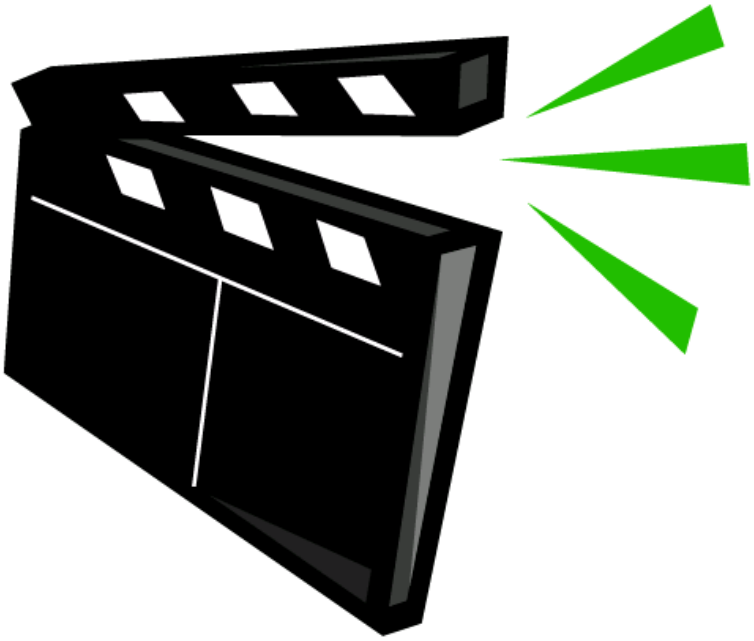


cobolscript®
Developer's Guide

cobolscript® Developer's Guide

Copyright © 1996-2017 Matt Dean. All Rights Reserved.

Copyright © 1996-2017 Matt Dean. All Rights Reserved.

This manual and its entire contents are copyrighted material. No part of this manual may be reproduced in any form or by any means, either electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Matt Dean. Information contained herein is subject to change without prior notice. All names and data in this manual are fictitious except where otherwise noted. The software described in this manual is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement.

The term "cobolscript" is a registered trademark of Matt Dean. All other product names, including but not limited to the terms "VACE" , "VACE Maintenance Workbench" are trademarks or registered trademarks of Matt Dean.

Microsoft Windows and MS-DOS are registered trademarks of Microsoft Corporation.
Linux is a registered trademark of Linus Torvalds.
FreeBSD is a registered trademark of FreeBSD Inc. and Walnut Creek CDROM.
SunOS, Solaris, and Sun are registered trademarks of Sun Microsystems, Inc.
SQL*Loader and Oracle are registered trademarks of Oracle Corporation.

All other brand and product names mentioned herein are trademarks or registered trademarks of their respective holders.

Matt Dean
PO Box 6066
Chattanooga, TN 37401

Phone: 423-888-7475
World Wide Web: www.cobolscripting.cloud

Enjoy your programming.

Table of Contents

Chapter 1	**Introduction to CobolScript® / Installation Instructions……**	**1**
	CobolScript Features……	2
	About this Manual……	3
	Installing CobolScript……	4
Chapter 2	**Getting Started with CobolScript®……**	**9**
	Creating and Editing CobolScript Programs……	9
	Running CobolScript from the Command Line……	10
	Running CobolScript in Interactive Mode……	14
	Running CobolScript from a Web Server and Browser……	17
Chapter 3	**CobolScript® Language Constructs……**	**21**
	Literals and Literal Keywords……	21
	Variables……	24
	Data and Copybook Files……	31
	Expressions and Conditions……	34
	Commands……	39
	CobolScript Reserved Words……	42
	Statements……	43
	Sentences……	44
	Comments……	45
Chapter 4	**File Processing and I/O……**	**47**
	Describing Files and Defining Data Records……	48
	Opening Files……	48
	Closing Files……	48
	Reading Records From Files……	49
	Overwriting a File……	50
	Appending Records to an Existing File……	50
	Writing to a File by Updating Existing Records ……	52
	Relative and Absolute File Positioning……	53
	Relational Database Interaction with CobolScript Standard Edition……	55
Chapter 5	**Building Web Based Systems……**	**63**
	Interacting with a Web Server and Web Browser……	64
	Creating Virtual HTML……	65
	Creating an HTML Form……	66
	Capturing Input Data from a Web Page……	66
	DISPLAY and DISPLAYLF……	68
	Retrieving Web Pages……	69
Chapter 6	**Network and Internet Programming Using CobolScript®……**	**71**
	Transferring Files using FTP……	71
	Using Email Commands……	73
	Using TCP/IP Commands……	75
Chapter 7	**Advanced Internet Programming Techniques Using CobolScript®..**	**83**

	Environment Variables..…….…….…….……	84
	CGI Form Components……..…….	86
	Using Hidden Fields……..	90
	Sending Email from CobolScript Using CGI Form Input...……..	92
	Using CobolScript to Transmit Files……..…….....	93
	Embedding JavaScript in CobolScript Programs….…….…….	95
Chapter 8	**Programming Techniques and Advanced CobolScript® Features….**	**99**
	Designing a Modular Program..….…….…….…….	99
	Manipulating CobolScript Variables…….	101
	Advanced CobolScript Features...…...…….…….……	102
Chapter 9	**CS Professional CodeBrowser™, AppMaker™, and Control Panel...**	**109**
	Feature Requirements..….…….…….…….	109
	Using CodeBrowser…….	109
	Building Executables with AppMaker.…….…….……	112
	Using the CobolScript Control Panel..…….…….……	113

Appendixes

Appendix A	**Language Reference…….…….…….…….…….**	**117**
Appendix B	**Function Reference…….…….…….…….**	**159**
Appendix C	**CobolScript® Constraints……..…….…….……...**	**175**
Appendix D	**Sample CobolScript® Programs……...…….…….**	**177**
Appendix E	**CobolScript® Picture Clauses…….…….…….…….…**	**181**
Appendix F	**CobolScript® Basic Program Structure…….……...**	**187**
Appendix G	**Setting Up ODBC and ODBC Data Sources for LinkMaker™……..**	**195**
Appendix H	**Using LinkMaker™ Embedded SQL in CobolScript® Professional..**	**223**
Appendix I	**CobolScript® Error Messages..…….…….…….…**	**231**

Glossary…….…….…….…….…….…….…….…….……... **259**
Index…….…….…….…….…….…….…….…….……. **265**

Chapter 1

Introduction to CobolScript® / Installation Instructions

CobolScript® is a powerful, easy to use, platform independent, internet-friendly programming language. With it, you will be able to quickly develop and test web-based systems, interface programs, and compact business applications. The natural syntax of CobolScript will help you to start programming productively in a short amount of time, provided you've had at least some exposure to other programming languages. This natural syntax, coupled with a variety of network and internet-specific commands, makes CobolScript a great alternative to more cryptic or complicated network programming languages. If you're an experienced internet developer, we think you'll find that certain web programming tasks that used to be difficult with your old language will be simple with CobolScript, and as a side benefit, your code will be more manageable and easier to maintain. If you've avoided web programming in the past because of its apparent complexity, CobolScript can open the door to a whole new style of application development for you, and can do it with a relatively small effort.

ICON KEY

➲ Important point

CobolScript is available for Microsoft Windows®, SunOS®, FreeBSD®, and Linux®. Any program developed and tested on one platform can be almost seamlessly ported to another supported platform. And like all web systems, CobolScript web apps can be executed from any machine that has a compatible browser and can access the web that is running CobolScript. For this reason, a well coded, web-based CobolScript system will not require modification if client machines are changed or upgraded, so long as the clients still have compatible browsers installed – a welcome change for anyone who has had to modify applications with client front-ends specific to their operating system.

Alternatively, a single web client can run different CobolScript applications that reside on separate servers. By linking small applications that are located on distinct servers to one another, you can create a complete web system, and the processing for this single, larger system will be spread across the servers. Figure 1.1 illustrates one possible architecture for such a system.

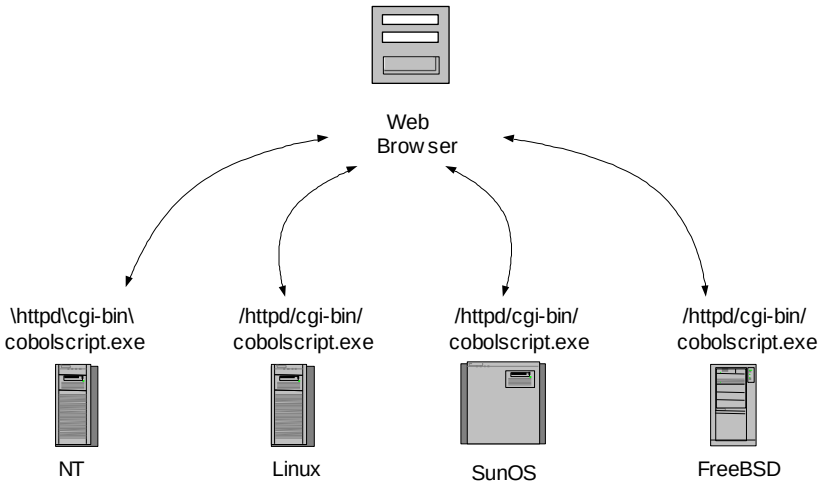


Figure 1.1 – A multi-server CobolScript application

In addition to the standard language commands and the internet processing commands available in CobolScript, other features provide the means to quickly and easily create programs with a wide range of functionality:

- Internetworking commands such as FTPPUT, FTPGET, SENDMAIL, and GETMAIL for transferring files and emails from within a CobolScript program.
- File processing commands for reading and parsing both fixed-format and delimited data files.
- Flexible naming syntax that allows underscores (_) and dashes (-) to be used interchangeably in variable names, to support both modern variable naming as well as COBOL-style variable naming.
- Advanced expression evaluator that does not require explicit spaces between expression components, even for subtraction operations, for programmers who are used to coding mathematical expressions in C or similar languages.
- Financial functions for calculating annuities and depreciation.
- Scientific, stochastic, and other higher math functions.
- Metric to English and English to metric system unit conversion functions.
- TCP/IP socket programming commands such as SENDSOCKET and RECEIVESOCKET, for creating client-server communications programs without web server software or FTP configuration.
- DNS commands such as GETHOSTNAME for incorporating internet information retrieval into programs.
- PIC X(n) picture clause that automatically calculates variable size based on VALUE clause, eliminating the need for time-consuming computations with FILLER variables, and an implied version of PIC X(n) that allows the FILLER keyword and picture clause to be eliminated entirely.
- REPLICA variable declaration syntax that permits the same elementary data item to be used in multiple group items.

- EXECUTE command for dynamic statement creation and execution.
- Intelligent error messaging that displays browser-based error messages when running programs from a browser, and text-based error messages when running programs from the command line, thereby speeding the debugging process.

Using these CobolScript features, you can develop programs to get and save web pages to text files, transfer files via FTP, send simple emails, retrieve emails, accept data from web page forms, create virtual HTML documents, and perform various file input and output operations.

CobolScript Professional Edition also contains a number of enhancements that enable professional development with CobolScript:

- CobolScript AppMaker™, which makes it possible to create executables from CobolScript programs.
- CobolScript CodeBrowser™, a browser-based utility to examine your code in colorized form.
- CobolScript LinkMaker™, a tool that enables you to directly embed SQL calls in your CobolScript program to access any data source for which you have an ODBC driver. On Unix platforms, LinkMaker™ is used in conjunction with UnixODBC, a freeware product.
- The CobolScript Control Panel, a graphical administration tool accessible from your web server machine (so long as both CobolScript and web server software are installed), for accessing other CS Professional features, and for administering your CS Professional system.
- Multidimensional array support.

About this Manual

This developer's guide should serve as both a guide for learning to program with CobolScript, and as a reference for your day-to-day programming. It should provide sufficient instruction for most experienced programmers to learn to develop CobolScript applications; however, in certain instances you may wish to find additional information:

- If you are completely new to the art of programming, you should probably familiarize yourself with introductory programming principles as well. Understanding the basics of programming will reduce the time it takes you to learn CobolScript.
- If you choose to program web applications using CobolScript, you should be familiar with HTML. HTML is relatively easy to learn, and many good web sites and books exist on the topic, so it would be redundant to include an HTML reference in this guide. A number of 'WYSIWYG' (What You See Is What You Get) software tools are also freely available and can assist you in prototyping your system and creating the HTML that will be displayed by your programs. Check www.download.com for the latest freeware and shareware WYSIWYG tools.
- Although web programming *is* addressed in this guide, you may also choose to seek more in-depth coverage of the subject, if, for instance, you want background information about CGI or about concepts not in this manual, such as cookie creation using CGI scripting.
- If you are interested in providing more real-time user feedback than is possible with just CGI scripting, or you want to distribute some of your web served application's processing to client machines, consider learning more about an appropriate embedded language like

JavaScript. These languages' scripts can be embedded in the HTML that is displayed by your CobolScript programs, so you can provide real-time, client-based processing while still using CobolScript. Our preferred client-side scripting language is JavaScript, since it loads and executes relatively quickly, and will run on both Netscape Navigator® and Internet Explorer®.

If you are looking for books on any of the above topics, we've found the Peachpit Press *Visual Quickstart Guide* series to be affordable, concise, readable for beginners but not overly simplified, and filled with good examples. Peachpit Press is on the Web at www.peachpit.com.

Installing CobolScript

System Requirements

A Pentium®-compatible machine (166 MHz and higher preferred) is required for the Windows®, Linux®, and FreeBSD® versions of CobolScript, a RISC-processor machine for the SunOS® version. 32MB of RAM is recommended for CobolScript Standard, more for programs of substantial size. 64MB of RAM is recommended for CobolScript Professional Edition.

Installing CobolScript on a Windows®-compatible machine

Step 1. Download CobolScript.

Create a directory such as C:\DESKWARE or C:\COBOLSCRIPT where you will keep CobolScript and your CobolScript programs. Download the file(s) to that directory from the Deskware Registered User Web Site. If you have downloaded a zip file (with the extension .zip), unzip it using WinZip or a similar product. The cobolscript.exe file is the CobolScript interpreter, and the .cbl files are the sample CobolScript programs. As you have already discovered because you are reading this, this manual is the file cbmanual.pdf, and requires that you have a free copy of Adobe Acrobat Reader®, version 4.0 or higher, installed on your computer to read and print it.

Step 2. Install CobolScript.

No special configuration is required for CobolScript to run. However, we recommend that you modify your PATH environment variable in your AUTOEXEC.BAT file to point to the location of the CobolScript engine. To do this, first save a copy of your old C:\AUTOEXEC.BAT file to a backup file such as C:\AUTOEXEC.BAK, then open AUTOEXEC.BAT in a text editor such as notepad, and modify the SET PATH= line. For example, if a line in your AUTOEXEC.BAT file reads:

```
SET PATH=C:\MOUSE;%PATH%;C:\PP\BIN\WIN32
```

you would change it to:

```
SET PATH=C:\MOUSE;%PATH%;C:\PP\BIN\WIN32;C:\DESKWARE
```

if you have saved the CobolScript engine to the C:\DESKWARE directory.

Step 3. Run CobolScript.

CobolScript can be run from the command line. Start an MS-DOS prompt, and type:

```
cobolscript.exe
```

to run CobolScript and see the command line options. To run a specific program from the command line, type:

```
cobolscript.exe <program-name>
```

where <program-name> is the name of the program you wish to run, along with a path if the program is not in the current directory. For example:

```
cobolscript.exe test.cbl
cobolscript.exe ..\testdir\test.cbl
```

For more information on running CobolScript from the command line, turn to the next chapter, *Getting Started with CobolScript*.

If you plan to do Web and CGI development, you will probably want to put CobolScript in your web server's CGI directory. Usually this directory has "cgi" or "cgi-bin" in the name, as in c:\httpd\cgi-bin for the OmniHTTPd web server. Just place the cobolscript.exe file in this directory. See the section titled **Running CobolScript from a Web Server and Browser** in Chapter 2, *Getting Started with CobolScript*.

If you don't already have a web server, *OmniHTTPd* is a freeware development-quality web server for Windows 95/98/NT®. Search the web for "OmniHTTPd" to find a copy.

Step 4. Configure ODBC on your computer.

If you have CobolScript Professional Edition and you want to access a database using LinkMaker™, you will need to set up an ODBC data source on your computer. Refer to Appendix H for complete instructions on how to do this.

Installing CobolScript on a Linux®, SunOS®/Solaris®, or FreeBSD® machine

Step 1. Download CobolScript.

Create a directory such as /deskware or /cobolscript where you will keep CobolScript and your CobolScript programs. Download the file(s) to that directory from the Deskware Registered User Web Site. If you have downloaded the complete file, un-tar it with the appropriate command (depending on your OS). Below are some un-tarring examples:

```
tar -xvf linuxcob.tar
tar -xvf suncob.tar
tar -xvf bsdcob.tar
```

Similar steps should be followed with other tar files; just use the same syntax as above and substitute the appropriate filename. The cobolscript.exe file is the CobolScript interpreter, and the .cbl files are the sample CobolScript programs. As you have already discovered because you are reading this, this

manual is the file cbmanual.pdf, and requires that you have a free copy of Adobe Acrobat Reader® 4.0 or higher installed on your computer to read and print it. Because there is not a version of Adobe Acrobat Reader® available for FreeBSD, if you have purchased this version of CobolScript you will have to print the manual from an Acrobat®-compatible OS (Windows®, Linux®, IRIX®, HP-UX®, AIX®, Solaris®, Macintosh®, etc.).

Step 2. Install CobolScript.

No special configuration is required for CobolScript to run. However, we recommend that you modify your PATH environment variable to point to the location of the CobolScript engine. To do this permanently (preferred), you can modify the appropriate line of your .profile file in your home directory. For example, if a line in your .profile file reads:

```
PATH=/bin:/sbin
```

you should change it to:

```
PATH=/bin:/sbin:/deskware
```

in the case where CobolScript is in the /deskware directory. If you are going to run CobolScript from your current directory only, make certain that "./" is also a component of the PATH variable.

To modify your PATH environment variable for the current session only, first type:

```
echo $PATH
```

at the command prompt to see the current value of your PATH environment variable. Next, on Linux® or Sun® machines, at the command prompt type:

```
PATH=$PATH:/deskware
```

where /deskware is the path to the CobolScript interpreter. In FreeBSD, you should instead type:

```
setenv PATH oldpath:/deskware
```

or alternatively:

```
set path=oldpath:/deskware
```

where *oldpath* is the original value of the PATH variable, and */deskware* is the path to the CobolScript interpreter. Your path will be changed for the current session.

Step 3. Run CobolScript.

CobolScript can be run from the command line. Bring up an xterm or command prompt, and type:

```
cobolscript.exe
```

to run CobolScript and see the command line options. To run a specific program from the command line, type:

```
cobolscript.exe <program-name>
```

where <program-name> is the name of the program you wish to run, along with a path if the program is not in the current directory. For example::

```
cobolscript.exe test.cbl
cobolscript.exe ../testdir/test.cbl
```

For more information on running CobolScript from the command line, turn to the next chapter, *Getting Started with CobolScript*.

If you plan to do Web and CGI development using CobolScript, you will probably want to put CobolScript in your web server's CGI directory. Usually this directory has "cgi" or "cgi-bin" in the name, as in /home/httpd/cgi-bin on Apache. Just place the cobolscript.exe file in this directory.

If you are doing CGI development and intend to read and write to files in your cgi-bin directory, make certain that the permissions on these files (and on the cgi-bin directory, and its parent directories) are correctly set. Use the **chmod** command at the command prompt to properly set file permissions. If this is not done, you will encounter difficulties when running scripts from a web browser, since these scripts generally run as user 'nobody', who does not have the same authority as you do when you are logged in at a command prompt, creating these files.

Step 4. Set up ODBC on your computer.

If you have CobolScript Professional Edition and you want to access a database using LinkMaker, you will need to set up an ODBC data source on your computer. Refer to Appendix H for instructions on how to set up UnixODBC (a freeware product from UnixODBC.org) so that you can connect directly to your data source.

Chapter 2

Getting Started with CobolScript®

Before you dive headfirst into CobolScript programming, you will need to learn the basics, like how to edit your CobolScript programs, how to run them, and how to debug them. This chapter aims to answer the basic logistical questions of CobolScript coding that you may have, as well as providing a background on CobolScript *interactive mode*, which contains some useful debugging tools. With the information here, you'll be ready to learn the CobolScript language.

ICON KEY

➲ Important point

Just as a note, all of the screens shown in this chapter, with the exceptions of the Windows®-specific information in Figures 2.1 and 2.2, are representative of any CobolScript platform; don't worry about whether the figure shows an MS-DOS screen or a Unix screen, because the syntax and output of the illustration would be the same no matter what the platform.

Creating and Editing CobolScript Programs

Use a standard text editor to create and edit your CobolScript programs. In Windows®, editors such as Notepad or Wordpad work well. If you use Wordpad, make certain you save your files as text

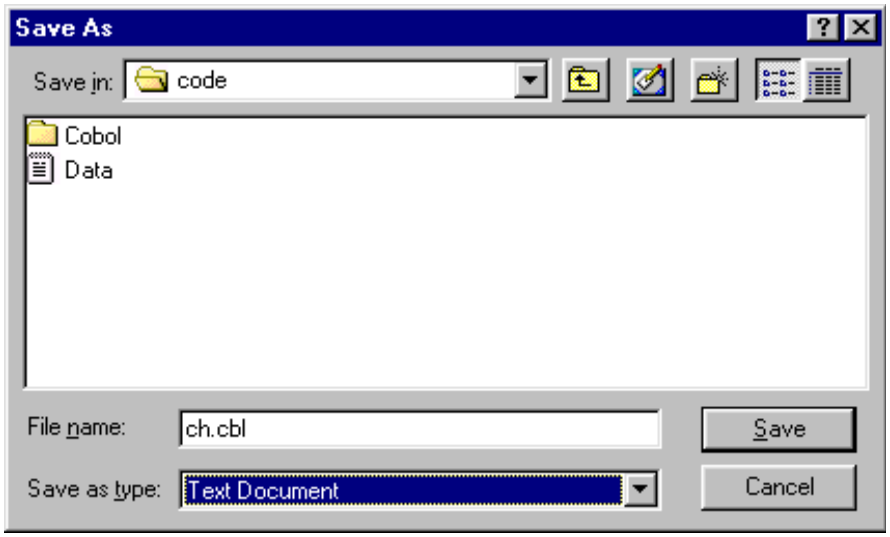


Figure 2.1 – Saving a CobolScript program in the Microsoft® Wordpad Save As dialog box.

documents, and specify the extension when naming your program, as in Figure 2.1, or Wordpad will save the file with a default extension of *.txt*. Also, in Wordpad you'll find it easiest if you choose a

fixed-width font for your editing such as Courier New. This will allow you to later open your programs in Notepad, MS-DOS EDIT, or in Unix without a loss of formatting. You will probably find yourself using the aforementioned MS-DOS EDIT text editor (accessible by typing the word **edit** at the DOS prompt) when debugging, because despite its old-fashioned appearance, it tracks the current column and row positions of the cursor, which can allow you to quickly locate a program line number. Figure 2.2 shows an EDIT screen, with the cursor positioned down and to the right of center; the resulting Line and Column position values appear in the lower right corner of the screen.

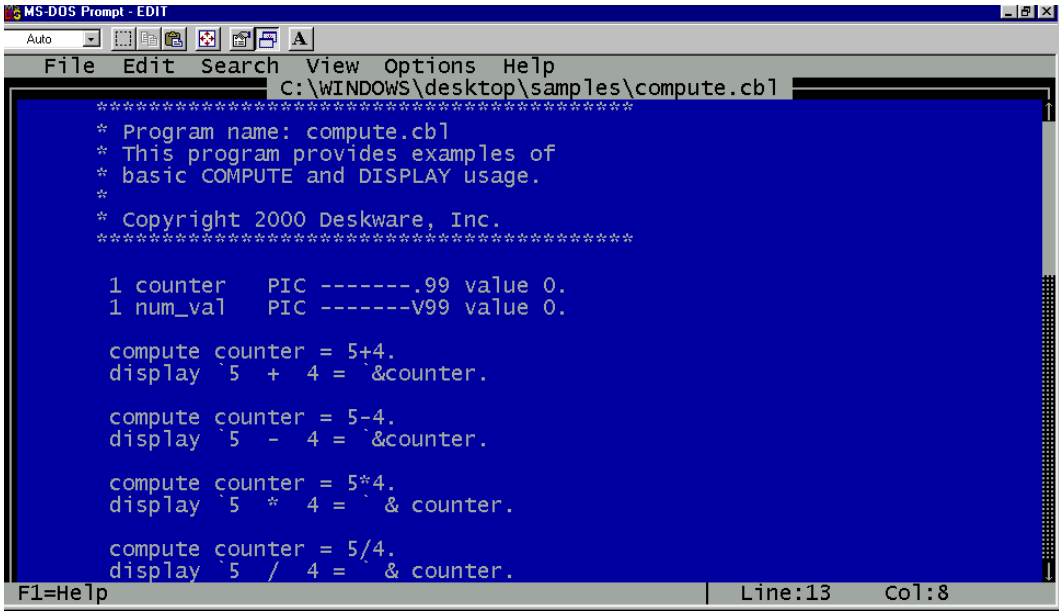


Figure 2.2 – The MS-DOS EDIT text editor, showing the current cursor position (Line and Column) in the lower right corner.

If you choose to edit your programs in Unix, any editor that saves documents as plain ASCII text will suffice. Like MS-DOS EDIT, **vi** is a useful editor because it provides the means to quickly navigate to a particular line number. Teaching vi is beyond the scope of this manual, however, so refer to a Unix or vi-specific reference for more information.

If you are using CobolScript Professional Edition, you will probably find CobolScript CodeBrowser™ to be a useful tool for printing and examining your programs; CodeBrowser™ is discussed in detail in chapter 9.

Running CobolScript from the Command Line

The simplest way to use CobolScript is by running a CobolScript program in command line mode. To do this, type:

```
cobolscript.exe <program-name>
```

at the command prompt, where <program-name> is the name of the program you wish to run (don't literally enclose the program name in < >; we use this syntax to indicate that program-name is an *argument* to the CobolScript executable, cobolscript.exe). This command assumes that you have already included your CobolScript directory in your PATH environment variable, or alternatively, that you are executing the command from within the CobolScript directory. If you need instructions

on how to include your CobolScript directory in your PATH variable, refer to the **Installing CobolScript** section of Chapter 1, *Introduction to CobolScript / Installation Instructions*.

If you're using the Windows® version of CobolScript (rather than a Unix version), running from command line mode means that you are running your CobolScript programs in an MS-DOS session. However, it is important to note that although your CobolScript applications can be run from the DOS prompt, CobolScript is *not* a DOS application; it is a native 32-bit application that excludes Windows-specific graphical components in order to minimize the CobolScript engine's footprint and to provide cross-platform capability. Graphical development with CobolScript is achieved through the use of a web server and browser-based applications, discussed in more detail in Chapters 6 and 8. See the section titled **Running CobolScript from a Web Server and Browser** later in the chapter for more information on getting started in a web-based environment.

Running CobolScript from Windows® command-line mode, you can drop the extensions if you like, and just type:

```
cobolscript <program-name>
```

Command line program execution will direct all output to the current command line window, and

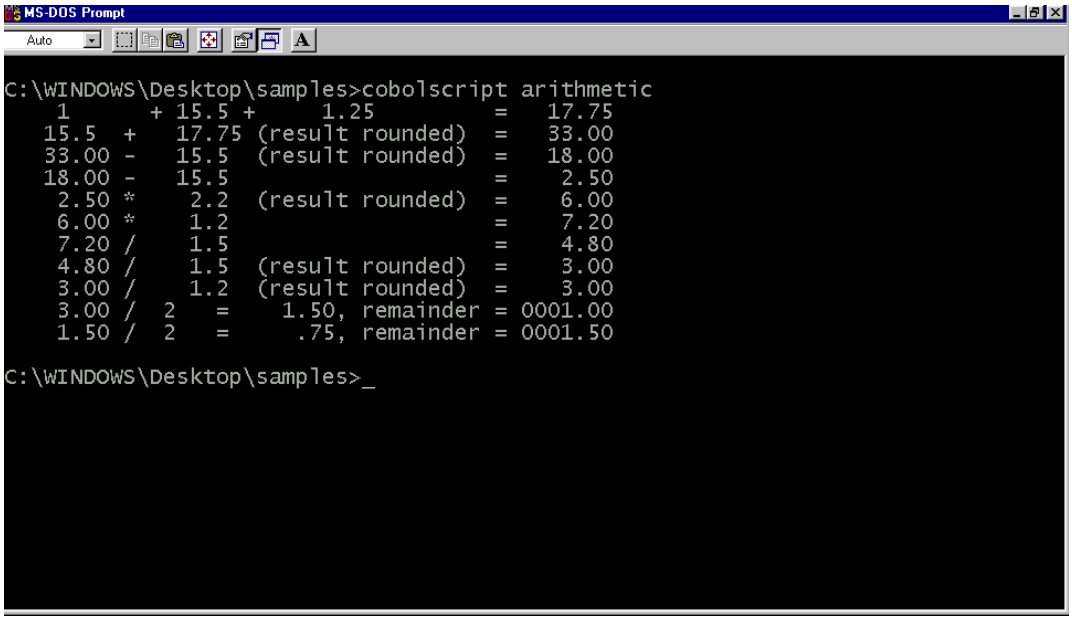


Figure 2.3 – Executing CobolScript programs from the command line prompt.

therefore all output will be plain text. Several of the example programs contained with CobolScript are designed to run in command line mode; figure 2.3 shows the output of the ARITHMETIC.CBL example program in an MS-DOS prompt window.

CobolScript also comes with a number of *command line options*. If you aren't already familiar with the term, a command line option is a switch that you set at the time that you call an executable program, which in this case is the CobolScript executable. These switches allow you to change some specifics in the way that CobolScript runs, at the time you run it. If you type `cobolscript.exe` at the command line prompt, without any program arguments specified, you will see a list of the CobolScript command line options.

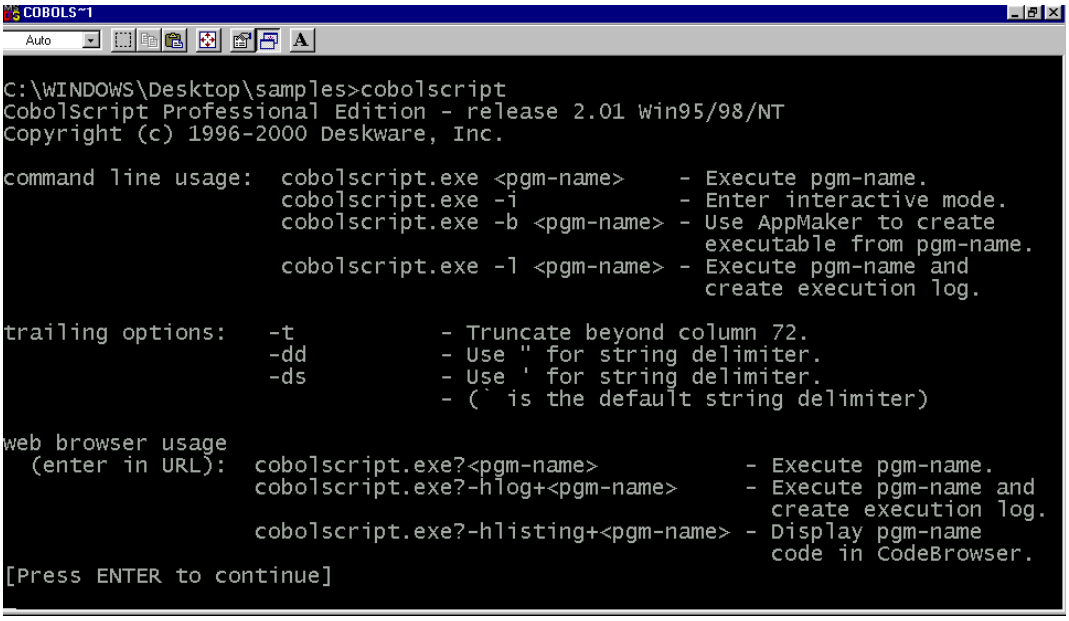


Figure 2.4 – CobolScript Professional command line options.

Figure 2.4 illustrates this in an MS-DOS window (on other platforms, you would need to specify the .exe extension to the CobolScript executable). The syntax of the CobolScript Standard Edition command line options is as follows:

```
cobolscript.exe  [-i|-l] <program-name> [-t|-dd|-ds]
```

- The **-i** option runs the interpreter in interactive mode; see below for more information on running in interactive mode. When the -i option is used, if a *program-name* is not specified, interactive mode will be entered with nothing in the program buffer. If *program-name* is specified, interactive mode will be entered, and *program-name* will be loaded into the program buffer.
- The **-l** option runs <program-name> and creates a listing of the program execution as a separate log file with the name *program-name.log*. For example, if your program name is test.cbl, and you type the following at the command prompt:

  ```
  cobolscript.exe -l test.cbl
  ```

  then a log file named `test.log` will be created in the working directory.

