepg = (unsigned short)*codepage;
  int            csize = (int)*charsize;
  int            bsize, offset, clen, ccnt;
  unsigned char *cptr;
  char          *foccodepage;

  /* External var override, normally var is not set. If trying to make an       */
  /* existing routine Unicode compliant without passing an extra var, this       */
  /* method can be used to get a code page value if following is added to        */
  /* the server profile (edasprof) or other application code:                    */
  /* -SET &RC = FPUTENV(11,'FOCCODEPAGE',&FOCCODEPAGE.LENGTH,&FOCCODEPAGE,D8) ; */

  foccodepage = getenv("FOCCODEPAGE");
  if( foccodepage != NULL )
  {
    codepg = atoi( foccodepage );
  }
```

```
    if( codepg == 65001 ) /* Unicode reference number used by server for UTF-8 */
    {
      /* Unicode UTF-8 */
      /* Pass 1. Calculate the byte length of 'instr' in character length 'charsize' */
      /* Pass 2. Copy each character from 'instr' to 'outstr' in reverse             */
      bsize = csize * 3; /* maximum byte size */
      for( ccnt = offset = 0; ccnt < csize && offset < bsize; ccnt++, offset += clen )
      {
        cptr = (unsigned char *)&instr[offset];
        if(      *cptr < 0x80 )  clen = 1;
        else if( *cptr < 0xE0 )  clen = 2;
        else                     clen = 3;
      }
      bsize = offset; /* actual byte size in utf-8 for charsize */
      for( offset = 0; offset < bsize; offset += clen )
      {
        cptr = (unsigned char *)&instr[offset];
        if(      *cptr < 0x80 )  clen = 1;
        else if( *cptr < 0xE0 )  clen = 2;
        else                     clen = 3;
        memcpy( &outstr[bsize - offset - clen ], cptr, clen );
      }
    }
    else
    {
      /* Non-Unicode */
      /* Copy each character from 'instr' to 'outstr' in reverse */
      for( offset = 0; offset < csize; offset++ )
      {
        outstr[csize - offset - 1] = instr[offset];
      }
    }
  }
```

Index

Feedback

Customer success is our top priority. Connect with us today!

Information Builders Technical Content Management team is comprised of many talented individuals who work together to design and deliver quality technical documentation products. Your feedback supports our ongoing efforts!

You can also preview new innovations to get an early look at new content products and services. Your participation helps us create great experiences for every customer.

To send us feedback or make a connection, contact Sarah Buccellato, Technical Editor, Technical Content Management at *Sarah_Buccellato@ibi.com.*

To request permission to repurpose copyrighted material, please contact Frances Gambino, Vice President, Technical Content Management at *Frances_Gambino@ibi.com.*

Information Builders

WebFOCUS

Stored Procedure and Subroutine Reference for 3GL Languages

WebFOCUS Reporting Server Release 8205

DataMigrator Server Release 7709 and Higher

DN4501044.0219

Information Builders, Inc.
Two Penn Plaza
New York, NY 10121-2898