- The **-t**, **-dd**, and **-ds** options are options that come *after* the program name:
  ⇒ The **-t** option causes CobolScript to truncate (and ignore) all characters beyond the 72nd column position when parsing the program; this mimics the way mainframe COBOL works. Your program file is not affected, just the execution of the program. The default (no –t specified) is for all characters in the program to be treated as code.
  ⇒ The **-dd** option causes CobolScript to recognize the double quote character (") as the string delimiter instead of the default, the accent symbol (`). To display a literal double quote when using this option, your program must use the keyword DOUBLEQUOTE. The -dd and -ds options are mutually exclusive.
  ⇒ The **-ds** option causes CobolScript to recognize the single quote character (') as the string delimiter instead of the default, the accent symbol (`). To display a literal single

quote when using this option, your program must use the keyword SINGLEQUOTE. The -dd and -ds options are mutually exclusive.

CobolScript Professional Edition also provides a utility to build executables from the command line, CobolScript AppMaker™. The syntax for creating an executable using AppMaker is:

```
cobolscript.exe -b <program-name>
```

If your program successfully loads, an executable will be created from it and placed in the working directory. For example, typing the following will create an executable named *test.exe* in the working directory:

```
cobolscript.exe -b test.cbl
```

CobolScript Error Messages in Command Line Mode

It will be normal for you to encounter bugs in your code while you are testing your CobolScript programs. In command line mode, error messages display directly to the screen in a text-based format. The error messages are quite specific, and will usually help you pinpoint the source of the problem with your code.

Multiple error messages are displayed when a single line of code causes multiple errors in the CobolScript engine; in these cases, one of these multiple errors should be obviously more specific than the others, and will better assist you in determining the problem than the more general messages. Multiple error messages, however, never indicate that there are unrelated errors on different lines of the program. This is because CobolScript is an interpreted language, and program execution is halted as soon as a single error is encountered. For this reason, you must re-run your program after correcting each error to determine if there are other errors in your code.

All error messages have an associated CobolScript Error Number, which displays along with the error message when the error is encountered; all error messages are explained in detail in Appendix

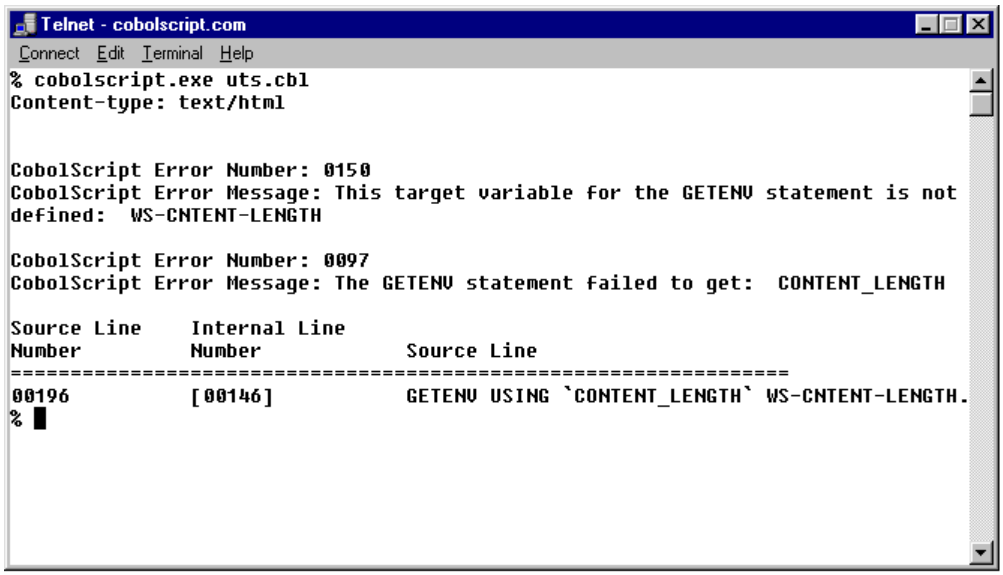


Figure 2.5 – CobolScript command line error message.

F, *CobolScript Error Messages*, in order of this CobolScript Error Number.

After the last error message is displayed for a particular error, the text of the line that caused the error is displayed, along with some line number information. The *Source Line Number* is the actual number of the line in the program text file that caused the error. Use this line number to navigate to the line of faulty code in your program with a text editor like MS-DOS EDIT or vi. The *Internal Line Number* indicates the number assigned to the Instruction Pointer (IP) at the time of the error. This number can be used when a program is run in interactive mode to determine the problem line, in conjunction with the **list** and **ip** interactive mode commands. Finally, the *Source Line* is the text of the line that caused the error. Figure 2.5 shows an example of a command line error and the resulting error messages.

Running CobolScript in Interactive Mode

From CobolScript interactive mode, you can load a program, execute it, step through and animate its execution, and examine the contents of your program's variables as they are populated. These features make interactive mode a great debugging tool. Interactive mode can be accessed by using the **-i** command line option when running CobolScript from the command prompt. Refer to the explanation of the CobolScript command line options above for the appropriate command line syntax. Figure 2.6 shows the start of a CobolScript interactive mode session on SunOS; interactive mode on other supported platforms is essentially the same.

Once you've started an interactive mode session, you'll see the CobolScript interactive mode prompt that looks like this:

```
cobolscript>
```

From this prompt, you can use all of the interactive mode commands, although some commands

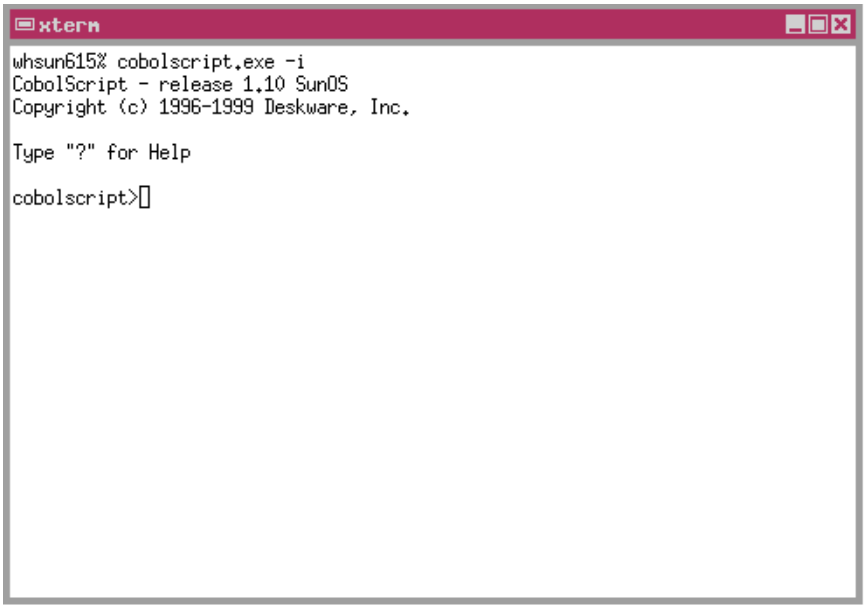


Figure 2.6 - Interactive Mode in SunOS®

will not work properly until a program has been loaded, and others will not work correctly until a program has been run. To see a help screen-style list of these commands, type a question mark (?) at the command prompt. Figure 2.7 shows a representation of this list of commands.

```
+-----------------------------------------------------------+
| CobolScript 2.01 Copyright (c) 1996-2000 Deskware, Inc. |
+-----------------------------------------------------------+
| COMMANDS:                                                 |
|                                                           |
| ?                     dump modules      positions         |
| ! <system command>    dump positions    q                 |
| animate <speed>       dump variables    run               |
| break <linenumber>    files             save <filename>   |
| clear                 help <command>    stack             |
| count                 ip                stepoff           |
| deskware              list              stepon            |
| display <variable>    load <filename>   variables         |
| dump listing          modules           ver               |
|                                                           |
|                                                           |
+-----------------------------------------------------------+
```

Figure 2.7 - Interactive Mode Help Screen example

Interactive Mode Commands

The following list defines the interactive mode commands. Online command-specific help is also available in interactive mode by typing help <command>.

Interactive Mode Command	Description
?	Displays all of the commands available in interactive mode.
! `system command`	Runs a system command on your machine. The system command must be an operating system command in the appropriate syntax for your operating system. Examples: `! dir` `! `dir | more`` `! `ls -al`` `! `chmod 777 test.cbl``
animate <speed>	Executes the code that is in the program buffer line by line, and displays each line of code as it is executed. The *speed* parameter controls the speed of the code interpreting and displaying process: the higher the number, the slower the lines of code will be displayed.
break <linenumber>	Sets a break point to halt program execution. The **break** command has the following forms: • **break** with no argument specified lists all current break points; • **break <linenumber>** sets a break point in a program's execution at *linenumber*; • **break clear <linenumber>** clears the existing break point at *linenumber*; • **break clear all** removes all existing break points.
clear	Removes the contents of the current program buffer. After the **clear** command is used, another program can be loaded into the buffer.
count	Displays the number of lines of code in the program currently loaded in the program buffer.
deskware	Displays Deskware, Inc. contact information.
display <variable>	Displays the contents of the specified *variable*. The **display** command can be used after **run**, **animate <speed>**, or **stepon** has been used to execute a loaded program.
dump variables **dump modules** **dump positions** **dump listing**	Creates a text file dump of all variable contents, a module list, a program listing, or a variable position listing, depending on the argument. Below are the names of the files that are created by each command: • **dump variables** - dump.var • **dump modules** - dump.mod • **dump positions** - dump.pos • **dump listing** - dump.lst

Interactive Mode Command	Description
files	Displays all of the files that a program used as it was executed. The **files** command can be used after **run**, **animate <speed>**, or **stepon** has been used to execute a loaded program.
help <command>	Displays command-specific help.
ip	Displays the current value of the CobolScript internal instruction pointer. This value is equivalent to the internal line number of the line that was just processed.
list	Displays the contents of the program buffer to the screen. The program buffer contains the lines of program code that were loaded with the **load <filename>** command.
load <filename>	Loads the contents of the specified program file *filename* into the program buffer. Once loaded, a program file can be executed by using the **run** or **animate <speed>** command.
modules	Displays all of the modules defined in the code that has been loaded into the program buffer.
positions	Displays all variables' byte offsets. The **positions** command can be used after a program has been executed using **run** or **animate <speed>**.
q	Quits interactive mode.
run	Executes code that has been loaded into the program buffer.
save <filename>	Saves the current contents of the program buffer to a text file *filename*.
stack	Displays the code lines that are currently on the CobolScript internal stack.
stepoff	Turns off step mode that was set using the **stepon** command. After step mode has been turned off, the **run** command will run programs normally, without stepping.
stepon	Places CobolScript in step mode. Once in step mode, the **run** command will begin interactive execution of the loaded program. Interactive execution means that the program is executed, one line at a time, by pressing the ENTER key. As the program is interactively executed, commands such as **variables, files**, **ip**, and **stack** can be used to display current information.
variables	Displays all of the variables used by a program, and the contents of those variables. The **variables** command can be used after **run**, **animate <speed>**, or **stepon** has been used to execute all or a portion of a loaded program.

Interactive Mode Command	Description
ver	Displays version information for your CobolScript installation.

Running CobolScript from a Web Server and Browser

With proper installation and web server configuration, CobolScript programs residing in the appropriate web server directory can be initiated by (and the output displayed in) a web browser. By placing your CobolScript programs on a server and accessing them with a browser, you can create graphical, efficient applications accessible from any computer with browser software installed on it, so long as the browsing computer has visibility to the web server computer, either across a network or the internet.

For your CobolScript web applications to run correctly, you should perform the following steps:

1. Place your programs, any text files used by your programs, and the CobolScript executable in your web server's cgi-bin directory. Consult your web server documentation if you do not know where the cgi-bin directory is, or you want to modify its location.
2. On Unix servers, use the **chmod** command to change the permissions on the files that you placed in the cgi-bin directory, as necessary. Since CGI scripts usually run as user 'nobody', the permissions on these files generally must be set to allow any user to have the appropriate access to all files used by your programs. As an example, suppose a CobolScript web program reads from and writes data to a file named DATA.TXT. The file DATA.TXT must then permit both reading and writing by any user, in order for the program to run successfully. In this case, typing

   ```
   chmod 666 DATA.TXT
   ```

   at the Unix command prompt will change DATA.TXT appropriately.
3. Make certain CGI scripting is turned on and permitted by your web server software; this is necessary for CobolScript applications to run correctly. Consult your web server documentation for information on how to enable CGI scripting if it is not already enabled.

4. After you have placed CobolScript and your CobolScript programs in your cgi-bin directory, you can execute the programs on the server with your browser by placing a "?" between the cobolscript.exe and the program's filename in the browser URL. Figure 2.8 illustrates the execution of a sample timesheet program initiated from a Netscape® browser, but running on a FreeBSD® server with Apache web server software; note the address in the Location: (URL) box, that runs the program uts.cbl in the cgi-bin directory with the syntax "cobolscript.exe?uts.cbl".

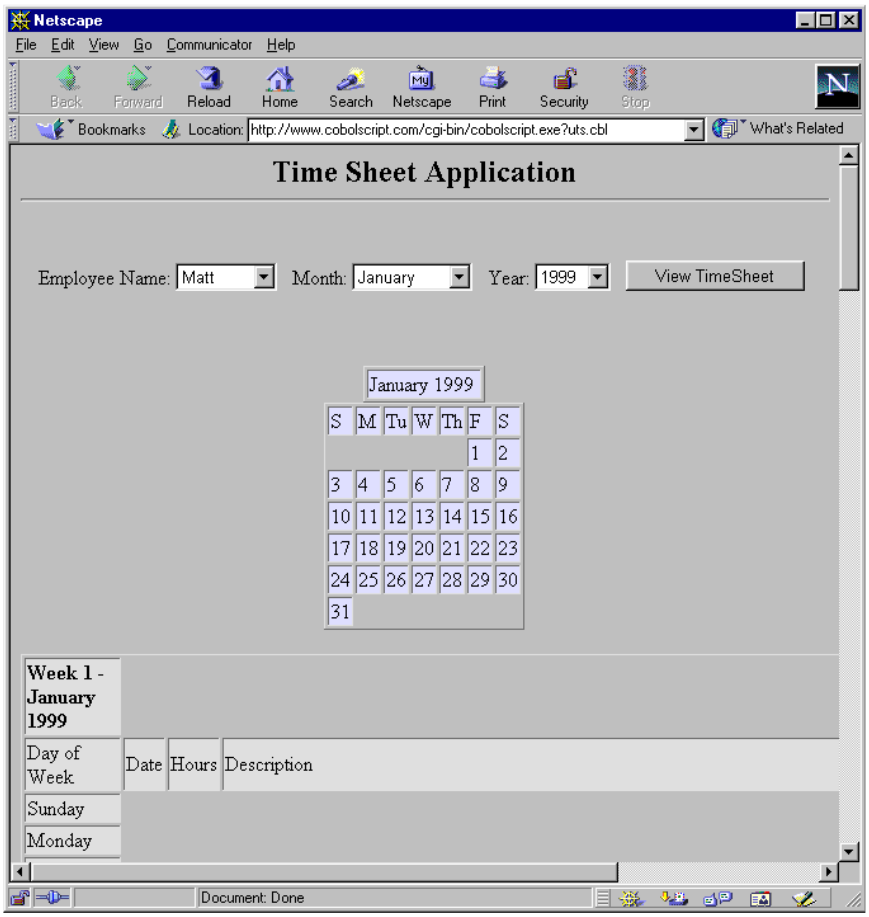


Figure 2.8 – CobolScript Web Application (Timesheet Program) Example

To generalize, any CobolScript program that has been placed, along with the CobolScript executable, in the appropriate directory on your web server's computer can be executed by using a URL of the following form:

http://<your ip address>/cgi-bin/cobolscript.exe?<program name>

You can see several more example of this on the Deskware samples web site at http://www.cobolscript.com/cgi-bin/cobolscript.exe?samples.cbl.

CobolScript Error Messages in Web Browser Mode

Besides the standard command line error messaging system explained in the **Running CobolScript from the Command Line** section in this chapter, CobolScript provides an integrated web-based error messaging system. This messaging system is unique in that CobolScript determines whether you are running a program from a web browser or the command line, and controls the display of the

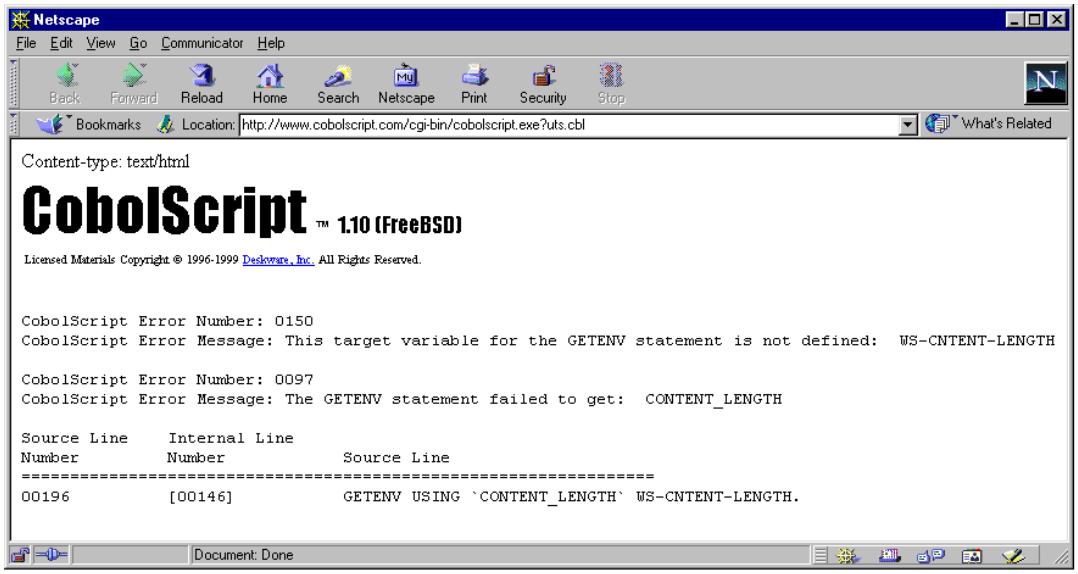


Figure 2.9 – Browser-based error message

error message accordingly. If you run a CobolScript program from you web browser and encounter an error, you will see the CobolScript Error Number and error message displayed in a consistent HTML-based format; if you run the same program from the command line and encounter the same error, the number and messages will display in a text-based format.

The web based error message in figure 2.9 illustrates this HTML-based error messaging system. In this particular example, a variable was misspelled in the GETENV statement, and was therefore undefined and caused an error.

In certain cases when running CobolScript programs from a browser, you will see a completely blank browser window, or an incomplete display of your HTML without a CobolScript error on the page. These cases can indicate errors in your HTML code as well as a CobolScript error. Check the page source from your browser to find any CobolScript error messages that are embedded in the HTML but did not successfully display. Correct the CobolScript error(s) first; if the page still fails to display properly, but there are no longer any CobolScript error messages in the page source, check your HTML syntax.

Chapter 3

CobolScript® Language Constructs

ICON KEY

➲ Important point

In CobolScript, there are several categories of constructs which form the foundation of the language. This chapter defines these constructs and their specific CobolScript syntax. Since CobolScript language constructs are not so different from the elementary components that comprise most other computer languages, you may opt to focus your attention only on those sections in this chapter that deal with material unfamiliar to you. Each CobolScript construct is unique in at least a minor fashion, however, so refer back to the appropriate section here if you are having difficulties with a particular construct.

With the exception of delimited string literals, all CobolScript alphanumeric syntax is case insensitive, meaning uppercase letters, lowercase letters, and any combination of these will work for any particular command, variable, or reserved word. This flexibility requires that you be cautious, however, when defining your variables; see the **Variables** section for more information.

The CobolScript language constructs are divided into the following categories:

- Literals and Literal Keywords
- Variables
- Data and Copybook Files
- Expressions and Conditions
- Commands
- Reserved Words
- Statements
- Sentences
- Comments

We explain each of these categories individually in the following sections.

Literals and Literal Keywords

Literals are any numbers or character strings which are meant to be taken literally by your program. Literals are perhaps best defined by what they aren't: A literal is not a variable, which has values substituted in for the variable name at the time the program is run, nor is a literal necessarily an expression, which is mathematically evaluated to arrive at a resulting value (although literals can comprise expressions). As you will see from the examples below, literals can only appear in places

within statements or variable definitions where they are used as a source for information, and never as a target, since a literal cannot change its value.

Numeric Literals

If a literal is numeric, and you want that numeric literal to be treated as a number by your program, it should not be enclosed in any offsetting quotes or string delimiters. Also, a numeric literal should not include any special formatting characters like commas or dollar signs; the only special characters allowed within a numeric literal are the negative sign (-) and the decimal point, indicated with a standard period (.). To use a numeric literal in your program, just insert the number, including any negative sign and decimal point, into your statement or VALUE clause in the appropriate position.

If you use a numeric literal in a VALUE clause, the variable being defined must also be numeric. Here are some examples of numeric literals in VALUE clauses in variable definitions:

```
1 variable_1   PIC $9,999.99 VALUE 2323.41.
1 variable_2   PIC S99,999.999 VALUE -32000.
```

If you have questions about the PIC clauses in the above variable definitions, picture clauses are explained completely in Appendix E, *CobolScript Picture Clauses*.

Here are some examples of numeric literals in code statements:

```
MOVE 5 TO variable_1.
SUBTRACT 6.23 FROM number_var_1.
MULTIPLY 2 BY -6 GIVING result_var.
COMPUTE result_var = -2.25.
```

Alphanumeric Literals

Alphanumeric literals, also known as *strings*, are any delimited character or string of characters which is to be taken literally by your program. Any character other than the string delimiting character, which is normally the accent symbol, can appear within a delimited string. See the subsection below for more information on string delimiters.

If you use an alphanumeric literal in a VALUE clause, the variable being defined must be of alphanumeric (PIC X) type. Here are some examples of alphanumeric literals in VALUE clauses in variable definitions:

```
1 variable_2   PIC XXX VALUE `123`.
1 FILLER       PIC X(n) VALUE `<BODY><HR><BR>"#1" Web Page</BODY>`.
```

If you want further explanation of the types of PIC clauses used in the above variable definitions, refer to Appendix E, *CobolScript Picture Clauses*.

Here are some examples of alphanumeric literals used in procedure statements:

```
MOVE `Y` TO variable_1.
IF condition_val = `E1qwT`
   CONTINUE
END-IF.
```

```
DISPLAY `Hello, 'Ray'.  `.
```

The CobolScript String Delimiter

The string delimiter in any language is the character that is used to signal the beginning and the end of alphanumeric literals. In most computer languages, the string delimiter is either the single or double quote, so strings enclosed in their delimiters are commonly referred to as being *quoted*. In CobolScript, however, the default string delimiter is the Gravè accent, or just plain accent (`). Since CobolScript also has command line options to permit the use of the single or double quote as the string delimiter (see the section titled **Running CobolScript from the Command Line** in Chapter 2, G*etting Started with CobolScript* for more details), we usually refer to alphanumeric literals simply as being *delimited* to avoid confusion.

The accent key is the key located in the upper left corner of North American keyboards, below the `Esc` key. Normally, both the tilde (~) and the accent (`) are on the same key. We selected the accent as the default string delimiter for CobolScript because HTML, which must be displayed from CobolScript web applications, requires the frequent use of double and single quotes; using a different character for the CobolScript string delimiter simplifies the creation of these strings. The alphanumeric literal in the following MOVE statement is standard HTML and illustrates this point well:

```
MOVE `<A HREF="/test.htm">Test Page</A>` TO url_var.
```

If you still prefer to use quotes, however, you can. Just create your program using either single or double quotes as the string delimiters, and run the program using the appropriate command line option. See the previously mentioned section in Chapter 2 for syntax information.

If you're an experienced C programmer, you may be curious about whether the backslash (\ ) has special meaning inside a CobolScript string. It doesn't. This is primarily because CobolScript strings must contain any client-side scripts that you choose to embed in your CobolScript-generated HTML. These scripting languages each may attribute special meaning to certain characters inside a string, and these special characters should not interfere with the original CobolScript string. Simply put, there is no 'escape' character, backslash or other, in CobolScript that will cause the character following it to be interpreted literally. Because of this, there is no direct way to display the current delimiter symbol from within a delimited string – a special keyword, not enclosed in delimiters, must be used instead.

To display a literal of the accent symbol from within a CobolScript program that uses the accent as the string delimiter, you must use the ACCENT keyword, as in:

```
DISPLAY ACCENT.
DISPLAY `The accent symbol: ( ` & ACCENT & ` ).`
```

The same rule applies if you are using double or single quotes as the string delimiter. When the double quote is your string delimiter, use the DOUBLEQUOTE keyword to display the symbol, as in:

```
DISPLAY DOUBLEQUOTE.
```

and when using the single quote as the string delimiter, use the SINGLEQUOTE keyword, as in:

```
DISPLAY SINGLEQUOTE.
```

Literal Keywords

Below is the complete list of literal keywords. Like ACCENT, DOUBLEQUOTE, and SINGLEQUOTE, each of these keywords represents a specific ASCII character constant.

Keyword	Symbol represented by keyword
ACCENT	`
CARRIGERETURN	{equivalent of ASCII character number 13}
CRLF	{equivalent of ASCII character number 13 + ASCII character 10; uses two bytes}
DOUBLEQUOTE	"
LINEFEED	{equivalent of ASCII character number 10}
SINGLEQUOTE	'
SPACE	{all blanks}
SPACES	{all blanks}
TAB	{equivalent of ASCII character number 9}
ZERO	**0**
ZEROS	**0**

Variables

Variables are information holders. In CobolScript, variables come in five basic forms, each of which has its own characteristics and utility. These five forms are:

- Elementary data items, which can be either numeric or alphanumeric;
- Group-level data items;
- FILLER variables, which are really a special category of elementary data item;
- REPLICA variables;
- OCCURS clause variables.

No matter what the form, a variable must first be *defined* in a program, and then, as the term *variable* implies, the variable's contents can be assigned and reassigned throughout the body of a program. In CobolScript, these value assignments are done with *VALUE clauses* and *assignment statements*. VALUE clauses are optional components of elementary data item variable definitions that establish an initial value for a variable; assignment statements are any procedure statements that modify a variable's contents.

A variable definition must follow certain rules of syntax, which are described below for each of the variable forms. A variable definition may be placed anywhere within a CobolScript program, meaning that variable definitions are not restricted to the Data Division as they are in COBOL. However, you should not define the same variable more than once within a program.

In CobolScript, variable names are not case sensitive, so WS-VAR, ws-var, and Ws-Var will all be treated internally as the same variable. For this reason, only one of these names should be defined in a program. Similarly, two variables that have the same alphanumeric name and differ only by underscore and dash separators within the variable name, such as WS-VAR and WS_VAR, will be treated interchangeably by certain CobolScript commands and should not both be defined in a single program.

The Elementary Data Item

An elementary data item (also referred to as a 'subvariable' or just 'elementary item') is any basic numeric or alphanumeric variable. An elementary data item cannot have subvariable components. The syntax of a normal elementary data item variable definition is:

```
<level-number> <variable-name> PIC <picture-clause> [VALUE <value-literal>].
```

The *level-number* is a one- or two-digit number from 1 to 99. Think of the level number as representing the outline position of a variable; the lower the level number, the higher the variable's rank in the outline, with 1 being the highest level. So long as you have defined at least one variable with a level of 1 in your program, the variables with level numbers greater than 1 will all be subvariables. This is best illustrated with an example:

```
1 text_input PIC X(40).
1 group_variable.
  2 components.
   3 component_1   PIC X(12).
   3 component_2   PIC $,999.99.
  2  val_1         PIC 99.
1 input_1          PIC X(25).
```

In the variable definitions above, text_input is both an elementary data item, because it doesn't have any subvariables beneath it, and is a level 1 variable. The variable group-variable is a group-level data item (explained in the subsequent section), which has two subvariables, components and val_1. The variable components is a group item itself, and has two subvariables, each of which are elementary items. The variable val_1 is an elementary data item, as is input_1.

The *variable-name* of an elementary data item is the name that will be used throughout the program to reference this particular variable.

The elementary data item variable's type, format, and length are all determined by the value of the *picture-clause* that immediately follows the PIC keyword. In CobolScript, all elementary item variables are assigned a fixed number of bytes according to the size specified in the picture clause, so you must allocate sufficient space for your variables when you create their picture clauses; otherwise, the variable values will be truncated and information will be lost. A picture clause can be of two basic types: numeric (PIC 9 format) or alphanumeric (PIC X format). The various picture clause formats, and their meaning, are explained fully in Appendix E, *CobolScript Picture Clauses*.

If you want to initialize the elementary data item variable to a value at the time you define it, you can include the VALUE keyword and follow it with a *value-literal* to assign to the variable. The value literal must be of a type that matches the picture type of the variable; in other words, a variable with a numeric picture clause must be assigned a numeric value literal, and a variable with an alphanumeric

picture clause must be assigned an alphanumeric literal. See the preceding section of this chapter for more information on literals.

These are some example elementary item variable definitions:

```
1 string_variable PIC X(10) VALUE `abcdefghij`.
1 input_var       PIC XX.
1 num_variable    PIC $,999.99   VALUE 679.
```

The Group-Level Data Item

A group-level data item (also referred to as a 'gldi' or just 'group item') is a hierarchical parent variable that is made up of other variables known as subvariables or component variables. Group items are similar to record variables or data structures in other programming languages; they're useful because they enable you to reference and transfer whole groups of variables by citing a single, succinct variable name. In CobolScript, group items are also used to define file records. See the **Data and Copybook Files** section of this chapter for more information on file records.

The syntax of a group-level data item variable definition is:

```
<level-number> <variable-name>.
      <subvariable-definition>.
      .
      .
      .
```

As in elementary items, the *level-number* of a gldi indicates the variable's position in the hierarchy; see the definition of level number for elementary data items for more information.

The *variable-name* of a group item is the name assigned to the variable, just as in elementary data items.

In group items, no PIC or VALUE clauses are allowed. This is because a gldi's structure is defined solely by its *subvariable-definitions*. A group-level data item's subvariables can be group items themselves, making possible multiple levels of grouping, or the subvariables can be elementary data item variables.

Below is a standard group-level data item variable definition. In this example, group_variable is the group item, and is composed of two elementary items:

```
1 group_variable.
  5  component_1   PIC XXX VALUE `mS1`.
  5  component_2   PIC $,999.99.
```

The FILLER Variable

The FILLER variable is a special type of elementary data item; it should only be used as a subvariable to a group item, because it is always given the name FILLER, and cannot be directly referenced. The syntax of a FILLER variable definition is:

```
<level-number> FILLER PIC <picture-clause> VALUE <value-literal>.
```

The *level-number* and *picture-clause* are the same as those for a normal elementary data item, except FILLER variables should never be level 1 variables (because they must be subvariables).

A VALUE clause should almost always be specified for a FILLER variable, since FILLERs generally act as constants in a program. In cases where the FILLER variable is just acting as a placeholder, a VALUE clause may not be necessary.

Once defined, FILLER variables can only be referenced and modified indirectly, through references to their parent variable. They should be used in cases where there is no need for a direct reference, such as when a component of a group item remains static throughout the program. In the example below, a FILLER variable is one of three subvariables that comprise the group item variable group_variable:

```
1 group_variable.
   5  component_1   PIC XXX VALUE `mS1`.
   5  FILLER        PIC X(n) VALUE ` has a dollar value of `.
   5  component_2   PIC $,999.99.
```

Using PIC X(n) with FILLER variables

The special picture clause PIC X(n) can (and generally should) be used with any alphanumeric FILLER variable for which you specify a VALUE clause. PIC X(n) automatically assigns a length to the FILLER variable based on the length of the VALUE clause, so that you don't have to calculate the variable length yourself when creating the picture clause. For example, in group_variable above, the FILLER variable is automatically assigned a length of 23 characters because the value clause is 23 characters long.

For more information on PIC X(n), see Appendix E, *CobolScript Picture Clauses*.

Implied PIC X(n) FILLER variables

FILLER variables using PIC X(n) can also be defined with a shorthand notation that eliminates the FILLER keyword, picture clause, and VALUE keyword. This is best illustrated with an example:

```
1 group_variable.
  5 `Enter your name here: `.
```

In group_variable above, there is a single FILLER variable, with a value of `Enter your name here: `. The above gldi is the exact equivalent of the following:

```
1 group_variable.
  5 FILLER PIC X(n) VALUE `Enter your name here: `.
```

This shorthand may only be used when the FILLER variable's value is an alphanumeric that is set off by delimiters.

REPLICA Variables

A REPLICA variable is a special type of elementary item variable that has the same name and level number as a previously defined elementary item variable, and refers to the same physical variable in memory as the originally defined variable. REPLICA variables are useful when defining multiple

group item variables that all require the same elementary item component; using a replica in these cases avoids the task of moving values back and forth between these elementary items.

REPLICA variables are defined with a level number, variable name, and the REPLICA keyword. PIC and VALUE clauses are not permitted in a REPLICA variable because they are not meaningful; this information is defined by the original variable (also called the *replica parent*), whose definition always precedes the REPLICA variable definition. Similarly, no VALUE clauses are permitted in replicas, and both the replica and the replica parent must be elementary item variables with the same level number. Here's the basic REPLICA variable syntax:

```
<level-number> variable_name REPLICA.
```

And here's a simple example of REPLICA usage:

```
1 group_variable_1.
   5  component_1   PIC XXX VALUE `mS1`.
   5  ` has a dollar value of `.
   5  component_2   PIC $,999.99 value 125.99.

1 group_variable_2.
   5 `The value in the component_1 replica variable is: `.
   5  component_1   REPLICA.

 DISPLAY group_variable_1.
 DISPLAY group_variable_2.

 MOVE `q72` TO component_1.
 DISPLAY group_variable_1.
 DISPLAY group_variable_2.
```

In the above example, the normal, full definition of component_1 occurs in the group_variable_1 group item definition; the second component_1, defined in group_variable_2, is a replica of the original component_1. Thus, component_1 inside group_variable_1 is the replica parent, and component_1 inside group_variable_2 is the replica. The output of the code above is:

```
mS1 has a dollar value of  $125.99
The value in the component_1 replica variable is: mS1
q72 has a dollar value of  $125.99
The value in the component_1 replica variable is: q72
```

The OCCURS Clause Variable

In CobolScript, the OCCURS clause variable is a special type of variable, either elementary or group item, that defines arrays of each of its subvariables. The OCCURS clause syntax excels over other types of array definition syntax when defining record arrays; this is because arrays of records fit naturally within the syntax of an OCCURS clause group item definition.

The syntax of an OCCURS clause group item variable definition is:

```
<level-number> <variable-name> OCCURS <n> TIMES.
      <elementary-item-definition> or <group-item-definition>.
      .
      .
      .
```

The syntax of an OCCURS clause elementary item variable definition is:

```
<level-number> <variable-name> OCCURS <n> TIMES PIC <picture-clause>
                                                VALUE <value-literal>.
```

An OCCURS clause variable is defined the same way as its underlying form (elementary or group item), except for the OCCURS clause. This clause is initiated by the OCCURS keyword; in the case of the OCCURS group item, it indicates that the subvariables that comprise this group are recurring. In the case of the OCCURS elementary item, it indicates that this particular variable is recurring. In either case, the number of times the OCCURS variable(s) recur is indicated by a positive (strictly greater than zero) integer value *n*, which can either be a numeric literal or a numeric variable.

When referencing an OCCURS variable, you must use an index to indicate which of the recurring variables you mean. The index must be an integer with a value from 1 to n. So, if the OCCURS variable is defined using either of these forms:

```
1 occurs_variable OCCURS 10 TIMES.
   5  component_1   PIC 99.
   5  component_2.
     10 component_2_1   PIC XX.
     10 component_2_2   PIC 99.
```

or,

```
1  component_1  OCCURS 10 TIMES PIC 99.
```

Then, component_1 is a recurring variable (along with component_2 and its subvariables in the group item example), and its index can be any number or variable with an integer value from 1 to 10, inclusive. So, to reference the third OCCURS variable of component_1 in a statement, we would use the syntax:

```
component_1(3)
```

or, alternatively:

```
component_1(integer_variable)
```

where integer_variable is an integer numeric variable that is equal to 3 at the time it is referenced. We can also use the syntax:

```
component_1(expression)
```

where expression is any valid mathematical expression that evaluates to a positive integer, such as the following expression, which again assumes a value of 3 for integer_variable:

```
component_1(((2^2)+integer_variable)%3)
```

The group item component_2 and its two subvariables, component_2_1 and component_2_2, can be referenced the same way as component_1; thus, all of the following forms are permissible:

```
component_2(3)
component_2_1(3)
component_2_1(integer_variable)
component_2_2( ((2^2)+integer_variable)%3 )
```

Specifying a VALUE clause for an elementary item that recurs initializes all OCCURS elements to the value-literal. For example, in the gldi below, component_1(1) through component_1(5) will have initial values of 05, and component_2_1(1) through component_2_1(5) will have initial values of `me`. Specifying a value clause for an OCCURS elementary data item has the same net effect, as in the second OCCURS clause definition below:

```
1 occurs_variable OCCURS 10 TIMES.
 5  component_1   PIC 99 VALUE 5.
 5  component_2.
   10 component_2_1   PIC XX VALUE `me`.
   10 component_2_2   PIC 99.
```

or,

```
1 component_1  OCCURS 10 TIMES PIC 99 VALUE 5.
```


Note that CobolScript Standard Edition only permits single-level OCCURS clauses. In other words, two-dimensional and higher arrays are not supported by the Standard Edition. This means that an OCCURS clause gldi that has any OCCURS clause subvariables is not permitted in the Standard Edition. See below for an explanation of multidimensional array usage in CobolScript Professional Edition.

Multidimensional Arrays Using CobolScript Professional

If you are programming with CobolScript Professional Edition, you can define OCCURS clause variables that contain other OCCURS clause subvariables. This type of variable is also known as a *multidimensional array* because its individual elements comprise an array that has more than one index argument, or dimension. Let's take a look at a basic multidimensional array definition using CobolScript Professional:

```
1 day_of_week OCCURS 7 TIMES.
  5 hour_of_day  OCCURS 24 TIMES.
    10 fahr_temp  PIC ---9 VALUE -300.
    10 barom_pressure PIC 99.99 VALUE 0.
```

In the definition above, 168 total instances of the fahr_temp and barom_pressure variables are created and initialized. Each elemental variable corresponds to a temperature and barometric pressure reading for a specific hour of the day on a specific day of the week. The value in a specific element is referenced using a two-argument array reference with the dimensions separated by commas, as in the following statement:

```
DISPLAY fahr_temp(1, 13).
```

This statement corresponds to displaying the temperature value for 1:00 PM on Sunday, assuming Sunday is treated as the first day of the week.

The same range of argument syntax is permissible in multidimensional arrays as in one-dimensional arrays, so that the following are all valid references, assuming var_idx1 and var_idx2 are both properly defined:

```
fahr_temp(7, var_idx2)
hour_of_day(6+1, var_idx2)
barom_pressure(var_idx1, var_idx2)
fahr_temp(var_idx1+1, var_idx2-1)
```

Additional array dimensions are declared using additional nested OCCURS clauses:

```
1 a OCCURS occurs_num TIMES.
 5 b PIC X VALUE `b`.
 5 c.
  10 d PIC 9  VALUE 1.
  10 e PIC XX VALUE `ee`.
 5 f OCCURS 2 TIMES PIC XX VALUE `ff`.
 5 g OCCURS 3 TIMES.
  10 h PIC XX VALUE `hh`.
  10 i OCCURS 4 TIMES.
    20 j PIC X VALUE `j`.
    20 k PIC X VALUE `k`.
    20 l OCCURS 2 TIMES PIC XX VALUE `ll`.
    20 m OCCURS 2 TIMES.
      30 n PIC X VALUE `n`.
```

Referencing syntax for variables with more than two dimensions is just an extension of the two dimension case, with additional commas separating the additional array dimensions:

```
MOVE `p` TO n(1,2,3,1).
DISPLAY `n(1,2,3,1) after move = ` & n(1, 1+1, occurs_num-1, 1).
```

There is no technical limit to the number of array dimensions that can be used in CobolScript Professional; however, the limit on the number of variables that may be declared in a single program creates a practical upper bound on the number of array dimensions. At any rate, careful programming will rarely warrant the use of more than three dimensions. Although exceptions may apply in certain mathematical programming cases, and in cases involving intentional denormalizing of data constructs, very large dimension arrays should generally be avoided in order to keep your programs comprehensible.

Data and Copybook Files

Like variables, data and copybook files hold information that can be used in a CobolScript program. Of course, files are external entities, and as such are independent of the program and are stored separately on disk. Files also have a total capacity that is generally only limited by your disk space,

rather than being controlled by program limitations (although there are limits on individual record sizes in data files).

Data Files

A CobolScript data file is just a special type of ASCII text file that contains data *records*. Records are a long string of data values, or *fields*. Each record is terminated with a linefeed.

Records have a specific layout, so that each record has the same number of fields, and each specific field within a record shares formatting characteristics with the field in the same position in the other records in the data file. For example, if the fifth field in a record is a numeric with the value `000311`, then the fifth fields in the other records in the file will also be six byte numerics. An example delimited data file with a delimiter of '|' and several records in it might look like this:

```
12051999|al@bbnb.net|Reynolds|Al|10 Meisenheimer Drive|Womack|MI|49332|
12051999|smith@ffdfff.com|Smith|Roy|511 Critical Pass|Boca Raton|FL|33983|
07061999|misterm@wyyyee.edu|M|Mr|302489|Rejkyavik||54663-211|
```

Data files can be either token-delimited or fixed format. In a delimited file, a single byte delimiter character of your choice is used to separate individual fields from one another, while in a fixed format file, a fixed number of bytes is assigned to each field, so there is no need for a delimiter.

To enable a particular data file to be processed by a program, you must first describe the file. This is done with the FD (File Description) statement. The FD statement has the following syntax:

```
FD <filename> RECORD IS <length> BYTES.
```

The *filename* argument is the alphanumeric literal or variable that indicates the name and path of the data file. It's best to keep your data files in the same directory as the CobolScript engine if you frequently move your code between machines with Windows file systems and ones with Unix file systems. This is because the directory symbol is different for these file systems ('\' versus '/') and your code will then require that you change this symbol every time you switch between the two platforms.

It's important that you specify the correct *length* argument, since this tells CobolScript where to end the record. A record is terminated with a carriage return-linefeed combination for Windows machines, and just a linefeed for Unix platforms; these terminating characters, however, are not included in the length argument, so that the same file can be described by the same FD statement, regardless of platform.

The length argument can be either a numeric variable or a numeric literal. In either case, it should have a positive (strictly greater than zero) integer value.

The length of a fixed width record is always equal to the sum of the lengths of the fields that comprise it; calculating the length of a delimited record is a bit more involved, but not difficult. The minimum length that you must use for a CobolScript delimited record, provided your delimiter requires one byte of storage, is always equal to the formula:

```
Sum of lengths of individual record fields + (number of fields in record)
```

In delimited files, CobolScript right-pads the records with spaces, so that each record is still the exact number of bytes specified in the length argument. This fact is relevant if you process a delimited data file created with another application: Although reading and appending to that file will work fine in CobolScript, updating existing records will not, since each record has a different size. For more on this topic, see Chapter 4, *File Processing and I/O*.

Once you've described a file, you must define a record variable with subvariables that represent each component field in the record. In CobolScript, you define record variables like any other group-level data item. You can define file record variables anywhere within a program, so long as the record definition appears prior to any file processing statements that make use of the record such as READ and WRITE. It's important to not leave any fields out of your record definition, and to define them with the proper format and length, especially if they are fixed-length format. An incorrect or incomplete record definition will cause your record subvariables to be populated with the wrong fields, and your data will be messed up, to say the least.

A FD statement for a fixed-width record, followed by the record definition for the file, might look like this:

```
1 filename_var PIC X(n) VALUE `file.dat`.
1 bytes_var    PIC 999  VALUE 100.
FD filename_var RECORD IS bytes_var BYTES.
1 record_var.
  5 rv_field_1 PIC X(50).
  5 rv_field_2 PIC X(10).
  5 rv_field_3 PIC X(40).
```

Data file manipulation is discussed in detail in Chapter 4, *File Processing and I/O*.

Copybook Files

Copybook files are external code files that can be loaded into a CobolScript program via a single statement. The contents of the copybook file are then treated as if they were part of the program. Copybooks are most commonly used to store variable definitions, especially record variable definitions, since the same data file is often used by multiple programs. Using a copybook to store a record variable definition reduces programming effort and eliminates the possibility of discrepancies in the definition across programs. Copybooks also work well for storing group-level data items that contain HTML that you want to replicate across your CobolScript CGI programs.

Copybook files are included in a program with the COPY or INCLUDE statement. These statements are special in that they can be located anywhere within a program. This allows the code in a copybook to be substituted into the program at any location, wherever the COPY or INCLUDE statement is placed.

In the following example, an INCLUDE statement is inserted into a program to include the file `testvars.cpy`, which is located in the parent directory of the CobolScript engine's directory, on a Windows machine:

```
FD test.dat RECORD IS 17 BYTES.
INCLUDE `..\testvars.cpy`.
```

Although it's possible to include a path in the INCLUDE statement like this example does, it's inadvisable if you frequently move your code between machines with Windows file systems and ones with Unix file systems. The directory symbol is different for these file systems ('\' versus '/') and your code will then require that you change this symbol every time you switch between the two platforms.

When we examine the contents of testvars.cpy, we see that this file contains a few simple variable definitions that can then be referenced by the calling program:

```
1 content_length PIC 9(5).
1 eof            PIC 9.
1 occurs_var OCCURS 5 TIMES.
  5 occurs_var_1 PIC 999.
```

Assuming that it follows the INCLUDE statement in our original program, the following MOVE is legitimate because the definition for eof is now included in the program's variables:

```
MOVE 1 TO eof.
```

For more information on the COPY and INCLUDE statements, see their respective entries in Appendix A, *Language Reference*.

Expressions and Conditions

Expressions and conditions can appear in multiple locations in a CobolScript program. Positional string reference and array arguments can be expressions; CobolScript COMPUTE statements, which assign a value to a single variable, permit the use of mathematical expressions in the assigning value; the CobolScript DISPLAY and DISPLAYLF statements allow expressions as arguments, and the expressions are then evaluated before the result is displayed; and the IF statement and all variations of the PERFORM .. UNTIL statements evaluate conditions. Below are the CobolScript rules of syntax and evaluation for expressions and conditions. See Appendix A, *Language Reference*, for the exact syntax of COMPUTE, DISPLAY, IF, and PERFORM.

Expressions

In CobolScript, an expression is any mathematical formula that has a single-value solution. An expression can consist of any other expressions, numeric literals, variables, functions, or assignment statements using mathematical operators. All variables used in an expression must be properly defined with a numeric picture clause prior to the expression's statement, because the variables' values will be substituted in prior to evaluating the expression. Functions, which are mathematical operations such as sine, cosine, present values, and the natural log, are described fully in Appendix B, *Function Reference*.

CobolScript permitted operators

Symbol	Meaning	Example	Example result
+	Add	5 + 2	7
-	Unary negative sign	-4	-4
-	Subtract	5 - 2	3
*	Multiply	2 * 2	4

Symbol	Meaning	Example	Example result
/	Divide	7 / 7	1
^	Raise to a power	2^4	16
\	Express in scientific notation	2\2	2 * 10^2 = 200
%	Modulus, or mod	10%4	2
=	Equals	1 = 3	0

Symbol	Meaning	Example	Example result
NOT =	Not equal to	1 NOT = 3	1
>	Greater than sign	18 > 1	1
		1 > 18	0
		1 > 1	0
<	Less than sign	18 < 1	0
		1 < 18	1
		1 < 1	0
>=	Greater than or equal to	18 >= 1	1
		1 >= 18	0
		1 >= 1	1
<=	Less than or equal to	18 <= 1	0
		1 <= 18	1
		1 <= 1	1
AND	Logical AND	2 AND 0	0
		5 AND 3	1
		0 AND 0	0
OR	Logical OR	1 OR 0	1
		0 OR 0	0
		3 OR 7	1
XOR	Logical exclusive OR	1 XOR 0	1
		0 XOR 0	0
		3 XOR 7	0
NOT	Logical NOT	NOT 1	0
		NOT 0	1
		NOT 9	0

Order of operations

Operations are not necessarily performed from left to right in an expression; instead, they are evaluated in an order that depends on the relative rank of the operation, so long as no parentheses are used. The order in which operations are performed in an expression, from first performed to last performed, is:

Order	Operation(s)
1	- (unary negative sign)
2	^ (power)
3	\ (scientific notation)
4	% (mod)
5	/, * (divide, multiply)
6	+, - (add, subtract)

Order	Operation(s)
7	>, <, >=, <= (greater than, less than, greater than or equals, less than or equals)
8	=, NOT = (equals, not equals)
9	NOT (logical not)
10	AND (logical and)
11	XOR (logical exclusive or)
12	OR (logical or)

Rather than memorizing the order of operations, we recommend that you always use parentheses in your expressions. This will ensure that operations are performed in the order that you wish, and will avoid confusion for anyone else who reads or maintains your code.

Example expressions

Expression	Meaning
5	The number 5.
X	The value of the variable X.
X + Y *or* X+Y	The value of the variable X plus the value of the variable Y.
X+Y + Z	The value of the variable X plus the value of the variable Y plus the value of the variable Z.
(((X+Y)/Z)%3) ^1.86 - SQRT(X)	The variable X plus the variable Y, all divided by Z, all mod'ed by 3, all raised to the power of 1.86, all minus the square root of the variable X (SQRT is a function).
3\2	3 multiplied by 10 to the power of 2, equivalent to 3 * (10^2).
ROUNDED(X*Y*Z-Q/5/4^0.34)	The variable X multiplied by the variable Y multiplied by the variable Z, all minus the value of: The variable Q divided by 5 divided by the value of: 4 to the power of 0.34. The result is passed as an argument to the ROUNDED function, and is rounded to the nearest integer.
X + SIN(PI(0)/2)	The variable X plus the sine of π/2 radians (SIN and PI are both mathematical functions).

Expression construction rules

- Any level of nesting using parentheses is permitted.

- There is a finite length of expression permitted; generally speaking, keep your expressions small enough to be easily understandable and you will avoid this limit. If you do encounter the limit, divide your expression up into multiple assignment statements.
- There is a finite length of individual token (argument not separated by spaces) permitted. Insert spaces between expression components if you encounter this limit.
- Spaces are not required between expression components if a symbol (non-word) mathematical operator is separating the components; however, you should generally use spaces when performing subtraction operations on variables with dashes or underscores in their names. To illustrate, the expression "VAR-1 minus six" can be written two different ways, but the first method is preferred:

  ⇒ (VAR-1 - 6)

  ⇒ (VAR-1-6)

  This is because if both VAR-1 and VAR-1-6 are defined variables, the meaning of the second example becomes unclear to anyone reading the code. In CobolScript, longer variable names are always substituted prior to shorter names, so that the second case above would always evaluate to the variable VAR-1-6. Even if both variables were defined, the first example would still evaluate to the quantity (VAR-1) minus 6, which is the desired result in this case.

- Alphanumeric variables or literals in expressions that are within COMPUTE statements are not allowed, even if the argument is in the context of a truth test. Thus, the statement:

  ```
  COMPUTE total = (alnum_var = `Y`).
  ```

  is illegal because it contains an alphanumeric variable (alnum_var) and an alphanumeric literal (`Y`), even though the expression would evaluate to a numeric result. To set values based on a test of alphanumerics, embed the assignment within an IF condition.

Conditions

Conditions are expression-like logic tests in IF and PERFORM .. UNTIL statements that evaluate to a numeric result. In CobolScript, conditions are less restrictive than expressions are, because conditions allow alphanumeric variables or literals to be included in tests. Like regular expressions, though, conditions must still evaluate to a single-value result. This numeric result determines whether the condition has evaluated to TRUE or FALSE; a result of exactly zero (0) is FALSE, while *any other result* is considered to be TRUE. Thus, the conditional statement below will evaluate to TRUE because the value of the condition is -40:

```
IF (4 + 6) *(-4) THEN
```

General condition rules

There are some general rules that govern conditions , no matter what form they take:

- As mentioned above, a condition will evaluate to FALSE only if its value is zero. Any other numeric result is TRUE.
- A condition must evaluate to a numeric result. Alphanumeric results are invalid.
- Any level of compound condition nesting using parentheses is permitted.

- There is a finite length of condition; generally speaking, keep your condition's component expressions small enough to be easily understandable and you will avoid this limit. If you do encounter the limit, assign the value of one or some of your expressions to a variable prior to evaluating the condition. Then, your condition can include the variable in place of the lengthy expression. If you cannot do this because you are evaluating alphanumerics, break your condition up into multiple conditions instead, and nest your IF statements.
- There is a finite length of individual token (component of condition which is not separated by spaces) permitted. Insert spaces between condition components if you encounter this limit.
- There is no support for implied subjects or implied operators in CobolScript conditions. You must completely write out your conditions. (If you're not familiar with these terms, don't worry. They are COBOL constructs that don't really have an equivalent in other computer languages.)

Condition syntax

CobolScript conditions come in two types: General logic tests, or Type I conditions, and tests of the type of value contained in an alphanumeric variable or literal, which are Type II conditions. This is the allowed syntax for both types of conditions, and rules specific to each condition type:

Type I conditions:

```
<Expression>
NOT <Expression>
<Expression> AND <Expression>
<Expression> OR  <Expression>
<Expression> XOR <Expression>
<Expression> [IS] [NOT]  =  <Expression>
<Expression> [IS] [NOT] EQUAL [TO]     <Expression>
<Expression> [IS] [NOT] >              <Expression>
<Expression> [IS] [NOT] GREATER [THAN] <Expression>
<Expression> [IS] [NOT] <              <Expression>
<Expression> [IS] [NOT] LESS [THAN]    <Expression>
<Expression> [IS] [NOT] >=             <Expression>
<Expression> [IS] [NOT] <=             <Expression>
```

Rules specific to Type I conditions:

- All Type I conditions may have numeric literals, numeric variables, alphanumeric variables, or string literals in their component expressions.
- Alphanumeric comparisons of letters assigns a greater value to letters that come later in the English alphabet. Therefore:

  ```
  `Z` > `A` evaluates to TRUE;
  `A` = ` ` evaluates to FALSE.
  ```

- Comparison of alphanumeric values to numeric values is permitted, but will default to an alphanumeric to alphanumeric comparison. Thus, the following condition and others like it will evaluate to TRUE:

  ```
  `9` = 9
  ```

Type II conditions:

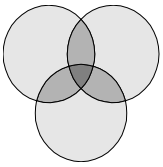
```
<Alphanumeric-val> [IS] [NOT] NUMERIC
<Alphanumeric-val> [IS] [NOT] ALPHABETIC
```

Rules specific to Type II conditions:

- Type II conditions are tests to determine whether the characters contained within an alphanumeric variable or literal are NUMERIC or ALPHABETIC.
- A NUMERIC value is any valid number, including any negative sign and decimal point. NUMERIC values may *not* include spaces; a value such as `5 ` will not be considered numeric.
- An ALPHABETIC value is any value that falls within the ranges A-Z and a-z, or is a space.
- All Type II conditions may operate only on alphanumeric variables or string literals.

Commands

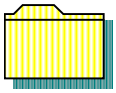
A command is the reserved word or words that form the foundation of a single procedural statement. In this section, we divide the commands into categories that can help give you a basic idea of what CobolScript commands can be used for. Refer to Appendix A, *Language Reference*, for detailed syntax rules governing each command as it is used in a complete statement.



General Program Control Commands

This group of commands is used to direct program flow, populate variables, and include code modules from external files in a program. Check the Language Reference for a command to determine its CobolScript syntax and its full capability.

ACCEPT	DISPLAY	INCLUDE	PERFORM..VARYING
ADD	DISPLAYLF	INITIALIZE	STOP RUN
COMPUTE	DIVIDE	MOVE	SUBTRACT
CONTINUE	GOBACK	MULTIPLY	UPPER
COPY	IF	PERFORM	LOWER
TRIM	LTRIM	RTRIM	TOK
GETCMDLINE	CREATESHMPOOL	PUTSHMPOOL	GETSHMPOOL
DETACHSHMPOOL			

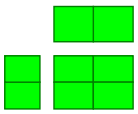


File Processing Commands

These commands execute file input and output operations on normal text files. Files in fixed width and delimited formats can be read into normal group-level data items, and normal group-level data items can be populated and then written to delimited or fixed-width files. Note that these

commands will only operate on ASCII files; no proprietary data formats are supported in CobolScript.

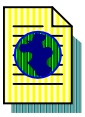
```
CLOSE           POSITION        REWRITE
OPEN            READ            WRITE

READBLOCK       WRITEBLOCK
```



LinkMaker™ Database Interactivity Commands

This group of CobolScript Professional Edition commands can be used to establish a connection with an external database, and directly embed SQL (Structured Query Language) in your programs to interact with that database. See Appendix H, *CobolScript Professional Edition Embedded SQL*, for more information on interacting with a database in your programs and on the general syntax of embedded SQL.

```
CLOSEDB         EXEC SQL        OPENDB
```



Web Processing Commands

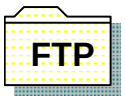
This group of commands can be used to simplify CGI programming and interaction with a web server. The ACCEPT DATA FROM WEBPAGE command gets CGI data that has been passed to it via the POST method from HTML forms; DISPLAYFILE enables binary files to be sent to the web visitor, while DISPLAYASCIIFILE sends ASCII text files; GETENV gets information about the web server; and GETWEBPAGE gets the content of a web page at a specified address.

```
ACCEPT DATA FROM    DISPLAYASCIIFILE   DISPLAYFILE   GETENV     GETWEBPAGE
WEBPAGE
```

Basic Email Commands

Simple emails may be sent and received using these commands. You must have an email (POP) account in order to use GETMAIL and GETMAILCOUNT. You must have access and permission to use an SMTP server to utilize SENDMAIL. A specific set of error-trapping variables is mandatory when using these commands; these variables can be used to redirect program flow when errors are encountered in the mail transfer process.

```
GETMAIL       GETMAILCOUNT    SENDMAIL      GETMAILSIZE    DELETEMAIL
```



FTP Commands

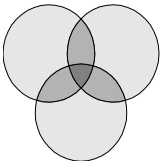
CobolScript provides standard FTP commands, so that you don't have to access the command shell in order to invoke and conduct file transfers. You can use these commands to send and receive files from within your CobolScript applications. A specific set of error-trapping variables is mandatory when using these commands; these variables can be used to redirect program flow when errors are encountered in the file transfer process.

FTPASCII	FTPCD	FTPCONNECT	FTPPUT
FTPBINARY	FTPCLOSE	FTPGET	

TCP/IP Commands

This group of commands provides the means to do TCP/IP socket programming using CobolScript. Socket programming is useful for building data interfaces over a network, and for other types of network communication tasks. A specific set of error-trapping variables is mandatory when using these commands; these variables can be used to redirect program flow when errors are encountered with a particular command.

ACCEPTFROMSOCKET	CONNECTTOSOCKET	GETHOSTNAME	RECEIVESOCKET
BINDSOCKET	CREATESOCKET	GETTIMEFROMSERVER	SENDSOCKET
CLOSESOCKET	GETHOSTBYNAME	LISTENTOSOCKET	SHUTDOWNSOCKET



Unix Shell-style Commands

These commands either mimic a Unix shell command (BANNER and CALENDAR), provide a unique twist on a shell command (GETBANNER and GETCALENDAR), or allow interaction with the host environment (CALL).

BANNER	CALL	GETCALENDAR
CALENDAR	GETBANNER	

Dynamic Processing Command

This command enables dynamic execution of CobolScript statements that are held within variables. This allows statements to be created 'on the fly' and is a basic construct of AI programming.

EXECUTE

CobolScript Reserved Words

This is a list of the reserved words in CobolScript, which includes commands, keywords, special division and section words, and words reserved for future use in later releases of the CobolScript engine. Not all words listed here necessarily have meaning to the current version of the CobolScript engine, but you should not use any of these exact words as variable or module names. This list does not include CobolScript function names, but you should also avoid naming any variables with the same name as any function. The complete list of functions is in Appendix B, *Function Reference*.

ACCENT	ELSE	IF	SENDSOCKET
ACCEPT	ELSIF	INTO	SENTENCE
ACCEPTFROMSOCKET	END	IS	SET
ADD	ENDIF	INCLUDE	SHUTDOWNSOCKET
ALPHABETIC	END-EXEC	INITIALIZE	SINGLEQUOTE
AND	END-IF	LENGTH	SLEEP
AT	END-PERFORM	LESS	SOURCE
AUTHOR	ENVIRONMENT	LINEFEED	SPACE
BANNER	EQUAL	LISTENTOSOCKET	SPACES
BINDSOCKET	EQUALS	MOVE	SQL
BY	EVALUATE	MULTIPLY	STOP
BYTES	EXEC	NEXT	SUBTRACT
CALENDAR	EXECUTE	NOT	TAB
CALL	FD	NUMERIC	THAN
CARRIAGERETURN	FILE	OBJECT	THEN
CLOSE	FILLER	OCCURS	TIME
CLOSEDB	FROM	OFFSET	TO
CLOSESOCKET	FTPASCII	OPEN	STOP
COMPUTE	FTPBINARY	OPENDB	SUBTRACT
COMPUTER	FTPCD	OR	TAB
CONFIGURATION	FTPCLOSE	PERFORM	THAN
CONNECTTOSOCKET	FTPCONNECT	PIC	THEN
CONTINUE	FTPGET	POSITION	TIME
COPY	FTPPUT	PROCEDURE	TO
CREATESOCKET	GETBANNER	PROGRAM-ID	UNTIL
CRLF	GETCALENDAR	READ	UPDATING
DATA	GETENV	READING	USING
DATE	GETHOSTBYNAME	RECEIVESOCKET	VALUE
DAY	GETHOSTNAME	RECORD	VARYING
DAY-OF-WEEK	GETMAIL	RELATIVE	WEBPAGE
DELIMITED	GETMAILOUNT	REMAINDER	WITH
DISPLAY	GETTIMEFROMSERVER	REPLICA	WORKING-STORAGE
DISPLAYASCIIFILE	GETWEBPAGE	REWRITE	WRITE
DISPLAYFILE	GIVING	ROUNDED	WRITING
DISPLAYLF	GOBACK	RUN	XOR
DIVIDE	GREATER	SECTION	ZERO
DIVISION	IDENTIFICATION	SENDMAIL	ZEROS
DOUBLEQUOTE	TRIM	LTRIM	RTRIM
TOK	GETCMDLINE	CREATESHMPOOL	PUTSHMPOOL

Statements

A statement joins a CobolScript command with arguments and other keywords to form a single, distinct operation. You can also think of a statement as being a 'step' in a program, since the CobolScript engine executes code in a statement-by-statement manner. Sometimes a statement is just a single-word command without arguments, as in the following two cases:

```
FTPASCII
CONTINUE
```

Normally, however, statements are composed of commands, arguments, and any additional keywords that are required to complete the statement, as in:

```
MOVE source_var TO target_var
COMPUTE target_var = Y + 1
DIVIDE 10 BY 3 GIVING div_result REMAINDER remain_result
GETENV USING `CONTENT-LENGTH` content_variable
```

All statements, like sentences, must begin after column 7 (the seventh character counting from the left-hand side of your text program file), meaning that the leftmost character in a statement should be in column 8 or higher.

You should indent statements that are nested within conditionals with a consistent offset for each successive level of nesting to make your code more legible. Appropriate indentation looks like this:

```
MOVE 20 to x.
PERFORM UNTIL (x < 2)
   COMPUTE target_var = SQRT(x)
   IF target_var < SQRT(2) THEN
      DISPLAY `x is less than 2`
   ELSE
      IF target_var > (SQRT(4)+1) THEN
         DISPLAY `x is greater than 9`
      END-IF
   END-IF
   MOVE target_var TO x
END-PERFORM.
```

A statement can be spread across multiple lines of your program if you wish, so long as all individual arguments and keywords within the statement remain intact. A statement should not, however, begin on the same line as a previous statement. The following lines, for example, are valid CobolScript code:

```
IF truth_test_var
   COMPUTE
   target-var = SQRT (x)
                + 1
   IF target-var
      < SQRT (2)
      DISPLAY `X is less than 2`
   ELSE
      DISPLAY `X > 2`
   END-IF
END-IF.
```

The following is not valid CobolScript code, since more than one statement is on a single line:

```
IF truth_test_var COMPUTE target_var = (6 + 2) END-IF.
```

You should be able to see by now that statements are really just a combination of the program elements previously discussed in this chapter, like commands, variables, expressions, conditions, and literals, in a way that makes sense to the CobolScript engine. For the exact syntax of each command's respective statement, see Appendix A, *Language Reference*.

Sentences

A program sentence is any phrase, statement, or group of statements in a program that is terminated with a period.

All sentences must begin after column 7 (the seventh character counting from the left-hand side of your text program file), meaning that the leftmost character in a statement should be in column 8 or higher.

In CobolScript, each of the following items constitutes a discrete and complete sentence, and therefore requires a period to terminate it:

- All Division and Section titles, as in 'PROCEDURE DIVISION.' and 'WORKING-STORAGE SECTION.'; see Appendix F, *CobolScript Basic Program Structure*, for more information on Divisions and Sections;
- The 'PROGRAM-ID.' and 'AUTHOR.' keywords in the Identification Division are each complete sentences on their own. Also, the argument to each of these keywords is a complete sentence;
- The 'SOURCE COMPUTER.' and 'OBJECT COMPUTER.' phrases in the Environment Division are each complete sentences on their own. The argument to each of these phrases is a complete sentence as well;
- All complete FD (File Description) entries;
- All variable definitions, whether group-level data item or elementary data item;
- Module (code paragraph) names;
- All complete statements that are not between PERFORM..END-PERFORM (an in-line perform) or IF..END-IF.

- If a statement is nested within an in-line perform or conditional, periods must *not* be used. The sentence in these cases terminates with the 'END-PERFORM.' or the 'END-IF.' keywords. If there are multiple levels of nesting, only the outermost level should be terminated with a period, as in the example that we used previously to demonstrate proper indentation:

```
MOVE 20 to x.
PERFORM UNTIL (x < 2)
   COMPUTE target_var = SQRT(x)
   IF target_var < SQRT(2)
      DISPLAY `x is less than 2`
   ELSE
      IF target_var > (SQRT(4)+1)
         DISPLAY `x is greater than 9`
      END-IF
   END-IF
   MOVE target_var TO x
END-PERFORM.
```

Comments

Comments are text that has no effect on your program's execution. Comments must begin with an asterisk (*) in column 7 (the seventh character counting from the left-hand side of your program file). Therefore, any line in a program that has an asterisk in column 7 will be ignored by the CobolScript engine, no matter what other text is on that line.

Well-placed, meaningful comments are critical to the readability and overall worth of your program. Explaining difficult-to-understand or non-intuitive code with a good comment will ultimately save you and anyone who edits your code a large amount of time.

Chapter

4

File Processing and I/O

Accessing and manipulating disk-resident data are tasks that must be performed by any application that has long-term information storage requirements. Almost all business applications utilize or manipulate external information in some form, and many scientific programs also have data input and output, so any good programming language must incorporate commands to enable the processing of data that is external to the program.

ICON KEY

➲ Important point

All native CobolScript data processing is done with ASCII text files, commonly referred to as *flat files*; this flat file processing is the primary focus of this chapter. CobolScript will correctly process data files that are either fixed field width or single-character delimited. If the data in the file is delimited, the parsing of the fields is handled internally by CobolScript.

The data records in CobolScript data files are stored sequentially, meaning one after another. Sequential organization is the most straightforward approach to organizing records within a file; the operations that can be performed on such a file are necessarily basic, and in CobolScript, input and output commands are restricted to entire-file operations (OPEN and CLOSE), entire-record operations (READ, WRITE, REWRITE), and an operation that moves the file pointer (POSITION). Nevertheless, if you have previously only dealt with relational database access methods to retrieve or modify data, you should pay special attention to this chapter, since data access methods such as direct SQL calls are strictly a CobolScript Professional Edition feature and are not available from within CobolScript Standard Edition.

It is, however, possible for CobolScript Standard Edition to interact with a relational database, if the RDBMS (relational database management system) supports stored procedures, these procedures can be called from the system prompt, and the RDBMS is able to direct the output from stored procedure calls to flat files. Our interaction technique, which uses a combination of stored procedure calls and intermediate flat files, is described in the last section of this chapter. Since your actual technique will vary depending on the relational database that you use and any firewall that may exist on your network, the information in this section is presented at a more conceptual level than the other sections in the chapter.

If you are programming with CobolScript Professional Edition, and you want to directly interact with a relational database using CobolScript LinkMaker™'s embedded SQL capability, refer to Appendixes G and H for instructions on configuring and using LinkMaker™.

Describing Files and Defining Data Records

Before any processing can be done on a data file, you must first describe it using an FD statement, and you must create a record variable that defines the individual fields within each data record. See the **Data and Copybook Files** section of Chapter 3 for more details on describing a file and defining a data record.

Opening Files

Before you can begin reading data from a file or writing data to a file, you must first *open* the file. Opening a file lets the operating system know that you intend to perform an input or output operation on that file, and prepares the file for subsequent operations. You can open a file in CobolScript for *reading*, *writing*, *updating*, or *appending*.

If you open a file for writing and the file already exists, its contents will be destroyed and a new file created in its place. Opening a file for reading, updating, or appending, however, will not destroy the file's contents.

The DELIMITED WITH clause can be added to an OPEN statement to indicate that a data file is delimited, meaning that fields are separated with a single-character delimiter that is specified after the WITH keyword. The absence of the DELIMITED WITH phrase indicates that the data file has fixed width fields, which will be separated based on the individual field sizes in the record definition.

Below are some examples of each variation of the OPEN statement, with and without the DELIMITED WITH clause:

```
OPEN test_file FOR READING.
OPEN `test.dat` FOR READING DELIMITED WITH `|`.
OPEN `test_file FOR WRITING.
OPEN test_file FOR WRITING DELIMITED WITH `,`.
OPEN `test.dat` FOR APPENDING.
OPEN test_file FOR APPENDING DELIMITED WITH `,`.
```


If you're working in a Unix environment, you must have the appropriate permissions set for your data files; specifically, read as well as write permissions must be set on all data files for all file processing options. Even files that are only opened for reading must have Unix write permissions set, because early versions of CobolScript used OPEN FOR READING to update records as well as to read them; to be backward compatible, current versions of CobolScript still support this format.

Closing Files

After you have finished working with a file, you must close it. Closing a file releases the file descriptor to the operating system; failing to close a file will cause the file to be locked and appear unavailable to other applications. Here is an example of the CLOSE statement:

```
CLOSE `test.dat`.
```

In the following CobolScript program, we simply open and close a file. Since it is opened for writing, the file will be created if it does not already exist, or overwritten if it does already exist.

```
1 io_file PIC X(n) value `IO.DAT`.
FD io_file RECORD IS 100 BYTES.

OPEN io_file FOR WRITING.
CLOSE io_file.
```

Reading Records From Files

The READ statement reads one data record from the data file and loads it into the target record variable. A single READ will read data until it reaches a line terminator, at which point it stops. The line terminator is the ASCII character or character combination that is used by your operating system to indicate the end of a line, usually either the carriage return or carriage return and linefeed characters in combination. The line terminator is not included in the record data.

The AT END clause of the READ statement is an error-trapping routine that recognizes when the end-of-file marker has been reached, and executes a specific statement when this condition is met. We have chosen to use a MOVE statement in this example; any simple one-line statement, such as DISPLAY or COMPUTE, could be substituted for the MOVE. The clause should be used in most cases; if the AT END clause is not specified, reaching the end of a data file will cause a CobolScript error.

Once a data record has been read and the target record variable populated, the component fields of the record variable can be used like any other variable. Below is some example code that utilizes the READ statement:

```
1 test_file PIC X(n) VALUE `TEST.DAT`.
FD test_file RECORD IS 100 BYTES.
1 input_record.
  5 ir_component_1 PIC X(50).
  5 ir_component_2 PIC X(50).
1 eof PIC 9 VALUE 0.

OPEN test_file FOR READING.

PERFORM UNTIL EOF
   READ test_file INTO input_record
     AT END MOVE 1 TO eof
   DISPLAY `Record component 1 is: ` & ir_component_1
END-PERFORM.
CLOSE test_file.
```

Overwriting a File

To overwrite a file, just open it for writing and write the new output to the file using the WRITE statement. Writing will put data from a source literal or variable into a single record in the file. In

this example, the fields comprising RECORD-VARIABLE are assumed to have already been populated:

```
OPEN test_file FOR WRITING DELIMITED WITH `|`.
WRITE record_variable TO test_file.
CLOSE test_file.
```

Appending New Records to an Existing File

To append records to the end of an existing file, open the file for appending and write each record to the file using the WRITE statement. Each WRITE statement will add the source record to the file as the last sequential data record. Here's the code for several appends to a delimited data file:

```
1 test_file PIC X(n) VALUE `test.dat`.
1 bytes_num PIC 99 VALUE 10.
FD test_file record is bytes_num bytes.
OPEN test_file FOR APPENDING DELIMITED WITH `,`.
WRITE `12345` TO test_file.
WRITE `1234`  TO test_file.
WRITE `123`   TO test_file.
CLOSE test_file.
```

The following output (highlighted in gray) will be written to the file test.dat:

```
12345,    `
1234,     `
123,      `
```

Each of the three records above is made up of three components: the source literal from the WRITE statement that created that record, followed by the comma delimiter, and then followed by enough spaces to make the total length of the record equal to ten characters. Note that even when files are opened as delimited files, CobolScript still right-pads the record with spaces until it is the total length declared in the FD statement (in this case, ten bytes). This padding is an intentional feature of CobolScript, because it simplifies the task of individually updating delimited data records. This also has relevance if you intend to update delimited data records created outside of CobolScript; see the next section on updating records for more information.

If the DELIMITED WITH option is absent from our code block, as in the following:

```
OPEN test_file FOR APPENDING.
```

Then, assuming that the FD statement and everything else in our original block of code does not change, the following output will be written to test.dat:

```
12345     `
1234      `
123       `
```

Now let's look at a slightly more complex case with a record variable that is made up of two fields. First, we'll describe the file and define the record variable:

```
1 test_file PIC X(n) VALUE `test.dat`.
1 bytes_num PIC 99 VALUE 9.
FD test_file record is bytes_num bytes.
1  record_var.
  5  field_1  PIC X(4).
  5  field_2  PIC X(5).
```

Next, we'll open the file and write some records. Note that this is a fixed width file, because there is no DELIMITED WITH clause in our OPEN statement:

```
OPEN `test.dat` FOR APPENDING.
MOVE `1` TO field_1.
MOVE `test` TO field_2.
WRITE record_var TO test_file.
MOVE `test` TO field_1.
MOVE `1` TO field_2.
WRITE record_var TO test_file.
CLOSE test_file.
```

The code above would produce the following output in the file test.dat:

```
1   test
test1   `
```

Note that each field inside a fixed width file has, not surprisingly, a fixed width. Therefore, the second field in the above example always begins in the fifth character of the record, regardless of the size of the first field.

Now let's take a look at what happens if we append delimited records instead of fixed width ones. We'll first modify the original OPEN statement to handle comma-delimited data:

```
OPEN test_file FOR APPENDING DELIMITED WITH `,`.
```

Our record should be two bytes larger than the fixed width record to account for the two comma delimiters that will be in each record, so we must also modify the VALUE clause in our bytes_num variable declaration:

```
1 bytes_num PIC 99 VALUE 11.
```

We could also have changed our bytes_num value with a MOVE statement, so long as it preceded our FD. Either way, with the two above modifications, our code would write the following to test.dat:

```
1,test,    `
```

```
test,1,
```

You can see that, unlike the fixed width file, the starting position of each individual field within a delimited record varies.

Writing to a File by Updating Existing Records

In certain situations, you will probably want to update a record that already exists in a data file without appending an additional record to the file. To update a record in a data file, you should first open the file for update using the UPDATING keyword, as in:

```
OPEN test_file FOR UPDATING.
```

Next, you should perform reads until you have read the record that you wish to update. Then, using the REWRITE statement, you can overwrite the old record, as in the following:

```
REWRITE record_variable TO test_file.
```

Here's some code that demonstrates this technique more completely:

```
1 eof          PIC 9 VALUE 0.
1 rec_found    PIC 9 VALUE 0.
1 rec_position PIC 999999.

1 test_file PIC X(n) VALUE `TEST.DAT`.
FD test_file record is 9 bytes.
1  record_var.
  5  field_1  PIC X(4).
  5  field_2  PIC X(5).

1 customer_of_interest PIC X(n) VALUE `Dave`.
1 new_field_2_val PIC X(n) VALUE `Davie`.

OPEN test_file FOR UPDATING.
PERFORM VARYING rec_position FROM 1 BY 1 UNTIL eof OR rec_found
   READ test_file INTO record_var
     AT END MOVE 1 TO eof
   IF field_1 = customer_of_interest
      MOVE 1 TO rec_found
      MOVE new_field_2_val TO field_2
      REWRITE record_var TO test_file
   END-IF
END-PERFORM.
CLOSE test_file.

IF eof
   DISPLAY `Customer record of interest was not found.`
END-IF.
```

Because CobolScript right-pads delimited records with spaces, each record is the exact number of bytes specified in the length argument to the initial FD statement. This allows any CobolScript data record, whether fixed format or delimited, to be updated in a simple and efficient manner with a simple record overlay, and without requiring any complex file reorganization for each update. However, if you process a delimited data file created with another application such as a Microsoft Excel® CSV (comma-separated values) file, CobolScript updates to this file will usually not work properly, since each record in the file will have a different byte length (reads and appends to the unmodified file will work correctly, however). The data must be copied to a different file via a CobolScript program before records can be individually updated. Here's an example of a program that does this (available in the sample program RECCOPY.CBL):

```
1 input_file PIC X(n) value `INPUT.CSV`.
FD input_file RECORD IS 100 BYTES.
1  input_record.
 5  ir_input_1  PIC X(33).
 5  ir_input_2  PIC X(32).
 5  ir_input_3  PIC X(30).
 5  ir_input_4  PIC X.

1 output_file PIC X(n) value `OUTPUT.CSV`.
FD output_file RECORD IS 100 BYTES.
1  eof   PIC 9 VALUE 0.

OPEN input_file  FOR READING DELIMITED WITH `,`.
OPEN output_file FOR WRITING DELIMITED WITH `,`.

PERFORM UNTIL eof
   READ input_file INTO input_record AT END MOVE 1 TO eof
   WRITE input_record TO output_file
END-PERFORM.

CLOSE input_file.
CLOSE output_file.
GOBACK.
```

Relative and Absolute File Positioning

If you regularly process a large number of records in flat files, you're probably aware of the time-consuming nature of sequential searches. As your file sizes increase, sequential search times increase by a proportional amount; if file sizes grow unchecked, search times will eventually become unacceptably long. In fact, this is perhaps the most critical limitation of flat file databases, and it is what prompts many organizations to opt instead for relational databases, more so than data granularity, manageability, or other considerations.

In CobolScript, flat file search times can be reduced by using the POSITION statement. This statement positions the file pointer at the beginning of a particular record within a text data file in a single step. If a data file uses a sequential numeric value as the record key value, a record within the file can be randomly (directly) accessed given that key value.

For COBOL developers, the POSITION statement functionality is similar to relative file processing.

POSITION works with standard text data files. The POSITION statement has two forms:

```
POSITION data_file AT RECORD record_number.
POSITION data_file RELATIVE OFFSET number_of_records.
```

The record_number value in the AT RECORD clause must be a positive integer in the range:

(1 <= record_number <= total number of records in file)

The record_number value (and hence the number of records in your data file) cannot exceed 2,147,483,647.

The number_of_records value used with the RELATIVE OFFSET clause must be an integer. This value indicates the number of records, counting from the current record, that the file pointer should be moved. Thus, a value of 1 will shift the file pointer one record forward in the data file; a value of –1 will shift the file pointer one record back. The number_of_records value must fall within the absolute range:

(-2,147,483,647 <= number_of_records <= 2,147,483,647)

Furthermore, a number_of_records value that causes the file pointer to be positioned before the beginning of the data file or after the end of the data file will cause a CobolScript error.

When using the POSITION statement, the number of bytes specified in the BYTES clause of the FD statement for your file must exactly match the number of bytes in the data file record; this value is used to reposition the file pointer, and a BYTES value that is larger or smaller than the actual data record size will cause the file pointer to be incorrectly positioned.

The following POSITION example uses the AT RECORD clause to access a particular record based on a sequential key value. The record is then read and displayed. After this, the file pointer is repositioned to the record prior to the record first read by using the RELATIVE OFFSET clause of POSITION:

```
1   filename_var  PIC X(n) VALUE `datafile.txt`.
1   bytes_num     PIC 99 VALUE 50.
FD  filename_var  RECORD IS bytes_num BYTES.

1   record_variable.
 5  order_nbr     PIC 99999.
 5  data_var      PIC X(45).

1   key_val       PIC 99999 VALUE 24331.

OPEN filename_var FOR READING.

POSITION filename_var AT RECORD key_val.
READ filename_var INTO record_variable.
```

```
IF order_nbr = key_val
   DISPLAY `For order number ` & order_nbr & `, data = ` & data_var
ELSE
   DISPLAY `Problem with order_nbr values in data file; check file.`
END-IF.

POSITION filename_var RELATIVE OFFSET -2.
READ filename_var INTO record_variable.
IF order_nbr = (key_val-1)
   DISPLAY `For order number ` & order_nbr & `, data = ` & data_var
ELSE
   DISPLAY `Problem with order_nbr values in data file; check file.`
END-IF.
CLOSE filename_var.
STOP RUN.
```

Relational Database Interaction with CobolScript Standard Edition

CobolScript Standard Edition can interact with a relational database if the database supports batch interaction from the system prompt, and if the database is able to direct the output from these batch interactions to ASCII text files. Ideally, the database will also support stored procedures. For table inserts, a batch row-loading utility such as Oracle's SQLLoader® will simplify the job.

We've devised a technique for database interaction with CobolScript Standard which we describe further below, but it may not work with your system since every database product is different.

Instead, we recommend you use the LinkMaker™ feature of CobolScript Professional Edition to embed SQL calls directly into your CobolScript code. If you have CobolScript Professional, read Appendixes G and H for further information on configuring LinkMaker™ and embedding SQL directly in your programs.

Note that network security configurations and firewalls may restrict your access to your database across your network. Even if you have complete access to your database, if you are using your CobolScript engine as a server-side language to complement your web server, you should be careful about which pieces of your database are made visible to the internet through SQL or stored procedure calls, especially if your database has sensitive data in it.

Regarding database security and information protection, in general, these are complicated topics beyond the scope of this manual. In larger organizations, network and database administration staff should normally be sought out and included in the decision-making process whenever there is the risk, however slight, of revealing sensitive information to the outside world. Most network administrators will appreciate it if you approach them prior to attempting to implement your idea.

We'll look at the three main SQL table interactions here (*select*, *insert*, and *update*). We exclude *delete* because in most production database cases, deletes are best handled by first updating a table row as 'to be deleted', and then deleting all such rows later in a batch stored procedure. Our explanations assume that you are already familiar with SQL and your particular relational database software. You should also have an understanding of how to write shell scripts for your operating system.

The Unix shell scripts that are included in this section are meant only as conceptual guidelines for your development; the database login portions of these scripts won't directly work with any one relational database product without at least minor modification.

Selects (Queries)

Select statements come in two forms, from a CobolScript perspective: Those that have static SQL, and those that require input from a CobolScript program.

Static Selects

Static selects are table queries that don't require any external parameters. It is just the SQL statement that remains static in a static query; the query results can change, even if the database remains unchanged between queries. This is because time constraints can be included in a static query, as in the following SQL statement:

```
SELECT customer_name
FROM   customer_table
WHERE  last_updated_datetime > (NOW – 1)
```

Assuming that the database is capable of converting the expression 'NOW – 1' into the datetime equivalent of 24 hours prior to now, there is no need for this query to incorporate external inputs. A Unix shell script that directs the output of this static query to a text file would look something like the script below:

```
#!/bin/ksh
sqllogin 'userid/passwd' <<EOF >queryresult.dat
   SET HEADING OFF
   SET ECHO OFF
   SET BREAK OFF
   WHENEVER SQLERROR pkg_output.screen_write('Database error'|SQLERROR)

   SELECT customer_name
   FROM   customer_table
   WHERE  last_updated_datetime > (NOW – 1)
   EOF
```

Two different approaches can be used to gather the result set from a static query inside a CobolScript program:

- The first approach is to run the query script in batch mode (on a daily basis, for instance) outside of the CobolScript program. Then, the CobolScript program only needs to open the data file and process the data. This approach puts the least strain on the database and on

your system, and returns a query result in the quickest time. The drawback to this method is that the data is not current at the time the CobolScript program is executed.

- Alternatively, you can call the shell script from within a CobolScript program using the CALL statement, and then open and read the resulting data from the shell script's output file using normal file processing methods. Here's some code that does this, along with a minor bit of code that takes advantage of the error trapping included in the above shell script. Assume the shell script above is named query.sh, and is in the same directory as our CobolScript engine:

```
CALL `query.sh >error.txt`.

OPEN `error.txt` FOR READING.
READ `error.txt` INTO ERROR-REC AT END MOVE `Y` TO WS-EOF.
CLOSE `error.txt`.
MOVE `N` TO WS-EOF.

IF ERROR-REC(1:14) = `Database error` THEN
   DISPLAY ERROR-REC
ELSE
   OPEN `queryresult.dat` FOR READING
   PERFORM UNTIL WS-EOF = `Y`
      READ `queryresult.dat` INTO QUERY-REC AT END MOVE `Y` TO WS-EOF
      DISPLAY QUERY-REC
   END-PERFORM
   CLOSE `queryresult.dat`
END-IF.
STOP RUN.
```

The results returned by this approach are essentially real-time. The drawback to this type of query is that it accesses the database every time this program is run.

Dynamic Selects

Dynamic selects are table queries that require external parameters, as in the following SQL statement:

```
SELECT customer_name
FROM   order_table
WHERE  customer_id = $customer_id_var
AND    order_number > $order_number_var
```

Here, the fields $customer_id_var (the value assigned to the shell script variable customer_id_var) and $order_number_var are passed in to the query from an external source (in this case, the shell script).

Here's our new shell script to handle the above query:

```
#!/bin/ksh
customer_id_var=$1
order_number_var=$2
```

```
sqllogin 'userid/passwd' <<EOF >queryresult.dat
   SET HEADING OFF
   SET ECHO OFF
   SET BREAK OFF
   WHENEVER SQLERROR pkg_output.screen_write('Database error'|SQLERROR)


   SELECT customer_name
   FROM   order_table
   WHERE  customer_id = $customer_id_var
   AND    order_number > $order_number_var
   EOF
```

This script is dependent on two input parameters ($1 and $2), which are then assigned to our two variables. The variable values are inserted into the WHERE clause, thereby changing our query condition and result based on external values.

Unlike static queries, dynamic selects must always be performed at the time the calling program is run, since their result set depends directly on parameters passed in from the calling program. Here's a portion of the CobolScript code to call the above shell script:

```
 MOVE `'101101'`  TO cust_id.
 MOVE `22345`   TO order_nbr.

* We build our CALL argument below.  All of the following target
* variables are assumed to be components of the group item
* input_group.
 MOVE `query.sh `   TO input_arg_1.
 MOVE cust_id       TO input_arg_2.
 MOVE ` `           TO input_arg_3.
 MOVE order_nbr     TO input_arg_4.
 MOVE ` >error.txt` TO input_arg_5.

* At this point, input_group has a literal value of
* `query.sh '101101' 22345 >error.txt`.  The two literals that follow
* query.sh are our two shell script parameters that will be used
* inside the WHERE clause of the query.
 CALL input_group.

 OPEN `error.txt` FOR READING.
 READ `error.txt` INTO error_rec AT END MOVE 1 TO eof.
 CLOSE `error.txt`.
 MOVE 0 TO eof.

 IF error_rec(1:14) = `Database error` THEN
    DISPLAY error_rec
 ELSE
    OPEN `queryresult.dat` FOR READING
```

```
   PERFORM UNTIL eof
      READ `queryresult.dat` INTO query_rec AT END MOVE 1 TO eof
      DISPLAY query_rec
   END-PERFORM
   CLOSE `queryresult.dat`
END-IF.
STOP RUN.
```

By building the CALL argument in this manner, you can easily pass the values in CobolScript variables as parameters to shell scripts. These parameters can then be used in select statement conditions that are inside the shell script.

Inserts

We'll be doing database inserts a bit differently than we handled queries, since inserts tend to be involve much more text input than dynamic select statements do.

A batch ASCII file loading utility will simplify the task of inserting database rows from CobolScript input. The insert example that we give below assumes that such a utility is available for you to use.

Here's the important CobolScript code for our insert:

```
FD `order.dat` RECORD IS 57 BYTES.
1  order_rec
 5  rec_cust_id           PIC X(10).
 5  rec_order_nbr         PIC 9(6).
 5  rec_order_val         PIC 99999.99.
 5  rec_tax_val           PIC 99999.99.
 5  rec_salesperson_nbr   PIC 9(5).
 5  rec_date_and_time_val PIC X(14).

1  order_info.
 5  cust_id           PIC X(10).
 5  order_nbr         PIC 999999.
 5  order_val         PIC 99999.99.
 5  tax_val           PIC 99999.99.
 5  salesperson_nbr   PIC 99999.
 5  date_and_time_val.
   10  date_val        PIC X(8).
   10  time_val        PIC X(6).

* First we assign our values to be inserted.  This is a simplification;
* It's likely that you would first collect at least some of this data
* from the user on a web page form or from keyboard input.
 MOVE `'101101'`  TO cust_id.
 MOVE `22345`     TO order_nbr.
 MOVE 199.95      TO order_val.
 MOVE 12.90       TO tax_val.
 MOVE 1226        TO salesperson_nbr.
```

```
ACCEPT date_val  FROM DATE.
ACCEPT time_val  FROM TIME.

MOVE order_info TO order_rec.

OPEN `order.dat` FOR WRITING DELIMITED WITH `,`.
WRITE order_rec TO `order.dat`.
CLOSE `order.dat`.

CALL `sqlins configfile.txt order.dat >loadinfo.txt`.
DISPLAYASCIIFILE `loadinfo.txt`.
STOP RUN.
```

Most batch loading utilities take a configuration file input and produce one or several file outputs. Normally, the configuration file names all the other files involved, such as the input data file, the output information file, and an output 'bad' record file that contains all data records that were not successfully inserted in the. In the CALL statement above, however, we include the order.dat and loadinfo.txt files to enhance your understanding of this operation, since we don't provide a configuration file example.

Consult your load utility's documentation for information on how to construct the load configuration file.

Updates

Database updates are perhaps the most code-intensive operations to perform using CobolScript. The technique we employ to do updates uses portions of both our dynamic select and our insert operation techniques.

We'll use the following update statement as our starting point:

```
UPDATE order_table
SET  customer_name    = $customer_name_var
    ,order_val        = $order_val_var
    ,salesperson_nbr  = $salesperson_nbr_var
    ,update_timestamp = TO_DATE('DDMMYYYYhh24miss', $date_and_time_val)
WHERE  customer_id   = $customer_id_var
AND    order_number  = $order_number_var
```

As was the case in our dynamic select example, the fields that are preceded by a $ sign are passed in to the update statement as shell script variable values. This time, however, we'll use an interim file to transfer these variables from the CobolScript program to the shell script, rather than pass all of these variables as parameters to the shell script.

The new shell script will extract all of our relevant variables from a data file that we generated in CobolScript. Since we're looking at the shell script before we examine our CobolScript program, assume for now that the data file update.dat is a comma-delimited file that contains our field data in a single record, and in the following order:

```
customer_name_var,order_val_var,salesperson_nbr_var,date_and_time_val,customer_id_var,order_number_var
```

The shell script is below. Note that we've chosen to use the Unix `cut` command to extract our CobolScript variable values from update.dat. Consult your man pages for an explanation of this command:

```
#!/bin/ksh
customer_name_var=`cut -f 1 -d ',' update.dat`
order_val_var=`cut -f 2 -d ',' update.dat`
salesperson_nbr_var=`cut -f 3 -d ',' update.dat`
date_and_time_val=`cut -f 4 -d ',' update.dat`
customer_id_var=`cut -f 5 -d ',' update.dat`
order_number_var=`cut -f 6 -d ',' update.dat`

sqllogin 'userid/passwd' <<EOF >updateresult.dat
   SET HEADING OFF
   SET ECHO OFF
   SET BREAK OFF
   WHENEVER SQLERROR pkg_output.screen_write('Database error'|SQLERROR)

   UPDATE order_table
   SET  customer_name    = $customer_name_var
       ,order_val        = $order_val_var
       ,salesperson_nbr  = $salesperson_nbr_var
       ,update_timestamp = TO_DATE('DDMMYYYYhh24miss',
                                   $date_and_time_val)
   WHERE  customer_id   = $customer_id_var
   AND    order_number  = $order_number_var
   EOF
```

And here's our CobolScript code to call the above shell script. Assume that the shell script is named update.sh and is located in the working directory of the CobolScript program:

```
FD `update.dat` RECORD IS 59 BYTES.
1  order_rec.
 5   customer_name_var     PIC X(10).
 5   order_val_var         PIC 99999.99.
 5   salesperson_nbr_var   PIC 9(5).
 5   date_and_time_val     PIC X(14).
 5   customer_id_var       PIC X(10).
 5   order_number_var      PIC 9(6).
1  order_info.
 5   customer_name_var     PIC X(10).
 5   order_val_var         PIC 99999.99.
 5   salesperson_nbr_var   PIC 9(5).
 5   date_and_time_val.
  10  date_val        PIC X(8).
  10  time_val        PIC X(6).
 5   customer_id_var       PIC X(10).
```

```
  5   order_number_var      PIC 9(6).

* First we assign our values to be updated.  This is a simplification;
* It's likely that you would first collect at least some of this data
* from the user on a web page form or from keyboard input.
 MOVE `Larry Melman` TO customer_name_var.
 MOVE 199.95         TO order_val_var.
 MOVE 1226           TO salesperson_nbr_var.
 ACCEPT date_val     FROM DATE.
 ACCEPT time_val     FROM TIME.
 MOVE `'101101'`     TO customer_id_var.
 MOVE `22345`        TO order_number_var.

 MOVE order_info TO order_rec
 OPEN `update.dat` FOR WRITING DELIMITED WITH `,`.
 WRITE order_rec TO `update.dat`.
 CLOSE `update.dat`.

* Since all of our variables were written to a file to be used by the
* shell script, we don't pass any parameters to the shell script when
* we call it.
 CALL `update.sh>error.txt`.

 OPEN `error.txt` FOR READING.
 READ `error.txt` INTO error_rec AT END MOVE 1 TO eof.
 CLOSE `error.txt`.
 MOVE 0 TO eof.
 IF error_rec(1:14) = `Database error` THEN
   DISPLAY error_rec
 ELSE
   DISPLAYASCIIFILE `updateresult.dat`.
 END-IF.
 STOP RUN.
```

Although the code for the update technique is a bit more involved than the code for our select and insert techniques (primarily because we use a data file interface with the shell script in the update, rather than passing parameters to the script), it's still relatively straightforward. Of course, if you don't exceed the shell script parameter limit, an update script can still be called using parameters, just like the dynamic select example.

Chapter 5

Building Web-Based Systems

This chapter will describe techniques that can be used for building web-based systems with CobolScript. Since CobolScript is an interpreted language, it lends itself well to the debugging and tweaking that are often necessary when outputting HTML documents. You'll find that it's very easy to write small pieces of CobolScript code and then run and re-run the code in your web browser to see if you get the desired results. CobolScript also has syntax specifically designed to simplify and quicken the development of web systems, such as the ACCEPT DATA FROM WEBPAGE statement, the GETENV command, and the GETWEBPAGE command, all of which are described in this chapter.

ICON KEY

➲ Important point

If you're still confused about why you need a language other than HTML to create web pages, the answer is that you don't, if all that you're interested in doing is displaying static web pages. However, if you want your site visitors to interact with your web pages in any way; if you want to display or not display certain HTML based on conditions; or if you want to build a web-based *system*, then a programming language like CobolScript, not just a markup language like HTML, is required. Furthermore, as you become more familiar with web programming., you will discover that using a web server and standard browsers to run CobolScript web-based systems that are internal to your organization (*intra*nets) can be an efficient and economical alternative to systems that have a client-side component that must be individually installed and managed on each user's machine.

CobolScript normally communicates with a web server through CGI (the Common Gateway Interface). The Common Gateway Interface is a type of protocol; it defines a method of interaction between the web server and external programs, which are normally run by the web server in only two situations:

- When a form on an active web page is submitted;
- When a URL that calls a program (as opposed to a URL that calls a static web page) is typed into the *Location:* text box, or its equivalent, in a browser.

When data from a web page is sent to a CobolScript program, the data is encoded in accordance with the CGI protocol. The CobolScript engine can automatically decode this data stream when it has been submitted via the *Post method* and place each field of data in a corresponding CobolScript variable. This makes CobolScript a very easy programming language to use for web and internet development. Instead of building interfaces to web servers, you can focus your programming efforts on the business logic that belongs in your code.

To run the program examples in this chapter, or to run any CobolScript web programs, for that matter, you must have access to a web server. You must also have installed the CobolScript engine

on the same machine as your web server software, ideally in the web server's cgi-bin directory. If you have installed the CobolScript engine on your PC, you can install web server software on your PC as well, which will allow you to test your web development code without uploading it to a different machine. By using a web server on your own PC, you won't even need an internet or network connection to run your code. The Apache web server and derivatives work well for Unix platforms, and OmniHTTPd is a good web server for Windows®. Both are free. For further information on how to install CobolScript for use with a web server, see the **Installing CobolScript** section of Chapter 1, *Introduction to CobolScript/Installation Instructions*. Refer to the section **Running CobolScript from a Web Server and Browser** in Chapter 2, *Getting Started with CobolScript*, for general information on steps you must take for your programs to be capable of being run from a browser.

Interacting with a Web Server and Web Browser

Figure 5.1 provides a (simplified) representation of the normal methods by which CobolScript interacts with a web server and browsers. The browser sends data to the server when a CGI form is submitted or a free-text URL calling a program is completed, and this information is then passed directly from the server to CobolScript. The CobolScript engine interprets the inputs and makes them available to your CobolScript program. Your program then creates custom web page content, either based on the browser inputs or other information, and delivers this content back to the browser (actually, this delivery is done via the web server, but this interaction is excluded from the diagram for the sake of clarity) in the form of *virtual HTML*. Virtual HTML differs from static

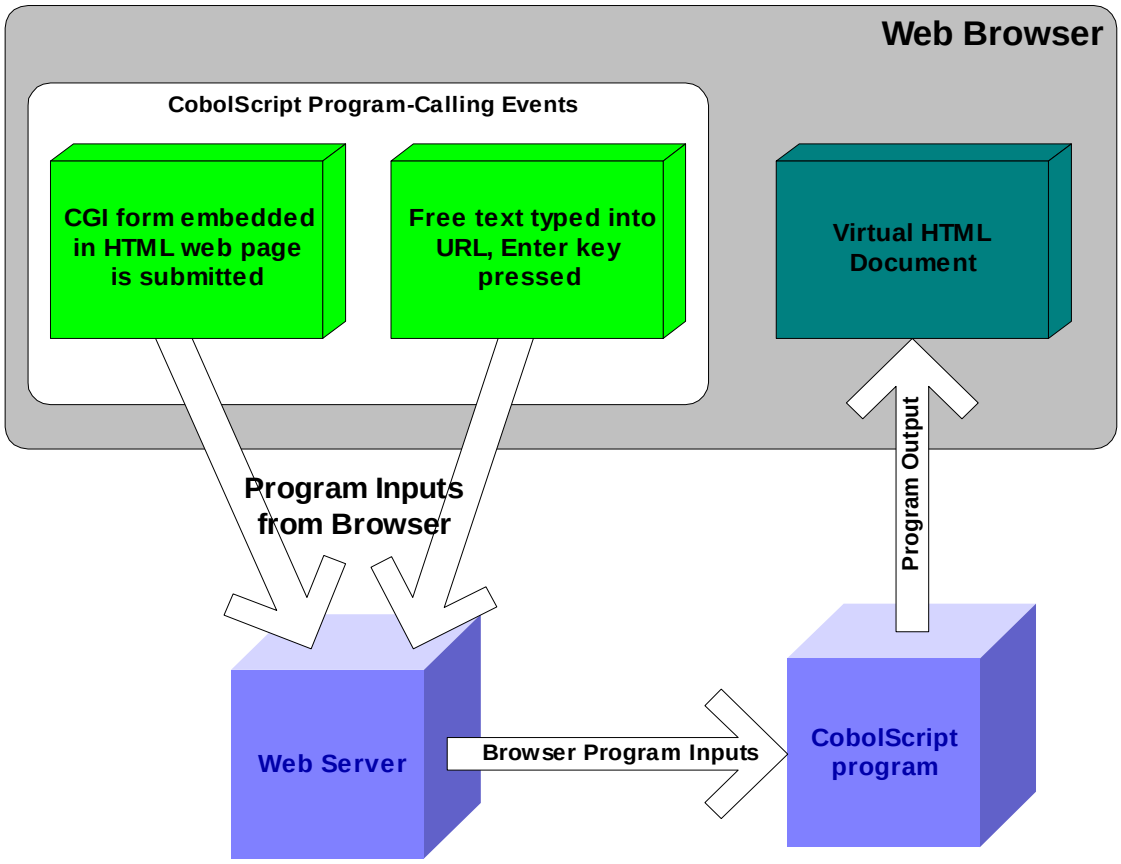


Figure 5.1 – A representation of CobolScript program interactions with a web browser and web server.

HTML in that virtual HTML is HTML code that has been output by a program, while static HTML resides in an independent HTML file. There is no syntactical difference between the two.

Creating Virtual HTML

Creating a virtual HTML document is simply a matter of displaying valid HTML to standard output. The example program below, which we'll call hello1.cbl, is very simple CobolScript code that will do just this, without any conditions or input processing.

To run the example, first place it in your web server's cgi-bin directory. Then, if you are running your browser and your web server on the same machine, and 127.0.0.1 is your web server's *loopback address* (the IP address that a machine typically uses to refer to itself), execute the program by typing http://127.0.0.1/cgi-bin/cobolscript.exe?hello1.cbl in your browser's URL window. If your web server is on a different machine than your browser but you know your server IP address, just substitute that address for 127.0.0.1.

You can also run this program from a command line by simply typing the following at the command prompt:

```
cobolscript.exe hello1.cbl
```

This will display the raw HTML output to your command line screen.

Here's the hello1.cbl code:

```
DISPLAY `Content-type: text/html `.
DISPLAY LINEFEED.
DISPLAY `<HTML><BODY>`.
DISPLAY `<CENTER>Hello World</CENTER>`.
DISPLAY `</BODY></HTML>`.
GOBACK.
```

You can see that the first text we display is the MIME header, which is this exact literal:

```
`Content-type: text/html`
```

This is followed immediately by the display of a LINEFEED character. Displaying a MIME header, followed by a linefeed, indicates to the web server that the program output that will follow the header will be a certain MIME type of input. In this case (and in the vast majority of your CGI programming), the MIME type is *text/html*, which means that we intend to output HTML content. The web server will recognize this MIME type and pass the remainder of our output on to the browser as HTML.

It's very important to remember to display the correct MIME header, followed by a line with only a linefeed, in the beginning of your CobolScript CGI programs. Failing to do this may prevent anything at all from displaying in your browser when you attempt to run your programs; depending on how your web server is configured, you may or may not get an appropriate error message in your browser window.

After the program has displayed all of the HTML (which is then transferred by the web server to the browser), it executes the GOBACK command to terminate processing, and your browser window will have the phrase "Hello World" in it.

Creating an HTML Form

If you want to create a web page that will allow your users to enter data, the simplest way to do this is by using an HTML form. Forms allow you to create text boxes, text areas, list boxes, check boxes, and radio buttons to collect data, reset buttons to clear data entries, and submit buttons to submit the data to a receiving program. See Chapter 7 for a detailed discussion on how to use each of the form components in programs.

The FORM tag, along with its end tag, are used to demarcate the form, which is essentially a data input area inside an HTML document. Every form has an associated action; this action is specified in the ACTION component of the FORM tag. The ACTION argument is an URL that names a CGI program that will be executed when the browser user submits the form. In the case of the program below, which we'll name hello2.cbl, the action is /cgi-bin/cobolscript.exe?hello2.cbl. In this example, when you submit the form on your web browser, it will run the hello2.cbl program again. Of course, since incoming data is not processed by this program, the data typed in the text box is lost after the form is submitted.

Here's the code for hello2.cbl:

```
DISPLAY `Content-type: text/html`.
DISPLAY LINEFEED.

DISPLAY `<HTML><BODY>`.
DISPLAY `<CENTER>Hello World</CENTER>`.
DISPLAY `<FORM ACTION="/cgi-bin/cobolscript.exe?hello2.cbl" `
        & `METHOD=POST>`.
DISPLAY `<INPUT TYPE=TEXT NAME="my_variable">`.
DISPLAY `<INPUT TYPE=SUBMIT VALUE="Click here to Submit">`.
DISPLAY `</FORM>`.
DISPLAY `</BODY></HTML>`.

GOBACK.
```

The program above uses a simple text box (created by INPUT TYPE=TEXT) to collect information. You'll notice that the text box has a NAME argument associated with it, and that the name is my_variable; this is the *CGI field name*. The CGI field name is the name of a CGI variable that will hold the contents of the text box when the form is submitted from the web page.

Capturing Input Data from a Web Page

At this point, you're probably wondering how the data from the CGI variable gets into a variable in a CobolScript program. In CobolScript, when your program needs to get form data from a web page, you just use a special form of the ACCEPT statement called ACCEPT DATA FROM WEBPAGE. Here's an example:

```
ACCEPT DATA FROM WEBPAGE.
```

This command will get the CGI data that was submitted, parse it, decode it, and place the contents in CobolScript variables that have the same names as the incoming CGI variables.

To accept data from a CGI form into a CobolScript program, you must define variables to capture the contents of the incoming CGI variables. The CobolScript variables must have the same names as the CGI variables. The program in this section, which we'll call hello3.cbl, accepts a CGI variable called "my_variable" into a like-named CobolScript 40 byte alphanumeric variable that we'll define here:

```
1 my_variable     PIC X(40).
1 content_length  PIC 9(05).
```

If you look at our hello3.cbl code segment below, you'll notice that we use the GETENV command before we accept the CGI data from the web page. This command gets the value of the web server environment variable that is specified as the GETENV argument and places its contents into a CobolScript variable. The environmental variable CONTENT_LENGTH holds the CGI query string's actual length. The query string is the raw data stream that the POST method uses to send data to a target program, so if this the length of this string is greater than zero, we know that there is data to accept. It's good practice to get the value of CONTENT_LENGTH at the beginning of your CobolScript program, because by doing this, you know whether or not there is CGI data waiting for you to process. If the value of CONTENT_LENGTH is zero, then you know that the user is simply running your web based application for the first time and has not submitted a form on it. If CONTENT_LENGTH is greater than zero, then you know that the user has submitted a form from your application.

The ACCEPT DATA FROM WEBPAGE command handles all of the parsing of the POST method-submitted data internally, so you don't have to worry about decoding the CGI data passed to the web server.

Here's the rest of the code for hello3.cbl:

```
GETENV USING `CONTENT_LENGTH` content_length.
IF content_length > 0
   ACCEPT DATA FROM WEBPAGE
END-IF.

DISPLAY `Content-type: text/html`.
DISPLAY LINEFEED.
DISPLAY `<HTML><BODY>`.
DISPLAY `<CENTER>Hello World</CENTER>`.
DISPLAY `my_variable: ` & my_variable.

DISPLAYLF `<FORM ACTION="/cgi-bin/cobolscript.exe?hello3.cbl" `
         & `METHOD="POST">`
DISPLAY `<INPUT TYPE="TEXT" NAME="my_variable" VALUE="`
        & my_variable & `">`.
DISPLAYLF `<INPUT TYPE="SUBMIT" VALUE="Click here to Submit">`
DISPLAYLF `</FORM>`.
DISPLAY `</BODY></HTML>`.
```

Again, if CONTENT-LENGTH is greater than zero, there is CGI data waiting to be accepted, and therefore the ACCEPT DATA FROM WEBPAGE statement should be executed. This statement will look at the CGI data stream being sent from the web server, decode it, and match the CGI form

variable names with CobolScript variable names. That is why both the CobolScript variable and the form field are named my_variable. Because these two names correspond, the data associated with the form field *my_variable* will be moved to the contents of the CobolScript variable *my_variable*. All decoding and parsing of the CGI data stream is performed automatically.

Important note: The maximum elementary variable size in CobolScript is 2,000 bytes. If you happen to have an individual CGI field that has contents greater than 2,000 bytes, only the first 2,000 bytes of data will be stored in any target CobolScript variable that is an elementary data item. The rest will be truncated.

DISPLAY and DISPLAYLF

The DISPLAY and DISPLAYLF commands differ most significantly in the way they handle group items. This has special relevance in the context of CGI development, since you may or may not want your HTML output to have line breaks in it that makes it more readable. The differences in the two are:

- The DISPLAY command will print a literal or the contents of any variable to standard output. After all of the arguments to DISPLAY have been displayed, a linefeed character displays, terminating the output. In the case of a group-level data item DISPLAY, all individual components of the group item will print on the same line.
- The DISPLAYLF command will print a literal or the contents of a variable to standard output, followed by an ASCII line feed character between each individual component of a group-level data item, or each individual argument, if multiple arguments are specified. After all of the arguments have been displayed, another linefeed character is displayed to complete the output.

Let's take a look at how DISPLAY and DISPLAYLF each display the following group-level data item. Note the use of the Implied PIC X(n) FILLER variables (explained in the **Variables** section of Chapter 3):

```
1 form_var.
 5 `<FORM ACTION=cobolscript.exe?test.cbl METHOD=POST>`.
 5 `<INPUT TYPE=TEXT NAME=field1>`.
 5 `<INPUT TYPE=SUBMIT VALUE=Submit>`.
 5 `</FORM>`.
```

The statement DISPLAY form_var will produce the following output (all on a single line):

```
<FORM ACTION=cobolscript.exe?test.cbl METHOD=POST><INPUT TYPE=TEXT NAME=field1><INPUT TYPE=SUBMIT VALUE=Submit></FORM>
```

The statement DISPLAYLF form_var will produce the following output:

```
<FORM ACTION=cobolscript.exe?test.cbl METHOD=POST>
```

```
<INPUT TYPE=TEXT NAME=field1>
<INPUT TYPE=SUBMIT VALUE=Submit>
</FORM>
```

Retrieving Web Pages

If you ever need to build an application that retrieves web pages, you can use the GETWEBPAGE command. It connects to a web server, retrieves a given web page, and saves it to a user-specified file.

The program below called WEB.CBL demonstrates the usage of the GETWEBPAGE command. It utilizes a standard data structure called TCPIP-RETURN-CODES. This group level data item will be populated with information from the specific web server you are accessing. TCPIP-RETURN-CODE is a number, while TCPIP-RETURN-MESSAGE is a string. Typically a successful return code for this operation will be zero, and the return message will contain a string describing the number of bytes received for a particular web document.

Here's a portion of the code for WEB.CBL:

```
1 TCPIP-RETURN-CODES.
 5 TCPIP-RETURN-CODE     PIC 9(07).
 5 TCPIP-RETURN-MESSAGE  PIC X(255).

MOVE `www.deskware.com` TO host_name.
MOVE `/cobol/cobol.htm` TO web_page_name.
MOVE `web.txt` TO file_name.
DISPLAY `<` host_name `>`.
DISPLAY `<` web_page_name `>`.
DISPLAY `<` file_name `>`.
GETWEBPAGE USING host_name web_page_name file_name.
DISPLAY `TCPIP-RETURN-CODES: ` TCPIP-RETURN-CODES.
GOBACK.
```

The host name in this example is a fully qualified domain name – www.deskware.com. It is also acceptable to specify a raw IP address as the host name argument. The file name argument is used to create a file with the HTML that you are retrieving. The named file is overwritten each time the GETWEBPAGE command is executed.

Chapter

6

Network and Internet Programming Using CobolScript®

ICON KEY

➲ Important point

While a combination of static HTML pages and basic CGI programs written in nearly any programming language can address the on-line requirements of internet information systems, few languages can satisfactorily address the interface and networking requirements of internet systems, at least not without compromising platform independence. With CobolScript, however, you can transfer files, receive and deliver email messages, and conduct point-to-point communications with other computers, all by using standard CobolScript commands. Because these commands all use the TCP/IP protocol or extensions such as FTP, SMTP, and HTTP, cross-platform communication is handled the same way as same-platform communication.

This chapter provides some basic examples of how to transfer files, send and receive emails, and program TCP/IP sockets. By learning and expanding on these examples, you will be able to create, in CobolScript code, the interfaces that your system requires.

Transferring Files using FTP

Sharing files is one of the fundamental motivations for networking computers. FTP (File Transfer Protocol) is a protocol for transferring files over a TCP/IP network. FTP is an effective way to share data between heterogeneous network hosts. CobolScript has commands that allow you to program FTP clients to transfer files to and from FTP servers.

Most computers on the Internet support FTP access. Before you can build a program that will access files on these FTP servers, however, you will need the following:

- The name of the system on the network that has the files you want to obtain, or on which you want to place files. In other words, you need to know the fully qualified domain name or IP address of the host that you want to transfer files from and to.
- A valid user name and password to use on the remote computer. Many remote computers will allow *anonymous ftp*, which allows you restricted FTP access by using the user name anonymous and your email address as the password.

FTP is extremely useful for transmitting data rapidly between sites that need to share information system data. Using FTP eliminates many usual considerations when transferring files. By using FTP:

- You won't need to worry about requiring both hosts to use the same types of disks or tapes to transfer files;
- You won't have to break up a file into several smaller files because the larger file won't fit on a single disk or as an email attachment.

CobolScript programs that transfer files using FTP commands can be scheduled to run at regular time intervals. This is allows you to have unattended file transfers between hosts.

When you try to connect to a remote computer using FTP, you will need to supply a valid user name and password. The CobolScript command FTPCONNECT is the command you should use to login to an FTP server. Here's an example:

```
MOVE `deskware.com` TO host_name.
MOVE `anonymous` TO user.
MOVE `interpreter@deskware.com` TO password.

FTPCONNECT USING host_name user password.
```

After you have connected to an FTP server, you should set the transfer type. This is done with the FTPASCII or FTPBINARY commands. If you will be transferring plain ASCII text files, you should use FTPASCII. By doing this, the server knows to convert the files to an ASCII format that your client computer can read. This is important because ASCII files on Windows machines are line terminated with carriage return and line feed ASCII characters, and on Unix-based machines, ASCII files are line terminated with only line feed characters. If you are connecting to a mainframe, text files are stored in EBCDIC format. Using the FTPASCII command before you transfer text files will ensure that you receive them in the ASCII format that is native to your client machine. Using the FTPASCII command is as simple as the following statement:

```
FTPASCII.
```

If you need to transfer binary data such as word processing documents or spreadsheet files, you should use the FTPBINARY command before transmitting files. This ensures that no ASCII translation is performed on your file during the transfer.

Another useful command is FTPCD. It allows you to change the directory on the FTP server that you are connecting to. Here's an example:

```
FTPCD USING `\ftp\data\interfaces`.
```

You should make sure that you use the correct directory naming structure for the FTP host that you are connecting to. The above example is a directory name on a Unix based host. If it were a Windows based server, you might use something like `C:\datafiles\output`, or on a mainframe you might use `"idy2v.data.acct"`.

The FTPGET and FTPPUT commands actually perform the file transfer operations. You should use FTPGET to get a file from an FTP server, and FTPPUT to send a file to an FTP server. Here are examples of these commands in complete statements:

```
FTPGET USING `order.dat`.
DISPLAY `FTPGET TCPIP-RETURN-CODES: ` & TCPIP-RETURN-CODES.
FTPPUT USING `order.dat`.
DISPLAY `FTPPUT TCPIP-RETURN-CODES: ` & TCPIP-RETURN-CODES.
```

Using Email Commands

Although you may never have thought of email as a system interfacing tool, this is in fact what it is, because email allows users to send and receive messages from a local machine to recipients on destination hosts, regardless of platform. Even if the email message is only textual, and is only meant to be read by the recipient and not cause any direct system action, the delivery and receipt of the email constitute a system interface.

A standard email message without attachments is simply a text file, made up of header lines that tell an email server how to deliver the message, and of the message content.

SMTP is an acronym for Simple Mail Transfer Protocol and POP3 for Post Office Protocol 3; they are the standard TCP/IP protocols for sending email and receiving email, respectively. CobolScript uses these protocols in its email commands, which enable the sending and receiving of simple email messages.

To use CobolScript to build programs that send email messages, you will need access to an SMTP server. Once you have this, you can use the CobolScript SENDMAIL command to send email. Here's an example:

```
COPY `tcpip.cpy`.
1 to_addresses.
 5  `<nobody1@ttttt.com>`.
 5  `Nobody <nobody2@ttttt.com>`.
 5  `nobody3@ttttt.com`.

1 from_address PIC X(n) VALUE `youremail@yourhost.com`.
1 subject PIC X(n) VALUE `mail.cbl test`.

1 message.
 5 `This is a test message from mail.cbl.`.
 5 FILLER PIC X VALUE LINEFEED.
 5 `Sent from me to you.`.

1 smtp_server PIC X(n) VALUE `yoursmptserver.com`.

SENDMAIL USING to_addresses
              from_address
              subject
              message
              smtp_server.
DISPLAY `TCPIP-RETURN-CODES: ` & TCPIP-RETURN-CODES.
```

Of course, you would substitute your addresses and message for the above addresses and message.

With CobolScript there are two commands for retrieving email messages, GETMAILCOUNT and GETMAIL. The GETMAILCOUNT command connects to your mail server and determines the number of messages in your inbox. The GETMAIL command retrieves a copy of a specific email message and saves its contents to a text file. GETMAIL does not remove the email message from the server. Here is an example of how to use these commands:

```
MOVE `youremail@yourhost.com` TO email_address.
MOVE `yourpassword` TO email_password.
MOVE 0 TO email_count.

GETMAILCOUNT USING email_address
                   email_password
                   email_count
                   smtp_server.
DISPLAY `Email count: ` & email_count.
DISPLAY `TCPIP-RETURN-CODES: ` & TCPIP-RETURN-CODES.

MOVE `youremail@yourhost.com` TO email_address.
MOVE `yourpassword` TO email_password.
MOVE 1 TO email_number.
MOVE `mymail.txt` TO email_file_name.
GETMAIL USING email_address
              email_password
              email_number
              email_file_name
              smtp_server.
DISPLAY`TCPIP-RETURN-CODES: ` & TCPIP-RETURN-CODES.
```

When the GETMAIL command retrieves an email message from a server, it appends the message to the specified text file. This means that if you want to retrieve a copy of all of your email messages, you should use GETMAILCOUNT to find out how many messages there are, and then perform a loop that retrieve each message. If you want each message to be in a separate text file, you should use a new text file name each time you call GETMAIL.

Important Note: When you are sending emails it is important to use a valid SMTP server. Generally it works best if your applications send all emails through your SMTP server, and then your SMTP server delivers the email to the user.

Using TCP/IP Commands

Several TCP/IP commands are available in CobolScript. They can be used for socket programming and obtaining DNS information about a host. They are:

- GETHOSTNAME
- GETHOSTBYNAME
- CREATESOCKET
- BINDSOCKET
- LISTENTOSOCKET
- CONNECTTOSOCKET
- ACCEPTFROMSOCKET
- RECEIVESOCKET
- SENDSOCKET
- SHUTDOWNSOCKET
- CLOSESOCKET

DNS Commands

The program below (which is the DNS.CBL sample program) demonstrates how to use the GETHOSTBYNAME command. You can run this program from your web browser by typing in the URL http://127.0.0.1/cgi-bin/cobolscript.exe?dns.cbl if you are running CobolScript and a web server on your local machine.

Both GETHOSTNAME and GETHOSTBYNAME require two special group level data items – TCPIP-HOSTENT and TCPIP-RETURN-CODES. These data structures are placeholders for return values that are populated when these commands are executed. The structures must be in your program in order for it to run properly when you use these commands.

GETHOSTNAME gets the TCP/IP hostname from your local machine and place the name in a CobolScript variable. The GETHOSTBYNAME is a much more advanced command. It contacts your DNS (Domain Name Server) and retrieves detailed information about a specified host name. It retrieves information such as aliases and host addresses associated with a particular domain name. Try running this example with some domain names like lycos.com or yahoo.com.

Here are the variable definitions for DNS.CBL. Note the two standardized TCP/IP structures that we mentioned earlier. These would normally just be placed in a copybook by themselves, such as tcpip.cpy, but we include them here to show their detail:

```
************************************
*             TCP/IP               *
*          DATA STRUCTURES         *
************************************
* GETHOSTBYNAME REQUIRES THE DATA    *
* STRUCTURE BELOW.  DO NOT CHANGE IT.*
************************************
 01 TCPIP-HOSTENT.
  05 TCPIP-HOSTENT-HOSTNAME                  PIC X(255).
  05 TCPIP-HOSTENT-NUM-ALIASES               PIC X.
  05 TCPIP-HOSTENT-ALIASES OCCURS 8 TIMES.
   10 TCPIP-HOSTENT-ALIAS                    PIC X(255).
  05 TCPIP-HOSTENT-ADDRESS-TYPE              PIC 9(7).
  05 TCPIP-HOSTENT-ADDRESS-LENGTH            PIC 9(7).
  05 TCPIP-HOSTENT-NUM-ADDRESSES             PIC X.
  05 TCPIP-HOSTENT-ADDRESSES OCCURS 8 TIMES.
   10 TCPIP-HOSTENT-ADDRESS                  PIC X(255).
************************************
* TCP/IP RETURN CODES DATA STRUCTURE *
* DO NOT CHANGE.                   *
************************************
 01 TCPIP-RETURN-CODES.
  05 TCPIP-RETURN-CODE                       PIC 9(7).
  05 TCPIP-RETURN-MESSAGE                    PIC X(255).

* Program-specific variables
************************************
 1 content_length   PIC 9(05).

 1 web_header_html.
  5 `Content-type: text/html`.
  5 ` `.
  5 `<HTML><BODY>`.
  5 `<BR>`.
  5 `<B>Sample CobolScript DNS Application</B>`.
  5 `<BR><BR>`.
  5 `Enter a Fully Qualified Domain Name or an IP address and then`
  5 ` click on the Resolve button.`.
  5 `<FORM ACTION="/cgi-bin/cobolscript.exe?dns.cbl" METHOD="POST">`.
  5 `<INPUT TYPE="TEXT" NAME="host_name" SIZE=60 VALUE="`.
  5 host_name  PIC X(80) VALUE `www.cornell.edu`.
  5 `">`.
  5 `<INPUT TYPE="SUBMIT" VALUE="Resolve">`.
  5 `</FORM>`.
  5 `<HR>`.

1 web_footer_html.
  5 `</BODY></HTML>`.
```

Here's our main paragraph of code for DNS.CBL. Since we're running this program from a browser, we first use the GETENV statement to determine whether we have input or not (see Chapter 5) and the output that we display is HTML:

```
MAIN.
   GETENV USING `CONTENT_LENGTH` content_length.

   IF content_length > 0
      ACCEPT DATA FROM WEBPAGE
   END-IF.

   IF host_name = SPACES
      MOVE `www.cornell.edu` TO host_name
   END-IF.

* Populate TCP/IP structure that is defined in included copybook.
   GETHOSTBYNAME USING host_name.

   DISPLAYLF web_header_html.
   PERFORM DISPLAY-TCPIP-INFO.
   DISPLAYLF web_footer_html.

   GOBACK.
```

The code module below displays each of the TCP/IP variables that are populated by the call to GETHOSTBYNAME, in an HTML table format. We're excluding most of this module's code from here because of its repetitive nature, but the entire code is in the DNS.CBL sample program:

```
DISPLAY-TCPIP-INFO.

  1 counter PIC Z9.

  DISPLAY `<TABLE BORDER=1 BGCOLOR="CCCCCC">`.

  DISPLAY `<TR BGCOLOR="lightgreen">`.
  DISPLAY `<TD><B>host_name:</B></TD>`.
  DISPLAY `<TD><B>` & host_name & `</B></TD>`.
  DISPLAY `</TR>`.
  .
  .
  .
  DISPLAY `</TABLE>`.
```

TCP/IP Socket Commands

CobolScript has commands for several TCP/IP socket operations. Socket programming is very similar to file I/O, except socket programming reads from and writes to sockets instead of files. A

socket is an endpoint of communication, created in software, and equivalent to a computer's network interface.

We have provided two sample programs that, when combined, demonstrate the use of socket operations – the first program is a socket server, and the second is its client. The server program should first be run from one command prompt window, and then the client program run from another . After they have both started, you can type a string in the client window that will be sent via TCP/IP to the server. This example can easily be modified to communicate with clients and servers on different platforms simply by changing the IP address (host name) parameters.

The server program (the sample program SERV.CBL) requires the same set of TCP/IP data structures that we defined in the previously discussed DNS.CBL program, as well the following user-defined variables:

```
1 host_name            PIC X(80).
1 socket_num           PIC 9(2).
1 connected_socket_num PIC 9(2).
1 port_num             PIC 9(5).
1 backlog_num          PIC 9(2).
1 string_var           PIC X(10).
1 receive_string       PIC X(20).
1  send_string         PIC X(20).
```

Here's the main code. Note the order of the socket server commands (CREATESOCKET, BINDSOCKET, LISTENTOSOCKET), which is necessary set-up for the socket before a connection can be accepted using ACCEPTFROMSOCKET:

```
*****************************************************
* This program requires that you have TCP/IP running
* on your machine.
*****************************************************
 MAIN.
   GETHOSTNAME USING host_name.
   DISPLAY `Starting Deskware Server on ` & host_name.

   MOVE 1 TO socket_num.
   MOVE 2 TO connected_socket_num.
   CREATESOCKET USING socket_num.
   DISPLAY `CREATESOCKET return code = <` & TCPIP-RETURN-CODE & `>`.

   MOVE 2500 TO port_num.
   BINDSOCKET USING socket_num port_num.
   DISPLAY `BINDSOCKET return code = <` & TCPIP-RETURN-CODE & `>`.

   MOVE 1 TO backlog_num.
   LISTENTOSOCKET USING socket_num backlog_num.
   DISPLAY `LISTENTOSOCKET return code = <` & TCPIP-RETURN-CODE & `>`.

   DISPLAY `Waiting to accept socket connection on port ` & port_num
```

```
           & `...`.
      ACCEPTFROMSOCKET USING socket_num connected_socket_num.
      DISPLAY `ACCEPTFROMSOCKET return code = <` & TCPIP-RETURN-CODE
           & `>`.

      MOVE SPACES TO receive_string.
      PERFORM ACCEPT-TCPIP-CONNECTIONS UNTIL receive_string(1:4) = `STOP`.

      DISPLAY `Shutting down Deskware Server`.

      SHUTDOWNSOCKET USING connected_socket_num 1.
      CLOSESOCKET USING connected_socket_num.

      SHUTDOWNSOCKET USING socket_num 1.
      CLOSESOCKET USING socket_num.
      GOBACK.

    ACCEPT-TCPIP-CONNECTIONS.
      MOVE SPACES TO receive_string.
      RECEIVESOCKET USING connected_socket_num receive_string.
      DISPLAY `TCP/IP return code = <` & TCPIP-RETURN-CODE & `>`.
      DISPLAY `TCP/IP return message = <` & TCPIP-RETURN-MESSAGE & `>`.

      DISPLAY `This was received: ` & receive_string.

      MOVE `GOT IT` TO send_string.
      SENDSOCKET USING connected_socket_num send_string.
      DISPLAY `TCP/IP return code = <` & TCPIP-RETURN-CODE & `>`.
      DISPLAY `TCP/IP return message = <` & TCPIP-RETURN-MESSAGE & `>`.
      DISPLAY `This was sent: ` & send_string.
```

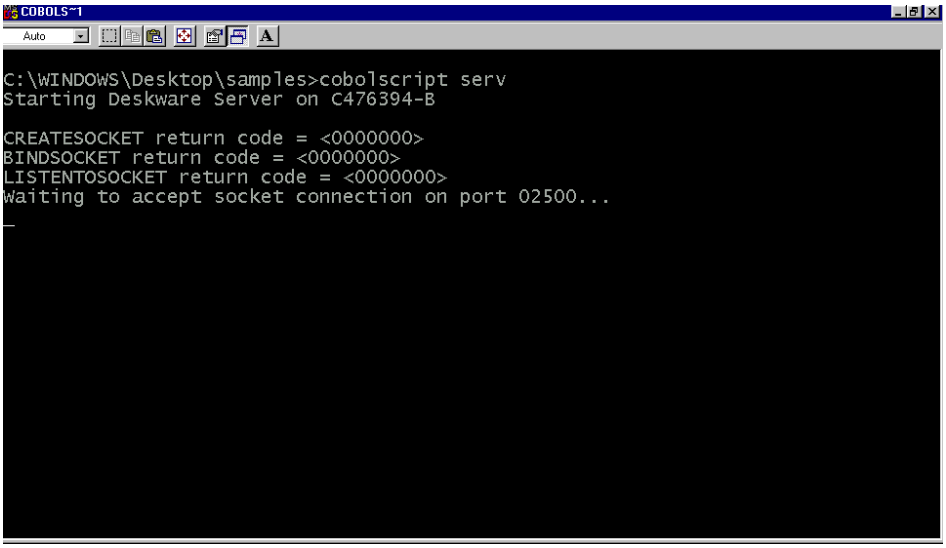


Figure 6.1 – Command prompt with server program running.

The client program (the sample program CLIENT.CBL) requires the same set of TCP/IP data structures as defined in DNS.CBL, and also the following user-defined variables:

```
1 host_name            PIC X(80).
1 socket_num           PIC 9(2).
1 connected_socket_num PIC 9(2).
1 port_num             PIC Z9999.
1 backlog_num          PIC 9(2).
1 string               PIC X(10).
1 receive_string       PIC X(20).
1 send_string          PIC X(20).
1 stop_var             PIC 9.
```

The client code in this example assumes that the client and server programs are running on the same machine (hence the move of the loopback address to host_name).

Note the interaction points between the previous server program and this client program; the server uses ACCEPTFROMSOCKET to accept a connection initiated by the client's CONNECTTOSOCKET statement. Once the connection is established, the server uses RECEIVESOCKET to receive the data transmitted from the client using SENDSOCKET. Once the transmission is complete, they reverse, and the server sends the string `GOT IT` back to the client as a way to confirm the data transmission. Here's our client code:

```
DISPLAY `Starting Deskware Client (type STOP to exit).`.
MOVE 1 TO socket_num .
CREATESOCKET USING socket_num.
DISPLAY `CREATESOCKET return code = <` & TCPIP-RETURN-CODE & `>`.

MOVE 2500 TO port_num.
```

```
* We are using the loop back IP in this example;
* uncomment the line below and comment out the move
* to actually get the host name
* GETHOSTNAME USING host_name
 MOVE `127.0.0.1` TO host_name.

 DISPLAY `Your hostname is: ` & host_name.
 CONNECTTOSOCKET USING socket_num host_name port_num.
 DISPLAY `CONNECTTOSOCKET return code = <` & TCPIP-RETURN-CODE & `>`.
 DISPLAY TCPIP-RETURN-MESSAGE.

 PERFORM SEND-DATA-TO-SERVER UNTIL stop_var.

 SHUTDOWNSOCKET USING socket_num 1.
 CLOSESOCKET USING socket_num.
 GOBACK.

 SEND-DATA-TO-SERVER.
   ACCEPT send_string FROM KEYBOARD
          PROMPT `Data to send to port 2500: `.

   SENDSOCKET USING socket_num send_string.
   DISPLAY `SENDSOCKET return code = <` & TCPIP-RETURN-CODE & `>`.
   DISPLAY TCPIP-RETURN-MESSAGE.

   MOVE SPACES TO receive_string.
   RECEIVESOCKET USING socket_num receive_string.
   DISPLAY `RECEIVESOCKET return code = <` &  TCPIP-RETURN-CODE & `>`.
   DISPLAY `This was received: <` & receive_string & `>`.
   DISPLAY `RECEIVESOCKET return code = <` & TCPIP-RETURN-CODE & `>`.
   DISPLAY TCPIP-RETURN-MESSAGE.

   IF send_string(1:4) = `STOP` THEN
      MOVE 1 to stop_var
   END-IF.
```

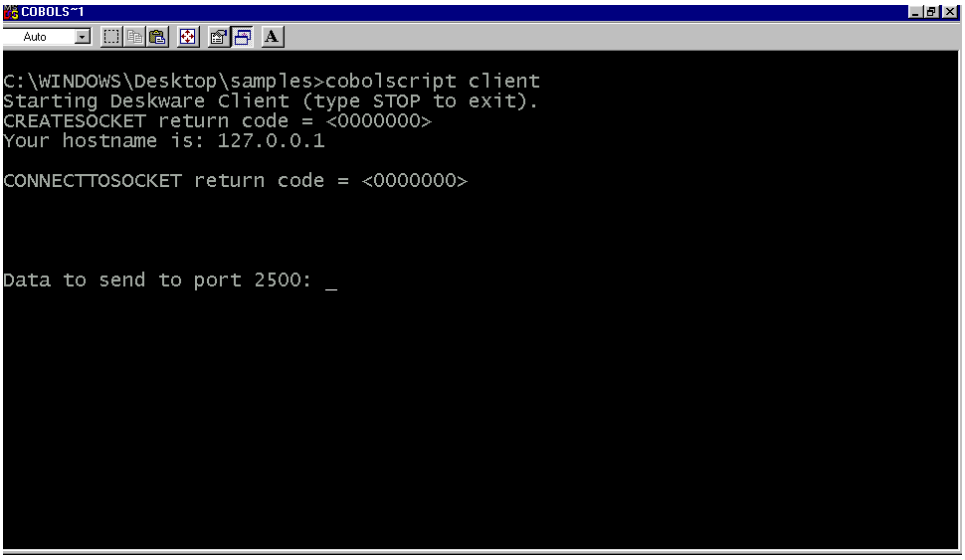


Figure 6.2 – Command prompt with client programming.

Chapter 7

Advanced Internet Programming Techniques Using CobolScript®

This chapter discusses advanced techniques for processing internet data retrieval using CobolScript. We also briefly discuss the use of embedded JavaScript in your CobolScript programs, for handling tasks suited for client-side processing.

Our discussion of CGI data retrieval and processing assumes that the incoming CGI data is always submitted using the POST method. With the POST method, URL-encoded data is delivered to the CobolScript program through standard input. The CobolScript engine reads all of this data, decodes it, and places it in corresponding CobolScript variables.

Also, all code examples assume that you've set your file permissions correctly. As mentioned in earlier chapters, if you're working in a Unix environment, always make certain that the file permissions on your CobolScript internet programs allow the CGI user (usually user 'nobody') to execute them.

All of our web and internet code examples also assume that you have not modified your web server software to make CobolScript your default CGI interpreter. However, making CobolScript the default CGI interpreter is usually relatively easy, depending on your web server. Doing so will simplify the URLs you use to call CobolScript programs; instead of calling a program with a URL such as the following:

http://www.cobolscript.com/cgi-bin/cobolscript.exe?samples.cbl

You would instead use a URL such as:

http://www.cobolscript.com/cgi-bin/samples.cgi

Or, if your web server is flexible enough to allow modification to the CGI program extension, even this:

http://www.cobolscript.com/cgi-bin/samples.cbl

However, by modifying your web server's configuration in this manner, you will disable any interpreted programs already existing on the server that relied on the previous configuration, and that were written in a different language such as Perl (these programs will be treated as CobolScript programs and will fail to run because they are not valid CobolScript code). Use your own discretion

in making this type of modification; a web system built from scratch, using only CobolScript code, is an ideal candidate for this kind of configuration change; a web system with existing interpreted code written in other languages is not. Consult your web server's documentation for more information on how to configure the default interpreter path and the default CGI extension.

Environment Variables

Environment variables are system variables that exist within a particular computer user's environment. With regard to a web server, the full set of environment variables is recreated each time a CGI process is executed. You can think of these variables as placeholders that a web server uses to pass data about an HTTP request from the server to the CGI-processing application, i.e., your CobolScript program.

With CobolScript, environment variables are accessed with the GETENV command:

```
GETENV USING <environment variable> <cobolscript variable>.
```

The names for environment variables are system-specific. Fortunately, most web servers have adopted many of the same names. Here are a few of the standard ones; experiment with these variables in the GETENV statement to determine the formats of the contents of each of these variables:

Environment Variable	**Description**
CONTENT_LENGTH	Size of the attached incoming CGI data in bytes (characters).
CONTENT_TYPE	The MIME type of the incoming CGI data
PATH_INFO	Path to be interpreted by the CGI application.
PATH_TRANSLATED	The virtual-to-physical mapping of the file on the system.
QUERY_STRING	The URL-encoded string that was submitted to the web server
REMOTE_ADDR	The IP address of the agent making the CGI request.
REMOTE_HOST	The fully qualified domain name of the requesting agent.
REMOTE_IDENT	Data reported about the agents' connection to the server.
REMOTE_USER	The User ID sent by the client agent.
REQUEST_METHOD	The request method used by the client. For CobolScript applications, this should be "POST".
SCRIPT_NAME	The path identifying the CGI application requested.
SERVER_NAME	The server name of the requested URL. This will either be the IP address of the server or the fully qualified domain name.
SERVER_PORT	The port where the client request was received by the server.
SERVER_PROTOCOL	The name and revision of the request protocol.

Environment Variable	Description
SERVER_SOFTWARE	The name and version of the server software. For example: "OmniHTTPd/1.01 (Win32; I386)"

Some web servers do not support all of these environment variables. You should consult your web server documentation to find out what environment variables are supported by your specific web server.

All normal web servers support the CONTENT_LENGTH environment variable. Because of this, we recommend getting this variable when your CobolScript application is first invoked via a web server, like this:

```
GETENV USING `CONTENT_LENGTH` content_length.
IF content_length > 0
   ACCEPT DATA FROM WEBPAGE
END-IF.
```

By doing this, you will know if a form was submitted to your application or not. If your application was called directly from a typed URL, outside of a form submission, the value of CONTENT_LENGTH would be 0 and you would not need to accept CGI data from the web server. Normally, when the ACCEPT DATA FROM WEBPAGE statement is executed, CobolScript will begin reading data from the CGI stream and place the contents in the appropriate CobolScript variables. Of course, it's not necessary to do this if no CGI data has been sent to the web server.

Sometimes, web servers are configured to not populate certain environment variables such as REMOTE_HOST. This is often done because there is a time cost in resolving the IP addresses of each client as it makes a request. However, you can still resolve these IP addresses by using the GETHOSTBYNAME command. Simply get the REMOTE_ADDR environment variable that contains the IP address of the client, and use this as the argument to GETHOSTBYNAME:

```
GETENV USING `REMOTE_ADDR` download_ip.
GETHOSTBYNAME USING download_ip.
MOVE TCPIP-HOSTENT-HOSTNAME TO download_host.
```

The GETHOSTBYNAME command will resolve the IP address to its fully qualified domain name, and the result will be placed in the TCPIP-HOSTENT-HOSTNAME variable. If the DNS server cannot resolve the IP address, the TCPIP-HOSTENT-HOSTNAME will be spaces. Also, for completeness, the TCP/IP return code values should always be examined after executing GETHOSTBYNAME to determine whether the command executed successfully or not.

CGI Form Components

As we saw in Chapter 5, the ACCEPT DATA FROM WEBPAGE statement can capture HTML form data. This data capture can be done from all of the possible submitting form components – text boxes, multi-line text boxes, list boxes, drop down list boxes, radio buttons, check boxes, and submit buttons. In this section, we describe how to process input from each of these components. The example program input.cbl illustrates the data capture for each of these components (except submit buttons). This sample program can be found in the sample programs included with CobolScript. The first screen of this sample program is shown in Figure 7.1.

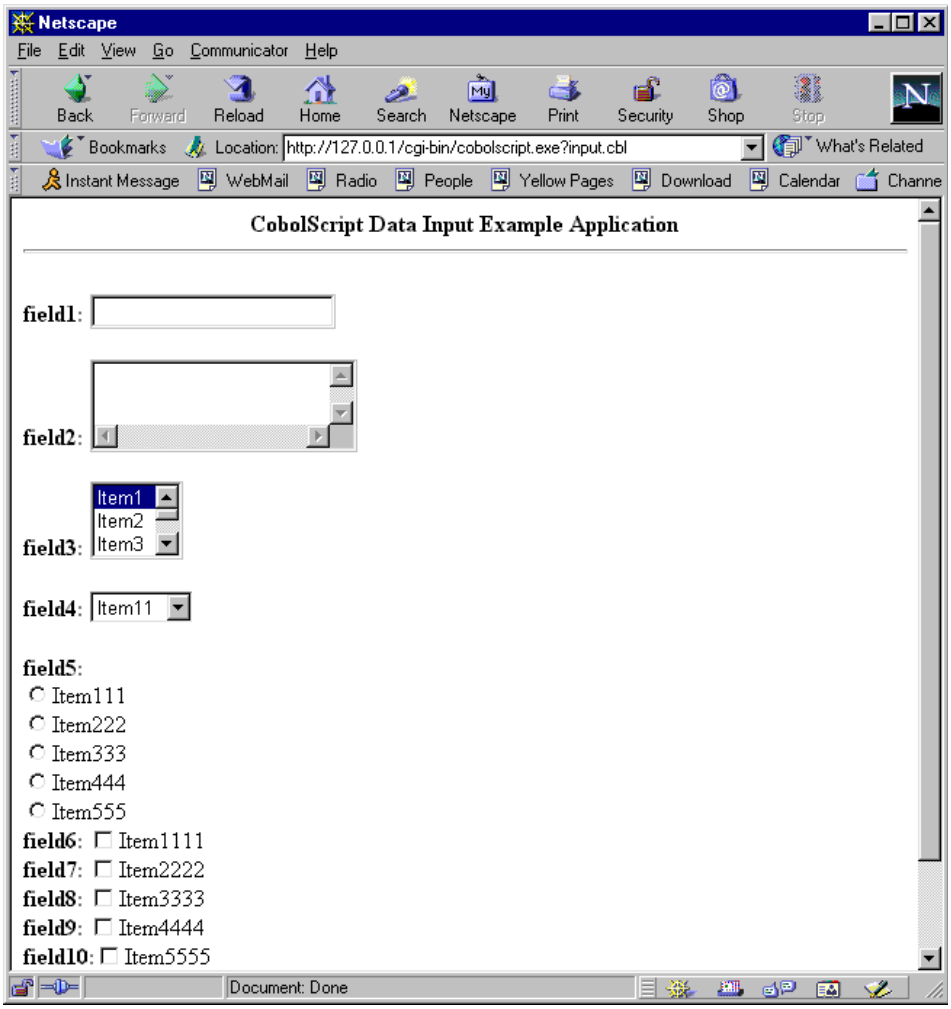


Figure 7.1 – Input.cbl sample program, as seen from Netscape browser.

All of the HTML form control tags that we discuss in this section have a common attribute called NAME. This attribute is of the form:

```
NAME=variable_name
```

Or, alternatively (quotes can be included or excluded from tag attribute values; they must be included, however, when spaces exist in the attribute's value):

```
NAME="variable_name"
```

where variable_name is the name of the CGI variable that will be passed to the receiving program when the form is submitted. The receiving CobolScript program, specified in the ACTION attribute of the FORM tag, must define a variable for each submitted form control with a NAME attribute in order for the ACCEPT DATA FROM WEBPAGE statement to work correctly.

Text Box Input

Text boxes are created with the <INPUT TYPE=TEXT… > tag. For instance, if our source CGI form submits a text input named field1, defined here:

```
<FORM ACTION="/cgi-bin/cobolscript.exe?receive.cbl">
<INPUT TYPE=TEXT NAME=field1><BR><BR>
<INPUT TYPE=SUBMIT>
</FORM>
```

Then, our receiving CobolScript program (which will be named receive.cbl, according to the ACTION attribute of the FORM tag above) must define a variable named field1:

```
1 field1  PIC X(20).
```

In the example program input.cbl, a text box named field1 is displayed to a web browser when the program is run. When the form is submitted from the web browser, the CobolScript program will get any data in the text box and place it in a CobolScript variable named field1.

Text boxes can also have preassigned values through the use of the VALUE attribute. In a more complex example than the one above, a CobolScript program we'll call recurse.cbl, that both displays a form and calls itself after accepting submitted input from the form, could have some code like the following:

```
DISPLAY `<INPUT TYPE=TEXT NAME=field1 VALUE="`.
IF field1 NOT = SPACES
   DISPLAY field1
END-IF.
DISPLAY `">`.
DISPLAY `<BR><BR>`.
DISPLAY `<INPUT TYPE=SUBMIT>`.
DISPLAY `</FORM>`.
```

In this case, the text box's VALUE will be assigned "" (a null value) if field1 is blank (all spaces); otherwise it will be assigned the value of the CobolScript variable field1.

Text Area Input

Text area controls are created with the <TEXTAREA … > tag:

```
<TEXTAREA NAME="field2" COLS=20 ROWS=2>
</TEXTAREA><BR><BR>
```

Our receiving CobolScript program must, in this case, define a variable named field2:

```
1 field2  PIC X(40).
```

In the example program input.cbl, a text area with the name field2 is displayed to the web browser when the program is run. When the form is submitted from the web browser, the CobolScript program will get any data in the text area and place it in a CobolScript variable named field2. Special characters like carriage returns and line feeds will be translated into HTML special characters such as
 and
 This is useful when you need to save the contents of a TEXTAREA to a file and later redisplay them in a web browser. The breaks and tabs will be preserved when you redisplay the HTML.

Because text areas can be large, and CobolScript variables are fixed width, you may find that you want a way to display only the initial populated portion of the text area input. HTML tends to ignore extra spaces, so it's not usually necessary to eliminate trailing spaces, but you may find it useful when working inside dynamically-sized HTML tables, since trailing spaces are taken into account when table elements are sized. The routine below accomplishes this with a PERFORM..VARYING loop and the use of positional string referencing:

```
PERFORM VARYING space_location FROM 40 BY -1
   UNTIL FIELD2(space_location:1) NOT = SPACE
END-PERFORM.

DISPLAY FIELD2(1:space_location)
```

List and Dropdown List Boxes

List and dropdown list boxes are displayed with the <SELECT … > tag inside an HTML form, like this:

```
<SELECT NAME="field3" SIZE=3>
<OPTION SELECTED>Item1
<OPTION>Item2
<OPTION>Item3
<OPTION>Item4
<OPTION>Item5
<OPTION>Item6
</SELECT><BR><BR>

<B>field4:</B>
<SELECT NAME="field4">
<OPTION SELECTED VALUE=Item11>Item 11
<OPTION VALUE=Item22>Item 22
<OPTION VALUE=Item33>Item 33
<OPTION VALUE=Item44>Item 44
<OPTION VALUE=Item55>Item 55
<OPTION VALUE=Item66>Item 66
</SELECT><BR><BR>
```

To retrieve these CGI fields, our receiving program defines two variables, one for each of the SELECT tags' names:

```
5 field3  PIC X(20).
5 field4  PIC X(20).
```

The SELECT tag is relatively straightforward: The value of each option is the text that immediately follows each OPTION tag, unless a VALUE attribute is specified in the OPTION tag. When an option is selected and the controlling form submitted, the SELECT variable is assigned the value of the option that was selected. By using a list box, you can limit the possible inputs that can be submitted, and therefore more readily direct processing based on these inputs. For instance, we could control processing based on the value assigned to field4 like this:

```
IF field4(1:6) = `Item33`
   DISPLAY `Item33 is currently out of stock`
ELSE
   DISPLAY field4 & ` has been ordered for you`
END-IF.
```

Radio Buttons

Radio buttons are created with the <INPUT TYPE=RADIO … > tag. They allow the selection of a single option from multiple options; when created properly, only one radio item may be selected from all radio buttons that have the same NAME value within a particular form. The VALUE tag is useful because you can put a compressed or numeric value in it and display a different text label in your web page, like this:

```
<INPUT TYPE=RADIO NAME=field5 VALUE=111>Item 111<BR>
<INPUT TYPE=RADIO NAME=field5 VALUE=222>Item 222<BR>
<INPUT TYPE=RADIO NAME=field5 VALUE=333>Item 333<BR>
<INPUT TYPE=RADIO NAME=field5 VALUE=444>Item 444<BR>
<INPUT TYPE=RADIO NAME=field5 VALUE=555>Item 555<BR><BR>
```

After the form is submitted, ACCEPT DATA FROM WEBPAGE will copy the VALUE of the radio button to the CobolScript variable with the same name as the buttons, so in this case, the following CobolScript field definition is required:

```
5 field5  PIC X(3).
```

Checkboxes

Checkboxes are created with the <INPUT TYPE=CHECKBOX … > tag. They allow multiple options to be selected or deselected from a group of options. When the form is submitted, those items that were checked will have their corresponding CobolScript variables populated with the VALUE specified for that check box item. Here is some example HTML for checkbox controls:

```
<B>FIELD6:</B> <INPUT TYPE=CHECKBOX NAME="field6"
VALUE="Item1111">Item1111<BR>
<B>FIELD7:</B> <INPUT TYPE=CHECKBOX NAME="field7"
VALUE="Item2222">Item2222<BR>
<B>FIELD8:</B> <INPUT TYPE=CHECKBOX NAME="field8"
VALUE="Item3333">Item3333<BR>
<B>FIELD9:</B> <INPUT TYPE=CHECKBOX NAME="field9"
VALUE="Item4444">Item4444<BR>
```

```
<B>FIELD10:</B><INPUT TYPE=CHECKBOX NAME="field10"
VALUE="Item5555">Item5555<BR>
<B>FIELD11:</B><INPUT TYPE=CHECKBOX NAME="field11"
VALUE="Item6666">Item6666<BR><BR>
```

Each checkbox field must have its own corresponding CobolScript variable, like this:

```
5 field6   PIC X(20).
5 field7   PIC X(20).
5 field8   PIC X(20).
5 field9   PIC X(20).
5 field10  PIC X(20).
5 field11  PIC X(20).
```

Using Hidden Fields

Hidden fields are actually just another type of CGI input, but they are special enough to warrant a section all their own. They are HTML form fields that are not visible in the browser window, but are still part of the underlying HTML form. They are useful for storing and passing information to the

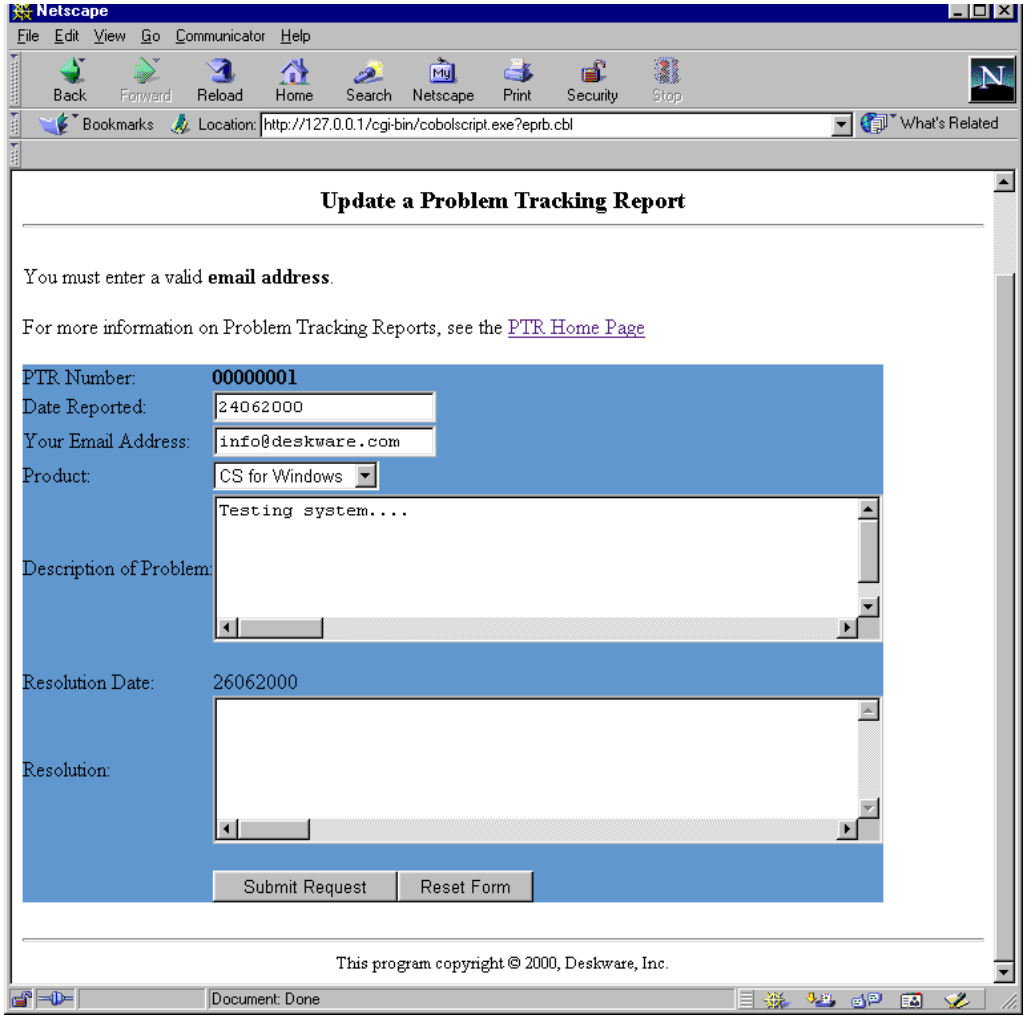


Figure 7.2 – Web page with hidden fields in the underlying HTML.

recipient program, and they can be used to maintain program continuity through a series of CobolScript-created pages without directly displaying all data to the browser window, and without writing to a temporary file. The sample problem tracking system uses hidden fields in the HTML forms it displays; figure 7.2 is a capture of the Update screen.

Figure 7.3 shows the HTML source to the screen in 7.2, complete with hidden HTML form fields. You can see the fields *update-record* and *record-key* have a TYPE="hidden".

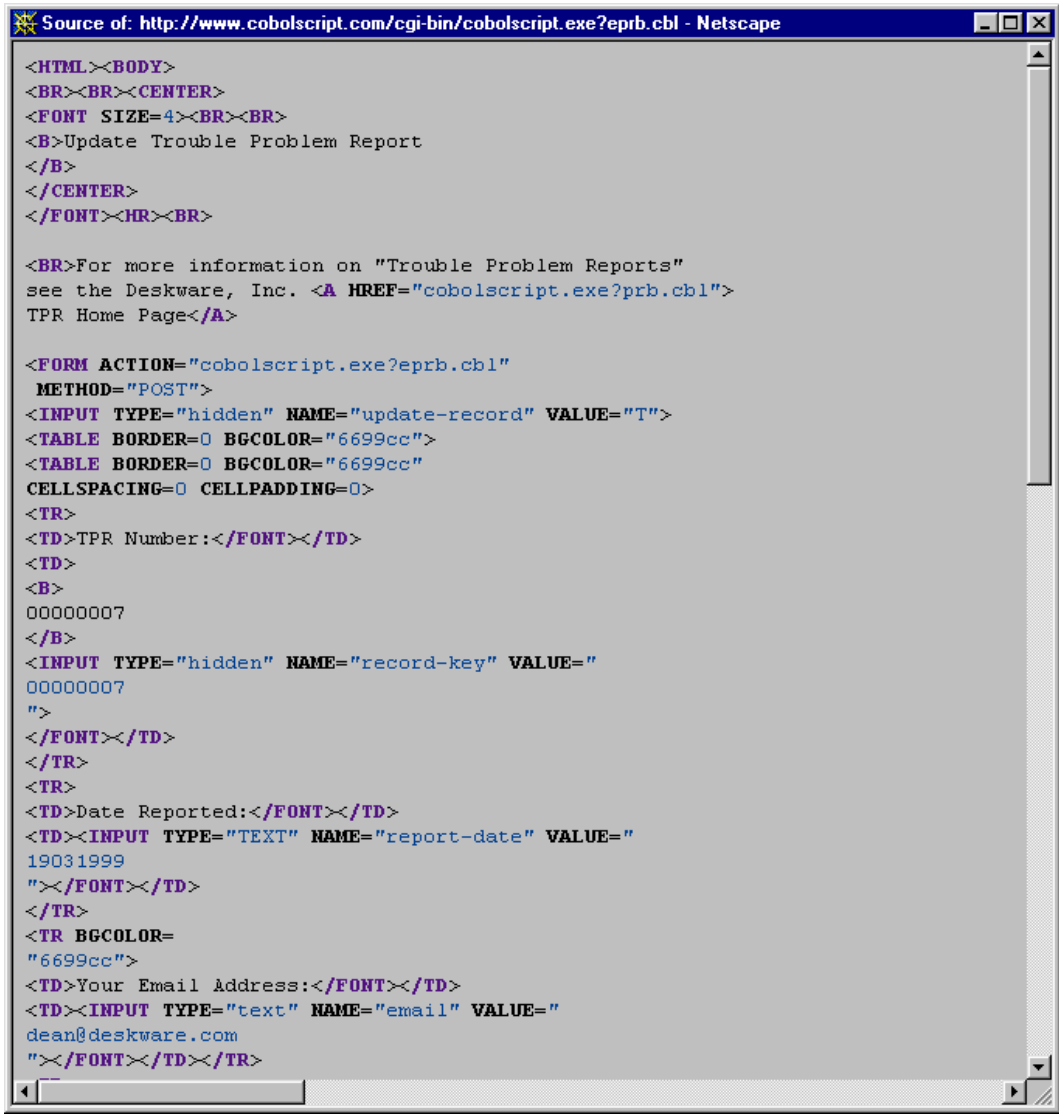


Figure 7.3 – HTML form with hidden fields, as seen from Netscape's source window.

When this form is submitted, the field *update-record* will pass a value of "T" to the CobolScript variable *update-record*. The form field *record-key* will pass a value of 00000007 to the CobolScript variable *record-key*. These fields are hidden on this form because we do not want them to be edited by the user. The *record-key* is used to determine which record needs to be updated after the form is submitted. This program is the Problem Tracking System example application (PRB.CBL) that comes with the sample programs included with CobolScript.

When you look at the source of the HTML form in Figure 7.3, you will notice that the hidden field *record-key* appears on three lines, like this:

```
<INPUT TYPE="hidden" NAME="record-key" VALUE="
00000007
```

```
">
```

The HTML is formatted in this way because we used DISPLAYLF to display the group level data item that contained the HTML. Had we used DISPLAY instead of DISPLAYLF, the entire text would have appeared on a single line. More on this below.

Here's a snippet of the CobolScript group item that contains the hidden field *record-key*. Here we spread the tag definition across three variables (two of these are implied FILLER variables, but variables nonetheless). This is a useful technique because it allows you to populate the CobolScript variable *record-key* with a value before displaying the group item:

```
5  `<INPUT TYPE="hidden" NAME="record-key" VALUE="`.
5  record-key    PIC 9(08).
5  `">`.
```

Sometimes when interfacing with other systems, particularly those written in Perl, the INPUT fields must be on a single line. In these cases, use DISPLAY than DISPLAYLF to print the relevant group item so that it prints on one line. At any rate, CobolScript is intelligent enough to process HTML forms that contain INPUT tags on single or multiple lines, so you won't encounter this issue unless you submit CGI data to non-CobolScript programs.

Sending Email from CobolScript Using CGI Form Input

As we discussed in Chapter 6, CobolScript has the capability to send simple emails, and this can easily be linked with data that has been submitted from a form, in order to create an auto-responder. The sample program email.cbl is an example of how to do this. The program is in the sample programs included with CobolScript. Figure 7.4 shows the application screen.

In email.cbl, email is sent using the SENDMAIL statement after fields corresponding to the to-address, from-address, subject, and message have been accepted from CGI input:

```
MOVE `yourservername.com` TO smtp_server.
SENDMAIL USING to_address
             from_address
             subject
             message server.
```

When sending an email with this command, you must be sure to supply a valid SMTP server name, which is the name of your sending mail server. CobolScript will then use this server to forward the email to the recipient.

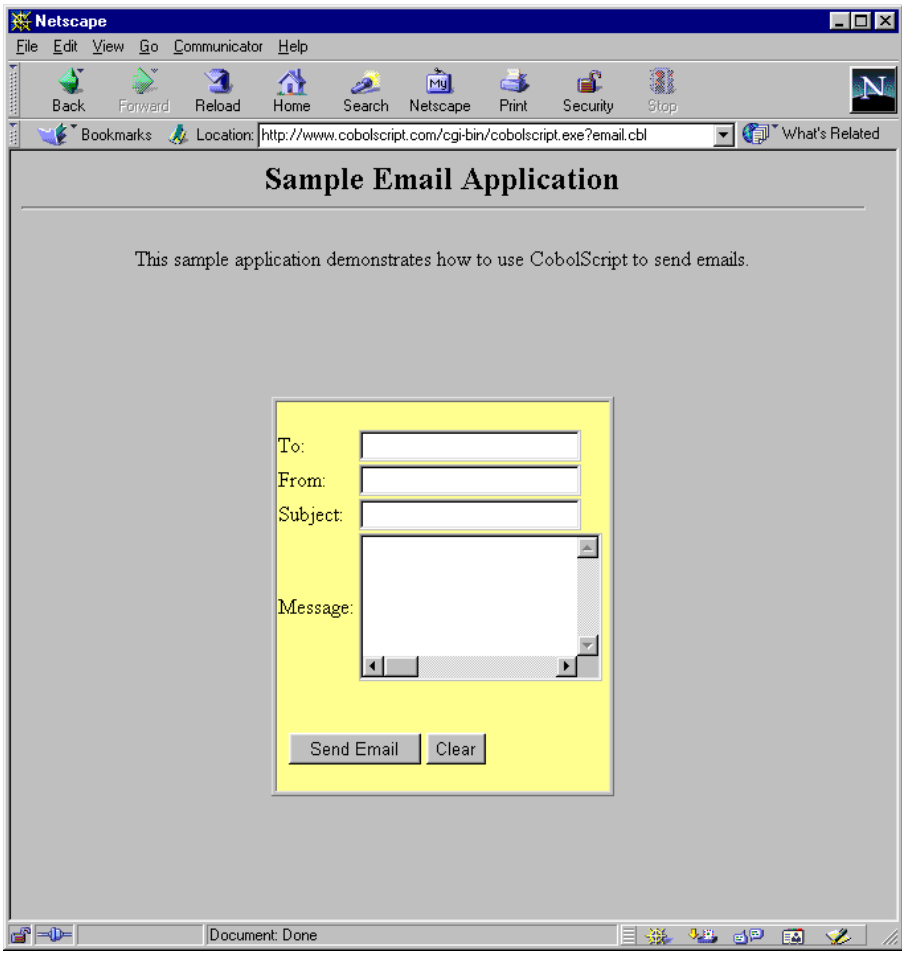


Figure 7.4 – The email.cbl sample application as seen in Netscape.

Using CobolScript to Transmit Files

Within HTML, you can provide links to files that can be downloaded by using the anchor tag (<A HREF= … >), but if you do this your users will be able to see the location of the file on your server when they view your HTML source. If you want to hide the location of your files and regulate who downloads files from your site, you can build a CobolScript program to directly send the file to the user's web browser.

CobolScript can be used to send a file to a client web browser. This is accomplished by sending the appropriate MIME header and then using either the DISPLAYFILE or DISPLAYASCIIFILE commands, depending on whether the file is binary or ASCII text. The user will be presented with a "Save As…" dialog box like the one in Figure 7.5, and will be allowed to save the file.

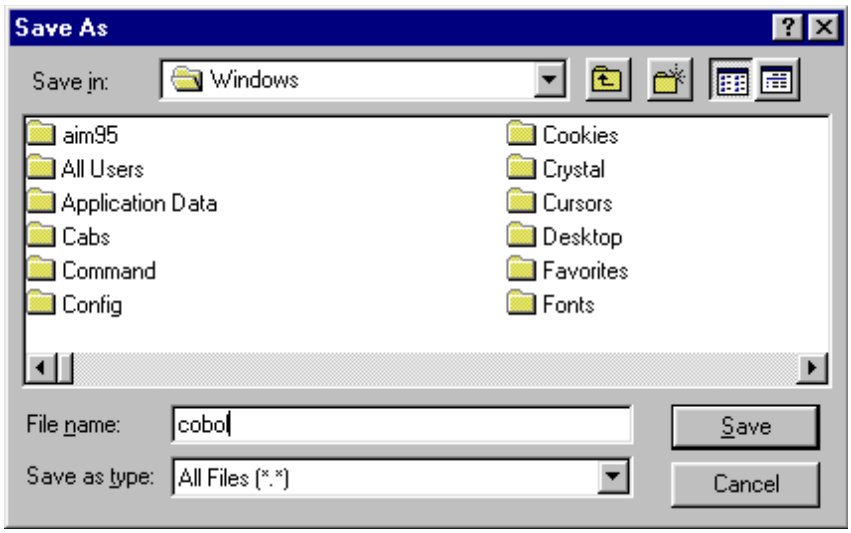


Figure 7.5 – The Save As… dialog box.

To use DISPLAYFILE or DISPLAYASCIIFILE, you should first build a program that displays a form that a user will submit when he wants to download a file. Within this form, specify the CobolScript program that will use the appropriate command to transmit the file. Typically this form will contain a submit button, and possibly some additional fields that you will use to validate the user, as in the following:

```
<FORM ACTION="/cgi-bin/cobolscript.exe?down.cbl" METHOD="POST">
<INPUT TYPE="hidden" NAME="user_id"   VALUE="md837653    ">
<INPUT TYPE="hidden" NAME="password_id" VALUE="83fFrR      ">
<INPUT TYPE="hidden" NAME="file"     VALUE="budgetfile  ">
<INPUT TYPE="Submit" VALUE="Download">
</FORM>
```

When this form is submitted, it will run the program you specify in the ACTION attribute of the FORM tag (down.cbl in this example). Your program can then accept authentication information and decide whether to transmit the file to that particular user based on this information. If you choose to not send the file, you can simply display an error page instead.

After you have validated the authentication information, you can begin transmitting the file to the user. There are two steps to this process. First, you will need to display a special MIME header. This mime header is what prompts the user's web browser to show the "Save As" dialog box. The file name that you use in your MIME header will be the default file name in the "Save As" dialog box. It is very important that the file size in your MIME header matches the **exact** file size of the file you wish to transmit; in bytes. If it doesn't, your file will not be transmitted correctly to the user.

After you have displayed the appropriate MIME header, you can use the DISPLAYFILE or DISPLAYASCIIFILE statement. This will transmit the contents of the file to the client's web browser after he selects the "Save" button from the "Save As…" dialog.

Here's a CobolScript code example with the appropriate MIME header and the DISPLAYFILE statement (DISPLAYASCIIFILE could be substituted for DISPLAYFILE below if the file to be transferred is a text file):

```
MOVE `budget.xls` to xfer_filename
MOVE `octet-stream` to xfer_method
MOVE 420000 TO xfer_filesize
DISPLAY `Content-type: application/` & xfer_method.
DISPLAY `Content-Disposition: inline; filename=`  & xfer_filename.
DISPLAY `Content-Description: ` & xfer_filename.
DISPLAY `Content-Length: ` & xfer_filesize.
DISPLAYLF.
DISPLAYFILE download_filename.
```

By using this technique, you can regulate downloads, and audit which users download your files. You can also build custom text files that will be sent to your users by displaying a MIME header and then displaying individual lines, one line at a time. If you do this, make certain that the amount of data you send matches the Content-Length specified in your MIME header.

Embedding JavaScript in CobolScript Programs

In some cases, you may want to have a portion of your application's processing take place on the client machine (the browser's computer). Client-side processing is useful for tasks like edit validations, because user feedback can be more real-time, and can be provided to a user prior to his submitting a form and reconnecting with the web server.

If you want to use client-side processing with CobolScript, we recommend you do it by embedding JavaScript in the HTML displayed by your CobolScript programs. JavaScript is relatively independent of browser manufacturer (it works with current versions of both Netscape® and IE®), runs on the client's web browser, and is very useful for basic data validation and checking. By embedding JavaScript-enriched HTML in your CobolScript applications, you can also reduce network traffic because checks can be performed on the data before it is submitted to the web for processing. If you're interested in using JavaScript, a pretty good (and reasonably priced) book for beginners is *JavaScript for the World Wide Web*, available from Peachpit Press.

Some situations where you might want to take advantage of JavaScript are those that require form

Figure 7.6 – JavaScript message box.

fields to be populated or data validation of numeric and alphabetic fields. Figure 7.6 provides an example of a message box generated by JavaScript upon a failed data validation.

The JavaScript function that displays this message box is listed below. It is a small function and can be easily embedded into a CobolScript program that displays HTML to a web browser.

```
function check_fields(form) {
  if
   (form.first_name.value==""||escape(form.first_name.value).match("%")
    != null){
      alert("You must enter an alphabetic first name.");
      form.first_name.focus();
      form.first_name.select();
      return false;
    }
}
```

CobolScript lends itself very well to displaying web pages, primarily because the nature of group-level data items allows entire HTML code segments to be isolated in your program (or in copybooks) in simple variable definitions. Because of this, you can create group items comprised of FILLER variables that contain your JavaScript code, and then just display the group level data item. By doing this, you can preserve the visual layout of your JavaScript code, and it will be relatively easy to debug from within your CobolScript program.

Following is an example of a group item named web_page_header that contains our JavaScript code from above:

```
1 web_page_header.
 5 `Content-type: text/html`.
 5 FILLER  PIC X    VALUE LINEFEED.
 5 `<HTML><HEAD><TITLE>Validate</TITLE>`.
 5 FILLER  PIC X    VALUE LINEFEED.
 5 `<SCRIPT LANGUAGE="JavaScript">`.
 5 `<!—Hide script from old browsers`.
 5 `   function check_fields(form) {`.
 5 `    if (form.first_name.value == ""`.
 5 `     || escape(form.first_name.value).match("%") !=null){`.
 5 `         alert("You must enter an alphabetic first name.");`.
```

```
5 `         form.first_name.focus();`.
5 `         form.first_name.select();`.
5 `         return false;`.
5 `         }`.
5 `   }`.
5 `   // End script hiding -->`.
5 `</SCRIPT>`.
```

Let's assume that we saved this variable definition, by itself, as a text file with the name HEADER.CPY. Then, this header and JavaScript are freely available to any CobolScript program, and including this file in any CobolScript program's variable definition is just a matter of using the COPY or INCLUDE statement to reference the copybook in your program code, like this:

```
COPY `HEADER.CPY`.

1  other_stuff   PIC 99.
.
.
.
```

Now, the header data can be displayed with this small piece of code:

```
DISPLAYLF web_page_header.
```

When this statement executes, all of the variables that comprise web_page_header above will be printed to standard output, which in this case means they'll be sent to the requesting client's browser window.

Breaking a web page document into separate group-level data items in this manner can make it very easy to maintain, and using copybooks to store these items can be a real timesaver when modifications to the group items have to be made.

Chapter

8

Programming Techniques and Advanced CobolScript® Features

In this chapter, we discuss the technique of modular program design, provide some detailed information on manipulating CobolScript variables using the MOVE statement, and discuss some advanced features that make CobolScript a truly unique programming language.

ICON KEY

➲ Important point

Designing a Modular Program

Modular programming is a way of organizing your program code to make the program easier to develop, understand, and maintain. A modular program is organized into paragraphs of code called *modules*. Modules are broken down into lines of code that perform one function or several closely related functions.

Modules are defined by paragraph names in the body of your CobolScript program. The names of your modules must start in column 8 and must be less than 80 characters in length. Your module names should also be descriptive, meaning, a module name should describe that module's function. It is also helpful to put a comment block right immediately before the module name. This should be a short description that a programmer can easily read in order to understand what the module does, and how it does it.

A program should be designed in a hierarchical fashion. Splitting a program up into modules facilitates the partitioning of logic into individual components that are easy to code and maintain. A program module should be as short as possible to perform a specific function in an independent manner. A good guideline is that a module should not be longer than one page of code.

To demonstrate the concept of modular programming we will create a program that displays a web page. The requirements of our programs are as follows:

- Print a header for our web page
- Print the body of our web page
- Print a footer for our web page

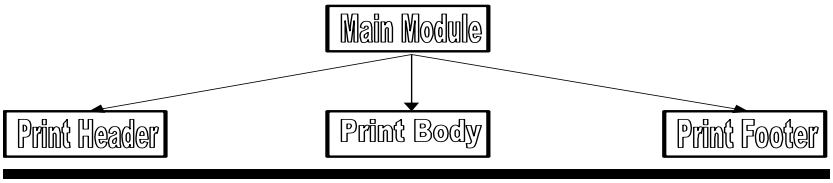


Figure 8.1 – Top-down design.

The requirements of our program can be easily broken down into a hierarchy. The hierarchy for our program is illustrated in Figure 9.1. This hierarchy can then be transformed into modules. We have named our modules relative to their function. They are as follows:

- MAIN
- PRINT-HEADER
- PRINT-BODY
- PRINT-FOOTER

MAIN is the main program module. It will call each of the three modules in sequence in order to display a simple web page with a horizontal rule at the top, the word "Deskware, Inc" formatted and centered in the page, and a horizontal rule at the bottom of the page.

The PERFORM statement controls the flow of the program. We PERFORM each of the three modules and then terminate program flow with the GOBACK statement. Below is a partial listing of our program (the sample program PAGE.CBL):

```
 MAIN.
    PERFORM PRINT-HEADER.
    PERFORM PRINT-BODY.
    PERFORM PRINT-FOOTER.
    GOBACK.

*****************************************************
* MODULE: PRINT-HEADER
* Prints header info for the html document.
*****************************************************
 PRINT-HEADER.
   DISPLAYLF `Content-type: text/html`.
   DISPLAY LINEFEED.
   DISPLAY `<HTML><BODY>`.
   DISPLAY `<HR>`.
   DISPLAY `<BR><BR><BR><BR><BR><BR>`.

*****************************************************
* MODULE: PRINT-BODY
* Prints body of HTML document
*****************************************************
 PRINT-BODY.
   1 company_name PIC X(n) VALUE `Deskware, Inc`.
```

```
    DISPLAY `<CENTER>`.
    DISPLAY `<FONT FACE="Impact" SIZE=7>` & company_name & `</FONT>`.
    DISPLAY `</CENTER>`.

*****************************************************
* MODULE: PRINT-FOOTER
* Prints trailer info for the HTML document
*****************************************************
 PRINT-FOOTER.
    DISPLAY `<BR><BR><BR><BR><BR><BR>`.
    DISPLAY `<HR>`.
    DISPLAY `</BODY></HTML>`.
```


This program is very simple and is meant only to illustrate modularity. It could have been written by using only one module instead of four. However, as your programs increase in size and complexity, modularity becomes increasingly important. Why? Quite simply, most of us aren't really capable of conceptualizing the intricate details of very large programs in our minds all at once. For this reason, dividing your code into modules allows conceptualization at different hierarchical levels, so that you as well as others will have an easier time creating and maintaining your code. Even when there is not much code in your program, dividing the logic up into modules can make it more readable. Also, modular code can easily be broken apart into separate copybook files later, allowing you to reuse particular pieces of code across programs using the COPY statement.

Manipulating CobolScript Variables

Basic Moves

Basic moves copy data from one variable to another or from a literal to a variable. The value on the left will be copied to the variable on the right:

```
MOVE `Deskware` TO name_var.
MOVE compnay_var TO name_var.
```

Segmented Moves

Segmented moves copy pieces of variables or *segments* to target variables (also known as a reference modification). The segmented move uses a variable name, a segment starting position, and length. It has the form of variable_name(start : length):

```
MOVE name(1:4) TO new_name.
MOVE `Desk` TO name(5:4).
```

Segmented moves can only be used on elementary items and are not allowed on group items. You can accomplish this same type of manipulation by moving a group item to another group item. The elementary items that are part of the target group item would simply have to have different picture lengths for each variable.

Elementary Item to Group Item Moves

Moving an elementary item to a group item is a great technique for parsing data. For example, if you have a variable that contains a 12 digit phone number you can parse it easily by moving it to a group item:

```
1 input_field PIC X(12).
1 phone_number.
  5 area_code PIC X(03).
  5 FILLER    PIC X.
  5 prefix    PIC X(03).
  5 FILLER    PIC X.
  5 exchange  PIC X(03).

MOVE input_field TO phone_number.
```

After this move has been executed, the three parts of the phone number will be placed in the variables area_code, prefix, and exchange.

Group Item to Elementary Item Moves

Moving a group item to an elementary item is a good way to build the contents of a variable. For example, by moving phone_number to input_field we can format a variable:

```
1 output_field PIC X(12).
1 phone_number.
  5 `(`.
  5 area_code PIC X(03).
  5 `)`.
  5 prefix    PIC X(03).
  5 `-`.
  5 exchange  PIC X(03).

MOVE `813`  TO area_code.
MOVE `555`  TO prefix.
MOVE `2494` TO exchange.
MOVE phone_number TO output_field.
```

The variable output_field will now have a value of `(813)555-1234`

Please refer to the sample program MOVE.CBL for more examples of moving variables.

Advanced CobolScript Features

These advanced CobolScript features are meant to add flexibility to your coding. Each is some feature not normally present in computer languages that are similar to CobolScript.

Expression Evaluation within the DISPLAY statement

It is possible to pass raw expressions to the DISPLAY statements as arguments. These expressions will be evaluated by the DISPLAY statement, and the result will display in a CobolScript-defined numeric format, with five post-decimal digits. Thus, the following is perfectly acceptable:

```
MOVE 2 TO radius.
DISPLAY `Area = ` & PI(0) * (radius^2).
```

And this will print the following to standard output:

```
Area = 12.56637
```

DISPLAYLF has this same capability.

Expressions as Segment Arguments and Occurs Clause Variable Arguments

Expressions can also be used as arguments to positional string references (also known as reference modification or segments), and as arguments to occurs clause variables, so long as each evaluates to an integer that is within the appropriate range. For example, the following code block that uses an expression in a positional string reference is a valid one (albeit a bit unusual):

```
1  var1   PIC X(30) VALUE `ABCDEFGHIJKLMNOPQRSTUVWXYZ1234`.
1  counter_var  PIC 999.

MOVE 24 TO counter_var.
DISPLAY `var1(2:24) = ` & var1(((counter_var/6)/2):counter_var-1+1).
DISPLAY `var1(2:24) = ` & var1(2:24).
```

The screen output for the above code block will be the following.

```
var1(2:24) = BCDEFGHIJKLMNOPQRSTUVWXY
var1(2:24) = BCDEFGHIJKLMNOPQRSTUVWXY
```

Both of these values are 24 characters long, beginning with the second character, of var1, since positional string referencing is always of the form:

```
string_variable_name(start_position : length)
```

The following code block that uses an expression in an OCCURS clause variable is also valid:

```
1  var1 OCCURS 4 TIMES  PIC XX.
1  counter_var  PIC 999.

MOVE 24 TO counter_var.
MOVE `WW` TO var1(counter_var/12).

DISPLAY `var1(2) = ` & var1(ROOT ((counter_var/6)^2, 4)).
DISPLAY `var1(2) = ` & var1(2).
```

The screen output for this code will be the following:

```
var1(2) = WW
var1(2) = WW
```

Intelligent Variable Parsing

As we mentioned briefly in the **Expressions and Conditions** section of Chapter 3, *CobolScript Language Constructs*, it is not necessary to separate individual expression components with spaces, so long as a parenthesis or simple (non-word) operator separates the variable or numeric components. However, since CobolScript allows dashes in variable names, and the symbol for the dash is the same symbol as the minus sign (-), expressions can be constructed where their meaning is uncertain. Take this expression, for example:

```
(WS-VAR-1+2)
```

If four variables have been defined in a program, one named WS, one named VAR, one named WS-VAR, and the other named WS-VAR-1, it's unclear which of the following is meant:

- The value in the variable WS-VAR-1, plus 2
- The value in the variable WS-VAR, minus 1, plus 2
- The value in WS, minus the value in VAR, minus 1, plus 2

The answer, for CobolScript, is that the first meaning (with the longest variable name) is always selected, if that variable name is defined. CobolScript uses an intelligent variable parsing algorithm to determine the value of a term like WS-VAR-1, and this algorithm prioritizes exact variable name matches over component subtraction. If WS-VAR-1 was not a defined variable, but WS-VAR, WS, and VAR still were, the second meaning above would then take precedence. Only in the case where WS-VAR-1 and WS-VAR had both not been defined, but WS and VAR had, would the expression evaluate to the third meaning.

As a result of this variable parsing, error messages related to undefined variables will sometimes name the undefined variable misleadingly. For example, if *none* of the above variables were defined, but you attempted to use the expression above in a statement, the error message would state that the variable *WS* had not been defined, rather than WS-VAR or WS-VAR-1. This is again because of the parsing algorithm; CobolScript attempts to find matches for smaller and smaller terms separated by dashes; when the term cannot be deconstructed any further (in this case, at the point when the term is WS) CobolScript stops and issues an error message. Since the line number of the error and the error message (indicating that a variable is undefined) are still correct, correcting this error is simply a matter of determining the variable name that *you* want defined, rather than what is indicated in the error message, and properly define it.

Dynamic File Naming

If you process many files of the same format and layout within a single program, you know that processing each file individually can be tedious and lengthy. To avoid this, you must reuse your file processing statements by placing them within a loop; but for this to work, the file name argument to your file processing statements, including the FD statement, must be dynamic. For this reason, we use the term *dynamic file naming*.

To dynamically name files, create a variable that will hold your file name, and then wait to create the FD for the file until after you've generated your file name. This works because there isn't an imposed order on statements in CobolScript programs. For instance:

```
* file name gldi variable definition
  1  file_name_gldi.
    5 FILLER      PIC X(n) VALUE `file`.
    5 counter PIC 99.
    5 FILLER      PIC X(n) VALUE `.dat`.

* file record definition
  1 file_record.
    5 field_1 PIC 99.
    5 field_2 PIC XX VALUE `AB`.

  PERFORM VARYING counter FROM 1 BY 1 UNTIL counter > 8
      FD file_name_gldi RECORD IS 4 BYTES
      OPEN file_name_gldi FOR WRITING
      PERFORM VARYING field_1 FROM 1 BY 1 UNTIL field_1 > 10
         IF field_1 > 5
            MOVE `CD` TO field_2
         END-IF
         WRITE file_record TO file_name_gldi
      END-PERFORM
      CLOSE file_name_gldi
  END-PERFORM.
```

The example above uses a counter variable to manipulate a numeric component of the dynamic file name, but the file names could also have been read from a file whose records contained the file names. The file names could also have been stored in an OCCURS variable, and the OCCURS index used as the counter variable to the outer PERFORM VARYING loop body.

Refer to the last code example of the next section for a more complex file naming example that makes use of the EXECUTE statement.

Dynamic Statement Creation and Execution

With most programming languages, the only dynamic components in a program at runtime are variables that store some type of value or point to a memory address. These variables can be examined and action taken based on their values, but the action itself (i.e., the code) must be created prior to runtime, and remains static throughout program execution.

In contrast, certain artificial intelligence languages like Prolog also provide the means to execute code statements that are created while the program is running. This is sometimes referred to as *dynamic programming*, which roughly means that code statements that are created by a program can then be executed by that same program.

CobolScript provides dynamic programming capability with the EXECUTE statement. The following code, for instance, has the net effect of displaying the literal "Hello, world.":

```
1 string_gldi.
 5 FILLER PIC X VALUE ACCENT.
 5 string_var PIC X(n) VALUE `Hello, world.`.
 5 FILLER PIC X VALUE ACCENT.
EXECUTE `DISPLAY ` string_gldi.
```

In the above example, the EXECUTE statement has two arguments, *`DISPLAY`* and *string_gldi*. Since string_gldi is a variable, the string that is actually processed by EXECUTE (and then directly executed by the CobolScript engine) is:

```
DISPLAY `Hello, world.`.
```

This is because all variable values are substituted prior to EXECUTE processing. Properly accounting for this substitution when using and understanding EXECUTE statements can be challenging until you become used to coding in this manner; the following code, which generates the same "Hello, world." output, illustrates this well:

```
1 string_var PIC X(n) VALUE `Hello, `.
EXECUTE `DISPLAY ` ACCENT string_var ACCENT ` & ` ACCENT `world.`
        ACCENT.
```

Of course, neither of the two examples above really demonstrates the utility of EXECUTE, since both execute a static DISPLAY statement that could have just as easily been coded directly. To uncover the real value of EXECUTE, we'll look at a more involved example that dynamically changes the name of the source variable in a MOVE statement that is the variable argument to EXECUTE:

```
1  move_exec.
  5  `MOVE line_`.
  5  num_position PIC 99.
  5  ` TO license_line_item`.

1 line_01 PIC X(7) VALUE `line111`.
1 line_02 PIC X(7) VALUE `line222`.
1 line_03 PIC X(7) VALUE `line333`.
1 line_04 PIC X(7) VALUE `line444`.
1 line_05 PIC X(7) VALUE `line555`.
1 license_line_item PIC X(7).

PERFORM UNTIL num_position = 5
   ADD 1 TO num_position
   DISPLAY `move_exec = ` & ACCENT & move_exec & ACCENT
   EXECUTE move_exec
   DISPLAY `license_line_item = ` & ACCENT & license_line_item & ACCENT
END-PERFORM.
GOBACK.
```

In this example, multiple MOVE statements are combined into a single EXECUTE statement inside a loop. The source variable component of the MOVE is dynamically changed from line_01 to line_02, line_03, line_04, and then line_05 because a portion of the source variable name is actually the value of the loop counter variable. This code produces the following output:

```
move_exec = `move line_01 to license_line_item`
license_line_item = `line111`
move_exec = `move line_02 to license_line_item`
license_line_item = `line222`
move_exec = `move line_03 to license_line_item`
license_line_item = `line333`
move_exec = `move line_04 to license_line_item`
license_line_item = `line444`
move_exec = `move line_05 to license_line_item`
license_line_item = `line555`
```

In the previous section, we examined a simple method to dynamically name files. If the file names vary considerably, however, naming them becomes more difficult than assigning a counter variable. An OCCURS variable can be used to store the different filenames, and then the OCCURS index used to retrieve each file name, but the OCCURS elements would still have to be assigned using individual MOVE statements. Using a text file to store and access the file names may work well for a large number of file names, but it can be overkill for a more modest number.

If the number of file names is relatively small, and you prefer keeping the list of file names inside the program that processes them, you can create a pseudo-array group item whose elementary members are the file names that you intend to process. Then, use the EXECUTE statement to perform a dynamic MOVE in order to reassign the file name variable, as in the following:

```
 1 file_name_list.
   5 file_name_01 PIC X(n) VALUE `first.dat`.
   5 file_name_02 PIC X(n) VALUE `second.dat`.
   5 file_name_03 PIC X(n) VALUE `third.dat`.
   5 file_name_04 PIC X(n) VALUE `fourth.dat`.
   5 file_name_05 PIC X(n) VALUE `fifth.dat`.
   5 file_name_06 PIC X(n) VALUE `sixth.dat`.
   5 file_name_07 PIC X(n) VALUE `seventh.dat`.
   5 file_name_08 PIC X(n) VALUE `eighth.dat`.

* file name target variable definition
  1  file_name_var PIC X(12).

* file record definition
  1 file_record.
    5 field_1 PIC 99.
    5 field_2 PIC XX VALUE `AB`.

* move statement to be executed
 1  move_exec.
  5  `MOVE file_name_`.
```

```
5  counter PIC 99.
5  ` TO file_name_var`.




PERFORM VARYING counter FROM 1 BY 1 UNTIL counter > 8
    EXECUTE move_exec
    FD file_name_var RECORD IS 4 BYTES
    OPEN file_name_var FOR WRITING
    PERFORM VARYING field_1 FROM 1 BY 1 UNTIL field_1 > 10
       IF field_1 > 5
          MOVE `CD` TO field_2
       END-IF
       WRITE file_record TO file_name_var
    END-PERFORM
    CLOSE file_name_var
END-PERFORM.
GOBACK.
```

Chapter

9

CS Professional CodeBrowser™, AppMaker™, and Control Panel

In addition to LinkMaker™ (discussed in appendixes G and H), CobolScript Professional Edition comes with several features not present in the Standard Edition that combine to make CS Professional a complete, enterprise-ready development solution. Using these additional features, you can create royalty-free, stand-alone executables from your CobolScript programs, browse your code using a colorizing utility, and administer your CobolScript environment.

Feature Requirements

CodeBrowser™ and the CobolScript Control Panel both require that you have web server software installed on your CS Professional-resident computer, and that the CobolScript engine be placed in your web server's cgi-bin directory. AppMaker™ can be run without a web server, using a specific command line option, or with a web server by using the Control Panel.

Additionally, the Control Panel can only be run from the machine on which CobolScript Professional and your web server are installed. This is done for security reasons.

Using CodeBrowser™

CodeBrowser™ is a code colorizing and viewing utility. CodeBrowser™ displays a colorized version of your program in a browser window, with a line number beside each line of code to assist you with the debugging process. Comments, keywords, and literals are each distinctly colorized in the browser.

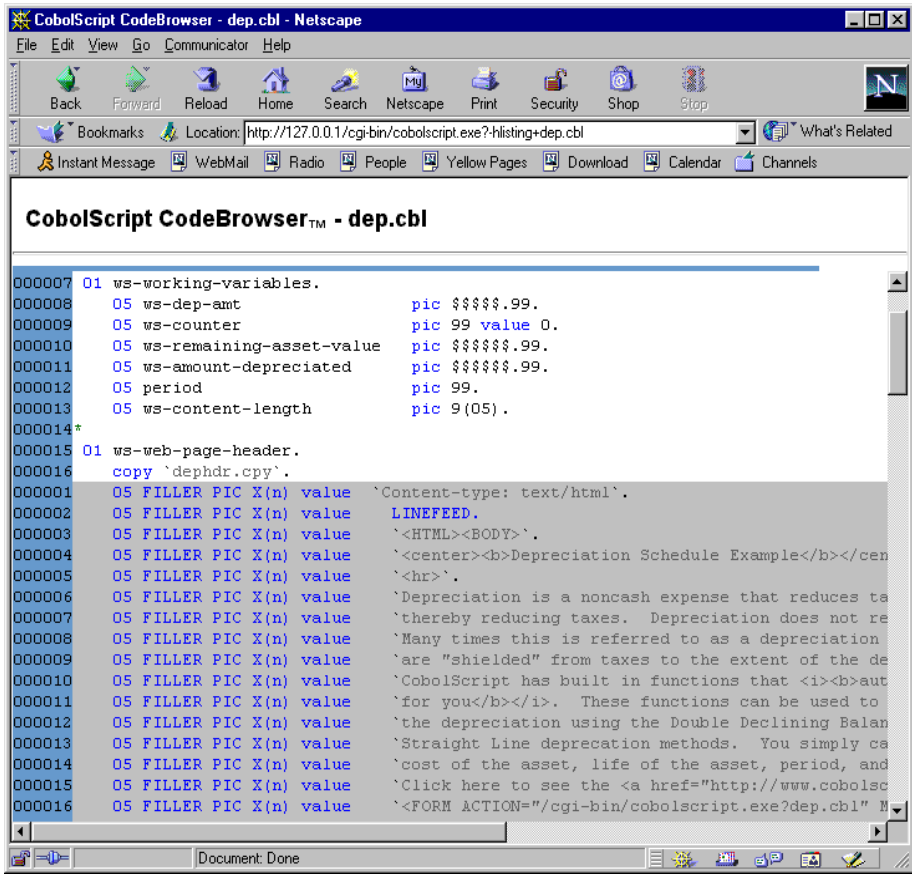


Figure 9.1 – Using CodeBrowser to browse a program that contains a copybook.

Copybooks that are included in your program appear as inline code in the CodeBrowser™ listing; they are differentiated with a gray background. Including copybook code in the CodeBrowser™ listing helps to provide a cohesive view of your entire program, and more meaningful code printouts and documentation.

The .csaccess File

In order for you to use CodeBrowser™, a file named *.csaccess* must exist in your web server's cgi-bin directory. CodeBrowser™ program listings may only be viewed for those CobolScript programs that have an entry in the *.csaccess* file. The contents of this file are the names (and relative paths, if any) of the programs that you wish to be made available for browsing, with a linefeed separating each program name. However, rather than creating and editing this file directly, you can use the Control Panel to administer *.csaccess*. See the section on the Control Panel later in this appendix for more information.

Anyone with access to your web site will be able to view CobolScript programs that have been added to the *.csaccess* file. This feature is useful for programming teams in different locations that are sharing development and test servers; these teams only have to enter the appropriate URL in their web browser to see a CobolScript program that resides on the server (see URL section below).

Before going live with an application, you should directly edit the *.csaccess* file and remove any entries for programs that you do not wish be made publicly visible with CodeBrowser™. You can also simply delete the contents of the file, which will prevent browse access on all programs. Anyone attempting

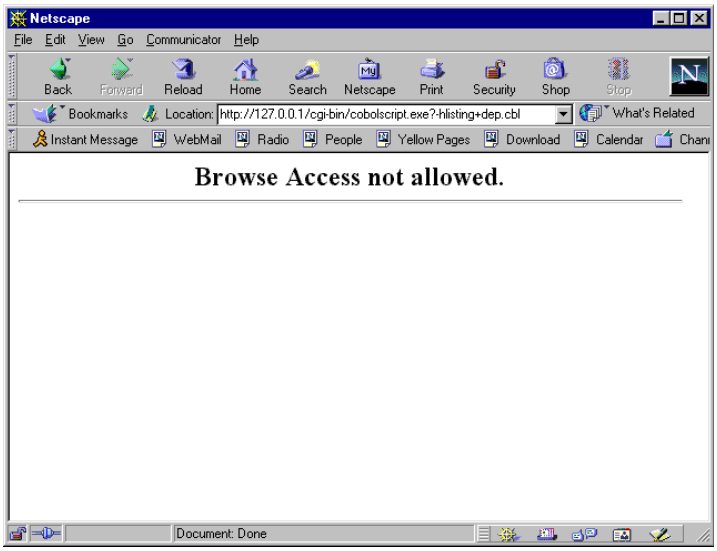


Figure 9.2 – CodeBrowser "Browse Access not allowed" screen.

to browse a program listing will be presented with a 'Browse Access not allowed' window as shown in Figure 9.2.

Running CodeBrowser™ from a URL

Once the *.csaccess* file has been configured, just enter the following URL (modified for your environment and program name) in your web browser to examine a program using CodeBrowser™:

```
http://<server-name>/cgi-bin/cobolscript.exe?-hlisting+<program-name>
```

Here, *server-name* refers to the host name or IP address of your CobolScript/web server machine, and *program-name* refers to the full name and relative path, if required, of your CobolScript program. In the following example, CodeBrowser will bring up a listing for the sample program *mail.cbl* on the server *www.cobolscript.com*, so long as *mail.cbl* is a valid entry in *.csaccess*:

```
http://www.cobolscript.com/cgi-bin/cobolscript.exe?-hlisting+mail.cbl
```

Of course, you can also link to this form of URL from other web pages or from HTML output of CobolScript programs. An HTML link for the program above could look like the following:

```
<A HREF="http://www.cobolscript.com/cgi-bin/cobolscript.exe?-
hlisting+mail.cbl">View Mail Program</A>
```

CodeBrowser™ can also be run from the CobolScript Control Panel. See the section on the Control Panel later in this appendix for more information.

Building Executables with AppMaker™

CS Professional provides the capability to create stand-alone executables from CobolScript programs using AppMaker™. This gives you the opportunity to sell or redistribute your CobolScript applications without disclosing your code, and without requiring that your customers purchase their own CobolScript license from Deskware (as is the case with CobolScript Standard Edition). You might also choose to build executables for an internet system, and then place those executables on your production web server, rather than placing raw code files on a production machine.

Executables can be built directly from the command line with the following syntax:

```
cobolscript.exe -b <program-name>
```

If your program successfully loads, an executable will be created from it and placed in the working directory. For example, typing the following will create an executable named *test.exe* in the working directory:

```
cobolscript.exe -b test.cbl
```

You can also build executable files by typing a specific URL into your web browser. This URL has the following format.

```
http://<server-name>/cgi-bin/cobolscript.exe?-b+<program-name>
```

Here, *server-name* refers to the host name or IP address of your CobolScript/web server machine, and *program-name* refers to the full name (and relative path, if required) of your CobolScript program. In the following example, an executable will be created for *write.cbl* on the server *127.0.0.1*:

```
http://127.0.0.1/cgi-bin/cobolscript.exe?-b+write.cbl
```

After the executable has been built, you will a web page similar to Figure 9.3. You can run the executable by clicking on the hyperlink that appears on the page.

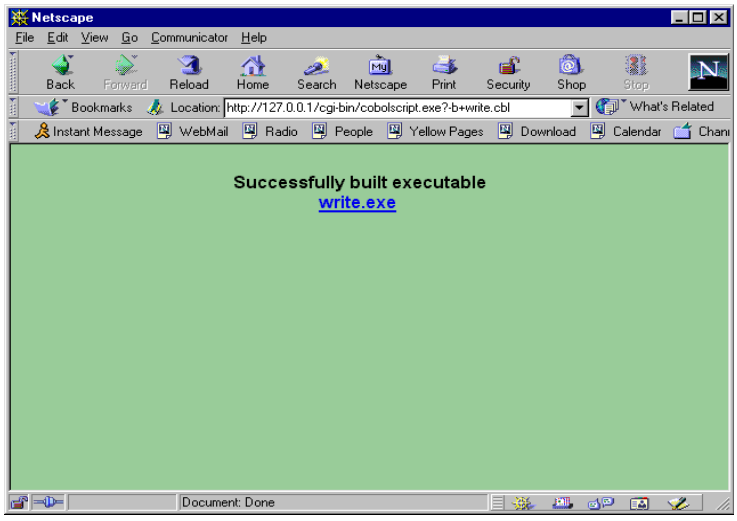


Figure 9.3 – Building an AppMaker executable from a web browser's URL.

AppMaker™ can also be run from the CobolScript Control Panel. See the section on the Control Panel later in this appendix for more information.

Using the CobolScript Control Panel

The CobolScript Control Panel is an administrative utility that is available only in CS Professional. The Control Panel provides access to other features of CS Professional, giving you the ability to run your CobolScript programs, browse your code, and build executables, all from within a visual environment.

In order for the Control Panel to work correctly, you must have web server software installed on your CS Professional computer, and the CobolScript engine must be located in the web server's cgi-bin directory. Also, for security reasons, the Control Panel may only be started from the machine on which CS Professional is installed.

To access the Control Panel, start a web browser and type in the following URL:

```
http://<server-name>/cgi-bin/cobolscript.exe
```

Here, *server-name* refers to the host name or IP address of your CobolScript/web server machine.

Most computers are configured with a 'loopback' value to refer to their own IP address. Since this address is often 127.0.0.1, the following URL will start the Control Panel on most web server machines with CobolScript installed in the cgi-bin directory:

```
http://127.0.0.1/cgi-bin/cobolscript.exe
```

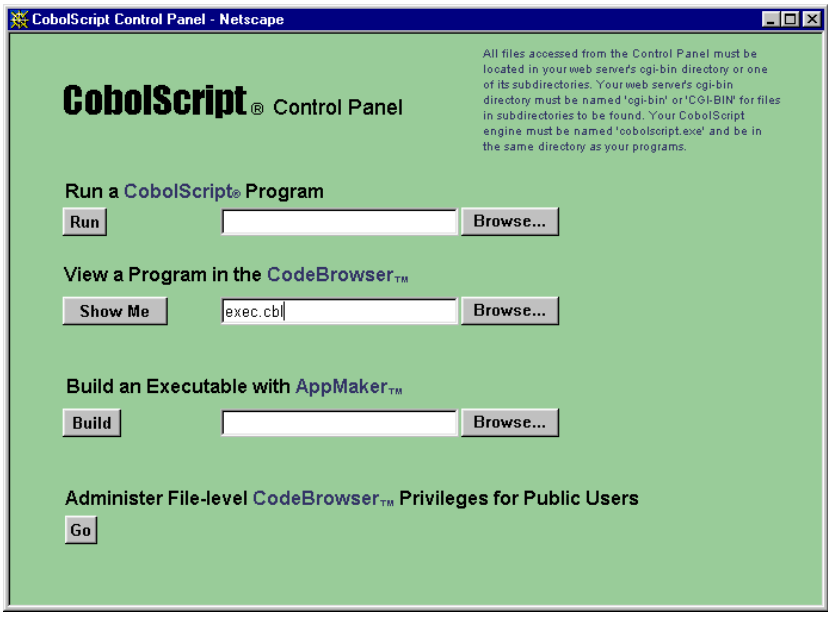


Figure 9.4 – CobolScript Control Panel.

Once you've submitted the appropriate URL, the CobolScript Control Panel will appear in a new window (see Figure 9.4). The following subsections explain Control Panel functionality.

Running a CobolScript program from the Control Panel

To run a CobolScript program from the Control Panel, enter the name of the program in the input box next to the *Run* button, or select the program by clicking on the *Browse* button to browse your filesystem. Once you've selected a program file, click *Run* to execute the program. This will allow you to run any CobolScript program that is in your web server's cgi-bin directory and that is designed to run through a web server (e.g., it displays correct MIME header information and HTML output).

Accessing CodeBrowser™ from the Control Panel

To run CodeBrowser™ from the Control Panel, enter the name of the program in the input box next to the *Show Me* button, or select the program by clicking on the *Browse* button to browse your filesystem. Once you've selected a program file, click *Show Me*. This will bring up a new window that contains a CodeBrowser™ listing of your program. Note that the program name must be in the *.csaccess* file for browsing to be permitted; see below for instructions on administering this file through the Control Panel.

Administering File-level CodeBrowser™ Privileges

CodeBrowser™ program listings may only be viewed for those CobolScript programs that have an entry in the *.csaccess* file. You can add these entries to *.csaccess* by clicking on the *Go* button from the Control Panel, which will open a new window called 'Administer Public CodeBrowser File Access' (see Figure 9.5).

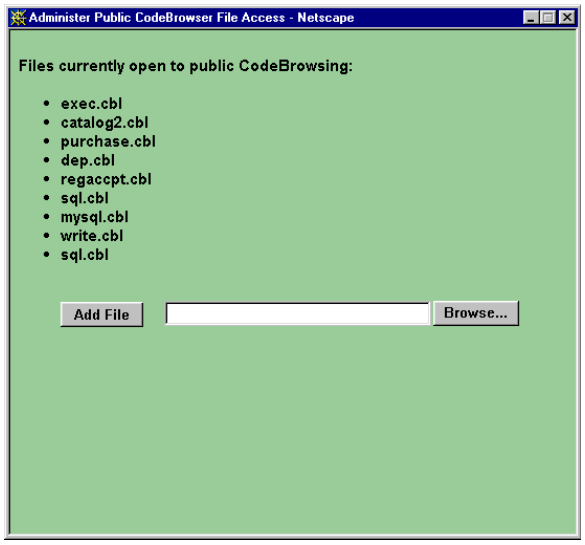


Figure 9.5 – Administering CodeBrowser privileges.

In this new window, you can add program files to *.csaccess* by entering the name of the file in the input box or by selecting the program by clicking on the *Browse* button, and then clicking on the *Add File* button. After you've finished adding files, simply close the window.

To remove public browsing capabilities on a program, you must directly edit the *.csaccess* file and manually remove the entry for the program you want to restrict. You can also delete the *.csaccess* file, which will prevent browse access on all programs.

Using AppMaker™ from the Control Panel

To use AppMaker™ to build an executable from the Control Panel, enter the name of the program in the input box next to the *Build* button, or select the program by clicking on the *Browse* button to browse your filesystem. Once you've selected a program file, click *Build*. A popup window will appear that shows that the executable was successfully built. Provided your application is designed for the web, you can then run the executable from the popup by clicking on the hyperlink. See Figure 9.6.

Figure 9.6– Creating an executable with AppMaker from the Control Panel.

Appendix

A

Language Reference

This appendix gives a detailed description of the command syntax used by CobolScript. For more information on specific components of CobolScript programs other than commands, such as variables, literals, and expressions, see Chapter 3, *CobolScript Language Constructs*.

ICON KEY

File I/O
Email

Usage for most of the commands listed in this appendix is demonstrated in one of the sample programs included with CobolScript. The sample programs are available for download from the Deskware Registered Developer Home Page – just login at www.cobolscript.com/cobolscript.exe?login.cbl using your Registered Developer ID and download the sample-programs-only file. A complete listing of these sample programs appears in Appendix D, *Sample CobolScript Program Files*.

Syntax and Description of Commands

Below is a legend that describes how the commands are documented.

Command:	***Command name***
Syntax	***Example syntax for a command.*** ***Variables and literals are enclosed in greater than/less than signs, e.g.,*** **<variable>** ***Optional syntax is enclosed in brackets, e.g.,*** **[ROUNDED]**
Description:	***Detailed description of what the command does***
Example Usage:	***Example illustrating the actual use of the command***
See Also:	***Other commands that are related to this command***
Sample Program:	***Filename of sample program that demonstrates the use of this command.***

Figure A.1 – The format of the command reference.

ACCEPT

Command:	**ACCEPT**
Syntax:	**Variant 1:** ACCEPT <accept-variable> FROM DATE. ACCEPT <accept-variable> FROM DAY. ACCEPT <accept-variable> FROM DAY-OF-WEEK. ACCEPT <accept-variable> FROM TIME. **Variant 2:** ACCEPT <accept-variable> FROM KEYBOARD [PROMPT <prompt-string>]. **Variant 3:** ACCEPT DATA FROM WEBPAGE.
Description:	The ACCEPT command has three variants: **Variant 1:** The basic variant of ACCEPT can be used to populate a numeric *accept-variable* with one of a number of variations of the current system date/time. The formats of the data returned to *accept-variable* by each of the date/time keywords are as follows: **Keyword** — **Format Mask** **DATE** — **DDMMYYYY**, where *DD* is the day of the month, ranging from 01 to 31, *MM* is the month of the year, ranging from 01 to 12, and *YYYY* is the four-digit year. **DAY** — **YYDDD**, where *YY* is a two-digit year code, and *DDD* is a day of the year ranging from 001 to 366. **DAY-OF-WEEK** — **d**, where *d* = 0 means Sunday, *d* = 1 means Monday, etc. **TIME** — **hhmmss**, where *hh* corresponds to hour of the day and ranges from 00 to 23, *mm* corresponds to minutes past the hour and ranges from 00 to 59, and *ss* corresponds to seconds past the minute and ranges from 00 to 59. **Variant 2:** ACCEPT <accept-variable> FROM KEYBOARD can be used to read a line from the standard input stream (normally the KEYBOARD) and store it in an alphanumeric *accept-variable*. When an ACCEPT FROM KEYBOARD command is processed, program flow is suspended until a line of keyboard input has been received. If the PROMPT clause is specified, *prompt-string* will display to standard output prior to the cursor prompt. Program execution is resumed when a line of standard input is terminated with a linefeed character; however, the linefeed character is not included in *accept-variable*. If the standard input stream is greater than the length of *accept-variable*, the data will be right-truncated. This variation of the ACCEPT command is also useful for getting raw, unparsed CGI (Common Gateway Interface) data from web pages. This is necessary for retrieving data from GET-method CGI form submissions, or for examining the raw input stream from POST-method submissions. Normally, however, ACCEPT DATA FROM WEBPAGE should be used for POST-method data retrieval – see below for more information. **Variant 3:** The ACCEPT DATA FROM WEBPAGE statement will accept CGI data from an HTML form that was submitted using the POST method, parse it, and place the contents in corresponding CobolScript variables. For this statement to work successfully, use the same field names in the receiving CobolScript program as are in the submitting POST-method CGI form. The ACCEPT DATA FROM WEBPAGE statement will then populate these CobolScript variables with the values that are in the incoming, like-named CGI variables; no additional parsing logic is required.

Command:	**ACCEPT**
	Refer to Chapters 6 and 8 for a more in-depth discussion of ACCEPT DATA FROM WEBPAGE.

Example Usage:	**Variant 1:**
	`ACCEPT date FROM DATE.`
	`ACCEPT day FROM DAY.`
	`ACCEPT day_of_week FROM DAY-OF-WEEK.`
	`ACCEPT time FROM TIME.`
	
	Variant 2:
	`ACCEPT stdin_var FROM KEYBOARD PROMPT `Enter input: `.`
	`ACCEPT raw_buffer FROM KEYBOARD.`
	
	Variant 3 (assumes two incoming CGI variables named cust_nm **and** order_nbr**):**
	`1 cust_nm PIC X(50).`
	`1 order_nbr PIC 9(10).`
	
	`ACCEPT DATA FROM WEBPAGE.`

Sample Program:	ACCEPT.CBL

ACCEPTFROMSOCKET

Command:	**ACCEPTFROMSOCKET**
Syntax:	ACCEPTFROMSOCKET USING <socket-number> <accept-socket-number>.
Description:	ACCEPTFROMSOCKET creates a new TCP/IP socket connection on *accept-socket-number* when a remote machine attempts to connect using a particular socket *socket-number*. *Socket-number* refers to the socket that has already been created in order to listen for a connection; when a remote computer attempts to connect on that socket, the ACCEPTFROMSOCKET command will accept the connection and create a newly connected socket on *accept-socket-number*. The ACCEPTFROMSOCKET command will cause CobolScript to suspend program flow until a socket connection is successfully established with a remote computer. After the new socket connection has been established, *socket-number* is freed and is ready to listen for another connection. This command is conventionally used only on the machine that is considered to be the server in two-way socket connections. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`ACCEPTFROMSOCKET USING socket_num_var`
	`connctd_socket_num_var.`
	
	The ACCEPTFROMSOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`5 TCPIP-RETURN-CODE PIC 9(07).`
	`5 TCPIP-RETURN-MESSAGE PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

Command:	**ACCEPTFROMSOCKET**	
See Also:	BINDSOCKET CLOSESOCKET CONNECTTOSOCKET CREATESOCKET	LISTENTOSOCKET RECEIVESOCKET SENDSOCKET SHUTDOWNSOCKET
Sample Program:	SERV.CBL	

ADD

Command:	**ADD**
Syntax:	**Variant 1:** ADD <number or variable> … TO <target-variable> [ROUNDED] **Variant 2:** ADD <number or variable> … TO <number or variable> GIVING <target-variable> [ROUNDED]
Description:	**Variant 1** of the ADD statement is used to add one or more numeric literals and/or numeric variables together, storing the result in the numeric *target-variable*. All literals and variables are added together to produce the result, including the value of *target-variable* prior to the addition. **Variant 2** of ADD is used to add one or more numeric literals and/or variables together, with the result stored in a *target-variable* whose original contents are not considered in the addition. Thus, if var has an initial value of 1, performing the operation: ADD 1 TO 1 GIVING var. will place a value of 2, not 3, into var. Both forms of ADD permit the use of the ROUNDED keyword, which rounds the target variable, after computation, to the nearest integer.

Example Usage:	**Variant 1:**
	`ADD 1 TO num_variable.`
	`ADD 1  2  3 TO num_variable.`
	`ADD var TO total.`
	`ADD 1.11  2  var  TO total ROUNDED.`
	
	Variant 2:
	`ADD value TO subtotal GIVING total.`
	`ADD 9.99  value TO subtotal GIVING total ROUNDED`

See Also:	COMPUTE SUBTRACT MULTIPLY DIVIDE
Sample Program:	ADD.CBL

BANNER

Command:	**BANNER**
Syntax:	BANNER USING <banner-input> <banner-character-input>
Description:	The BANNER command displays a Unix-style banner to the screen. The contents of *banner-input* are the large characters of the banner; the contents of *banner-character-input* are the component characters of the banner, which are the small characters used to make the banner letters. If *banner-character-input* is equal to a single space (`` ` `` or the SPACE keyword), the component character of each large letter will be a smaller version of itself, e.g., `BANNER USING `TEST` SPACE`

Command:	**BANNER**

	will generate the following screen output:
	
	`TTTTTTT EEEEEEE  SSSSS  TTTTTTT`
	`   T    E       S     S    T`
	`   T    E       S          T`
	`   T    EEEEE    SSSSS     T`
	`   T    E             S    T`
	`   T    E       S     S    T`
	`   T    EEEEEEE  SSSSS     T`

Example Usage:	`BANNER USING `TEST`  `#`.`
	
	`BANNER USING `TEST`  ` `.`
	
	`BANNER USING `TEST` SPACE.`
	
	`BANNER USING banner_contents banner_char.`

See Also:	GETBANNER
Sample Program:	BANNER.CBL

BINDSOCKET

Command:	**BINDSOCKET**
Syntax:	BINDSOCKET USING <socket-number> <port-number>.
Description:	The BINDSOCKET command binds a socket *socket-number* to a specific TCP/IP port *port-number* on the local machine. After this command is executed , the operating system will associate *port-number* with *socket-number*. This command is conventionally used only on the machine that is considered to be the server in two-way socket connections. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`BINDSOCKET USING socket_num_var port_num_var.`
	
	The BINDSOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE       PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE    PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET CLOSESOCKET CONNECTTOSOCKET CREATESOCKET LISTENTOSOCKET RECEIVESOCKET SENDSOCKET SHUTDOWNSOCKET
Sample Program:	SERV.CBL

CALENDAR

Command:	**CALENDAR**
Syntax:	CALENDAR USING <year-input> <month-input>.
Description:	The CALENDAR command displays a calendar for a given year *year-input* and month *month-input*. The *year-input* and *month-input* should be numeric values; if they are variables, their variable declarations must have numeric picture clauses. Any fractional component to *year-input* or *month-input* will be ignored, e.g., a *year-input* of 1957.75 will be processed as 1957. CALENDAR does not support pre-Julian calendar dates, i.e., any date prior to August 1752.

Example Usage:	`CALENDAR USING 2001 1.`
	
	`CALENDAR USING year_var month_var.`

See Also:	GETCALENDAR
Sample Program:	CALENDAR.CBL

CALL

Command:	**CALL**
Syntax:	CALL <system-command-literal | variable> <system-command-literal | variable>… .
Description:	CALL is used to call a shell command. Essentially, *system-command-literal* or the contents of v*ariable* are executed at the operating system's command prompt. Multiple arguments may be specified for a CALL command, and group items may be used as CALL arguments. CALL is an extremely powerful and versatile command, so use caution when implementing a program that uses CALL, especially when that program receives data from web input or other unauthorized user input. It's generally inadvisable to perform a CALL on any user input value that has not first been validated or examined by your program, since CALL provides access to operating system commands.

Example Usage:	**Example with one literal argument:**
	`CALL \`dir *.txt\`.`
	
	Example with one variable argument:
	`MOVE \`ls *.tmp\` TO system_command.`
	`CALL system_command.`
	
	Example with one literal and one variable argument:
	`MOVE \`*.cbl\` TO wildcard_variable.`
	`CALL \`ls -l \` wildcard_variable.`
	
	Example with gldi variable argument:
	`1 system_command.`
	` 5 \`ls\`.`
	` 5 \` *.tmp\`.`
	
	`CALL system_command.`

Sample Program:	CALL.CBL

CLOSE

Command:	**CLOSE**
Syntax:	CLOSE <filename>.
Description:	The CLOSE command is used to close a text data file *filename* that was previously opened with the OPEN statement.

Command:	**CLOSE**

Example Usage:	`CLOSE `TEST.DAT`.`
	
	`CLOSE test_file.`

See Also:	FD OPEN POSITION READ REWRITE WRITE
Sample Program:	IO.CBL

CLOSEDB

Command:	**CLOSEDB (*CobolScript Professional Edition Only*)**
Syntax:	CLOSEDB USING <return-code-variable>.
Description:	The CLOSEDB command closes an open LinkMaker™ database connection and populates *return-code-variable* with an integer value of 1 (success) or 0 (failure). This command is used after a connection has been established with a data source using the OPENDB command. See Appendixes G and H for more information about configuring and using LinkMaker™.
Example Usage:	`CLOSEDB USING ret_code.`
See Also:	OPENDB, EXEC SQL
Sample Program:	SQL.CBL

CLOSESOCKET

Command:	**CLOSESOCKET**
Syntax:	CLOSESOCKET USING <socket-number>
Description:	The CLOSESOCKET command closes the specified TCP/IP socket connection *socket-number*. It should only be called after the SHUTDOWNSOCKET command has been issued, to ensure a graceful socket termination. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`CLOSESOCKET USING socket_num_var.`
	
	The CLOSESOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET BINDSOCKET CONNECTTOSOCKET CREATESOCKET LISTENTOSOCKET RECEIVESOCKET SENDSOCKET SHUTDOWNSOCKET
Sample Program:	SERV.CBL

COMPUTE

Command:	**COMPUTE**
Syntax:	COMPUTE <compute-variable> [ROUNDED] = <expression>.

Description:	The COMPUTE statement is used to evaluate a normal mathematical *expression*, and place the result in *compute-variable*. Refer to the Expressions and Conditions section of Chapter 3, *CobolScript Language Constructs*, for details on the various forms that expressions are permitted to take.
	
	COMPUTE also supports the use of functions; see Appendix B, *Function Reference*, for complete details on the functions supported.
	
	The use of alphanumeric variables or string literals in a COMPUTE statement is illegal. Also, only one variable can be acted upon at a time in a CobolScript COMPUTE statement. This means that multiple assignment statements must be used to assign multiple variables.
	
	To identify size errors (encountered when a COMPUTE result is larger than the target variable's picture clause permits) first check the expression result in a condition, since size errors do not cause direct program errors. For instance, the following three statements will place a value of 11 in num_variable without causing a direct program error:
	
	`1 num_var PIC 99 VALUE 0.`
	`1 increment_var PIC 999 VALUE 111.`
	
	`COMPUTE num_var = num_var + increment_var.`
	
	This type of overflow can be trapped by first checking the expression with a conditional statement, as in the following:
	
	`IF (num_var + increment_var) >= 100`
	`DISPLAY `Limit bypassed``
	`ELSE`
	`COMPUTE num_var = num_var + increment_var`
	`END-IF.`

Example Usage:	`COMPUTE var = var + 5.`
	
	`COMPUTE depreciation =`
	`DDBAMT(cost, life, period, salvage-value).`
	
	`COMPUTE delta = (((x+y)/z)%3)^1.86 - SQRT(x).`

See Also:	ADD SUBTRACT MULTIPLY DIVIDE
Sample Program:	COMPUTE.CBL

CONNECTTOSOCKET

Command:	**CONNECTTOSOCKET**
Syntax:	CONNECTTOSOCKET USING <socket-number> <ip-address> <port-number>.
Description:	The CONNECTTOSOCKET command attempts to establish a remote TCP/IP connection with the machine at *ip-address* using a socket *socket-number* and a port *port-number*. *Ip-address* can be a raw IP address or any valid host name on the network or internet that will accept the communication.

Command:	**CONNECTTOSOCKET**
	This command is conventionally used only on the machine that is considered to be the client in two-way socket connections. It requires that the remote machine accept the connection with ACCEPTFROMSOCKET or an equivalent command. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`CONNECTTOSOCKET USING socket_num_var`
	`host_name_var`
	`port_num_var.`
	
	The CONNECTTOSOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET BINDSOCKET CLOSESOCKET CREATESOCKET LISTENTOSOCKET RECEIVESOCKET SENDSOCKET SHUTDOWNSOCKET
Sample Program:	SERV.CBL

CONTINUE

Command:	**CONTINUE**
Syntax:	CONTINUE.
Description:	The CONTINUE statement can be used as a 'do-nothing' statement in IF .. ELSE clauses or anywhere else in a program. It is treated as a normal line of code, but does not have any consequences and passes control to the next statement. Use it when you wish to structure a condition as IF .. ELSE, but there is no logic to be executed for the IF case, only for the ELSE case. See the Example Usage.

Example Usage:	`IF variable1 = 5`
	`  CONTINUE`
	`ELSE`
	`  DISPLAY `variable1 is not equal to 5``
	`END-IF`

Sample Program:	NEXT.CBL

COPY

Command:	**COPY**
Syntax:	COPY <copybook-literal>.
Description:	COPY loads the file named by the literal value *copybook-literal* into a CobolScript program. The code that is in the copybook file is loaded and executed as if it were part of the loading program, exactly in the position of the COPY statement. In CobolScript, there is no material difference between INCLUDE and COPY.
Example Usage:	`COPY `COPYBOOK.CPY`.`

Command:	**COPY**
	`COPY `copybook.cpy`.`
See Also:	INCLUDE
Sample Program:	COPY.CBL

CREATESOCKET

Command:	**CREATESOCKET**
Syntax:	CREATESOCKET USING <socket-number>.
Description:	The CREATESOCKET command creates a socket descriptor, or virtual circuit, on a TCP/IP socket *socket-number*. Once created, this socket descriptor can then be used with other CobolScript socket commands. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`CREATESOCKET USING socket_num_var.`
	
	The CREATESOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET BINDSOCKET CLOSESOCKET CONNECTTOSOCKET LISTENTOSOCKET RECEIVESOCKET SENDSOCKET SHUTDOWNSOCKET
Sample Program:	SERV.CBL

CREATESHMPOOL

Command:	**CREATESHMPOOL**
Syntax:	CREATESHMPOOL <share memory id>
Description:	The CREATESHMPOOL command create a share memory pool and returns a handle that uniquely identifies that storage.
Example Usage:	`CREATESHMPOOL WS-SHM-ID`
See Also:	PUTSHMPOOL, GETSHMPOOL, DETACHSHMPOOL
Sample Program:	SHM1.CBL, SHM2.CBL

DELETEMAIL

Command:	**DELETEMAIL**
Syntax:	DELETEMAIL USING <email address> <email password> <email number> <pop3 server>
Description:	The DELETEMAIL command will remove an email from an email server. The account and password must be supplied as well as the number of the email to remove.
Example Usage:	`DELETEMAIL USING WS-EMAIL WS-PASS WS-EMAIL-NUM WS-POP3`

Command:	**DELETEMAIL**
See Also:	SENDMAIL, GETMAIL
Sample Program:	SIZE.CBL

DETACHSHMPOOL

Command:	**DETACHSHMPOOL**
Syntax:	DETACHSHMPOOL <shared memory pool id>
Description:	The DETACHSHMPOOL command terminates your usage of a share memory pool. This should be done when you are finished with it.
Example Usage:	`DETACHSHMPOOL WS-SHM-ID`
See Also:	CREATESHMPOOL, PUTSHMPOOL, GETSHMPOOL
Sample Program:	SHM1.CBL, SHM2.CBL

DISPLAY

Command:	**DISPLAY**
Syntax:	DISPLAY <literal1> & <literal2> & … <variable1> <variable2> <expression1> <expression2>
Description:	The DISPLAY statement is used to display literals, variables, and expressions to the standard output device (normally the screen in command-line mode, and the web browser when using CobolScript with a web server). Because CobolScript allows expressions inside DISPLAY statements, individual arguments to DISPLAY must be clearly separated using the ampersand (&). Displaying group items is permitted. Using group items as DISPLAY variables is especially useful when constructing web pages, both for code clarity and reusability purposes (group items can be stored in separate copybooks and used by multiple programs using the COPY and INCLUDE statements). Use of positional string referencing and the use of expressions as arguments in positional string referencing are both permitted in DISPLAY statements. See the Example Usage below. When directly displaying expressions, five significant digits will usually follow the decimal point if the expression's value is non-integer. If the expression's value is extremely large, however (>1,000,000,000), some precision may be lost in the fractional portion of the value. CobolScript has an absolute limit of 16 digits of precision, and will not correctly display or perform computations on any number, expression or variable, with more than 16 total digits. Displaying numeric variables is preferred to displaying expressions when format masks are relevant, or when a value has more than five decimal places; this is because variables will be displayed according to their defined picture clause format. Numeric variables, however, are limited to ten total digits of precision for values less than 100,000,000, slightly more digits of precision for values equal to or higher than 100,000,000, with a absolute maximum of 16 digits of precision. To use a variable in place of an expression, simply define a variable and assign it to the expression of interest using a COMPUTE statement; then DISPLAY the variable in place of the expression. The CobolScript string delimiter is the ` (the accent key, usually located in the upper left corner of American keyboards, below the `Esc` key). String literals must be enclosed by ` in order for them to display properly. Alternatively, the string delimiter can be changed for a particular program run by setting the appropriate command line option. Refer to the section Running CobolScript from the Command Line, in Chapter 2, *Getting Started with CobolScript*, to learn more about command line options.

Command:	**DISPLAY**

Example Usage:	**DISPLAY with multiple arguments:**
	`DISPLAY var1 &`
	`  var2 & var3.`
	
	Expression example:
	`DISPLAY output + 5.`
	
	Positional string referencing example (with expression as argument):
	`DISPLAY `Hour: ` & time(start_pos:start_pos+1).`
	
	Group level data item example:
	`1 group_level.`
	`  5 `This is`.`
	`  5 ` a test.`.`
	
	`DISPLAY group_level.`

See Also:	DISPLAYLF, DISPLAYFILE
Sample Program:	DISPLAY.CBL

DISPLAYASCIIFILE

Command:	**DISPLAYASCIIFILE**
Syntax:	DISPLAYASCIIFILE <filename>
Description:	The DISPLAYASCIIFILE command will display the contents of the specified ASCII file *filename* to the standard output device. DISPLAYASCIIFILE is useful for displaying individual files that contain raw HTML to the calling browser window, so long as the appropriate MIME header information is first displayed; this can be useful if you wish to clearly separate program logic from HTML without going through the effort of placing the HTML into group item variables. See the Creating Virtual HTML section of Chapter 5, *Building Web-Based Systems*, for information on displaying MIME headers. DISPLAYASCIIFILE can also be used within a CobolScript program to transfer an ASCII file to a remote user. This is useful for user-initiated downloads through CGI form submissions on a web site that requires user verification or other logic to execute prior to the actual file transfer. See Chapter 7, *Advanced Internet Programming Techniques Using CobolScript* for more information on how to use DISPLAYASCIIFILE in this manner. DISPLAYASCIIFILE should only be used to display files that are ASCII text; use DISPLAYFILE to display binary files.

Example Usage:	`DISPLAYASCIIFILE `test.dat`.`
	
	`DISPLAYASCIIFILE filename_var.`

See Also:	DISPLAYFILE, DISPLAY, DISPLAYLF
Sample Program:	DOWN.CBL

DISPLAYFILE

Command:	**DISPLAYFILE**
Syntax:	DISPLAYFILE <filename>
Description:	The DISPLAYFILE command will display the contents of the specified binary file *filename* to the standard output device. DISPLAYFILE can be used within a CobolScript program to transfer a binary file (such as an executable) to a remote user. This is useful for user-initiated downloads through

Command:	**DISPLAYFILE**
	CGI form submissions on a web site that requires user verification or other logic to execute prior to the actual file transfer. See Chapter 7, *Advanced Internet Programming Techniques Using CobolScript* for more information on how to use DISPLAYFILE in this manner. DISPLAYFILE should only be used to display binary files; use DISPLAYASCIIFILE to display ASCII text files.

Example Usage:	`DISPLAYFILE `test.exe`.`
	
	`DISPLAYFILE filename_var.`

See Also:	DISPLAYASCIIFILE, DISPLAY, DISPLAYLF
Sample Program:	DOWN.CBL

DISPLAYLF

Command:	**DISPLAYLF**
Syntax:	DISPLAYLF <literal1> & <literal2> & … <variable1> <variable2> <expression1> <expression2>
Description:	DISPLAYLF is the same as DISPLAY, but displays a trailing linefeed character after every elementary item argument has been displayed, including those cases where the initial argument is a group item.

Example Usage:	**Example with gldi argument:**
	`1 group_level.`
	`  5 `This is`.`
	`  5 ` a test.`.`
	`DISPLAYLF group_level.`
	
	Example with multiple elementary arguments:
	`1 var1 PIC X(N) VALUE `This is`.`
	`1 var2 PIC X(N) VALUE ` a test.`.`
	`DISPLAYLF var1 & var2 & `.. .. ..`.`

See Also:	DISPLAY, DISPLAYFILE
Sample Program:	DISPLAY.CBL

DIVIDE

Command:	**DIVIDE**
Syntax:	**Variant 1:** DIVIDE <number or divisor-variable1> … INTO <dividend-variable> [ROUNDED] **Variant 2:** DIVIDE <number or divisor-variable1> … INTO <number or dividend-variable> GIVING <result-variable> [ROUNDED] [REMAINDER <remainder-variable>] **Variant 3:** DIVIDE <number or dividend-variable> BY <number or divisor-variable> GIVING <result-variable> [ROUNDED] [REMAINDER <remainder-variable>] If a REMAINDER clause is specified in Variant 2 of the DIVIDE statement, only a single divisor may be specified. Only one divisor and one dividend may be specified in Variant 3 of the DIVIDE statement, regardless of whether the REMAINDER clause is used.
Description:	**Variant 1** of the DIVIDE statement is used to divide one or more numbers and/or numeric *divisor-variables* into a target numeric *dividend-variable*. The result is stored in

Command:	**DIVIDE**
	the *dividend-variable*, and its previous value is overwritten. This form of DIVIDE is equivalent to the COMPUTE statement: COMPUTE dividend-variable = dividend-variable/divisor-variable1/divisor-variable2/… . **Variant 2** of the DIVIDE statement is used to divide one or more numbers and/or *divisor-variables* into a number or *dividend-variable*, and the result is stored in a separate *result-variable*, thereby preserving the value in the *dividend-variable*. This form of DIVIDE is equivalent to the COMPUTE statement: COMPUTE result-variable = dividend-variable/divisor-variable1/divisor-variable2/… . **Variant 3** of the DIVIDE statement is used to divide a number or *dividend-variable* by a single number and/or *divisor-variable*. The result is stored in a separate *result-variable*. This form of DIVIDE is equivalent to the COMPUTE statement: COMPUTE result-variable = dividend-variable/divisor-variable. **Variants 2 and 3** of DIVIDE permit the usage of the REMAINDER keyword, which stores the remainder from the division operation in a separate *remainder-variable*. The remainder is the portion of the dividend that would be left over if the result were forced to be an integer value. Using the REMAINDER keyword in a DIVIDE statement is equivalent to executing two separate COMPUTE statements, the first the actual division, and the second the remainder calculation using the modulus (%) operator: COMPUTE result-variable = dividend-variable/divisor-variable. COMPUTE remainder-variable = dividend-variable % divisor-variable. **All variants** of DIVIDE permit the use of the ROUNDED keyword, which rounds the target variable, after computation, to the nearest integer.

Example Usage:	**Variant 1:**
	`DIVIDE 1 INTO num_variable.`
	`DIVIDE 1  2  3 INTO num_variable.`
	`DIVIDE value_var INTO total.`
	`DIVIDE 1.11  2  value_var INTO total ROUNDED.`
	
	Variant 2:
	`DIVIDE value_var INTO subtotal GIVING total.`
	
	`DIVIDE 9.99  value_var INTO subtotal`
	`GIVING result ROUNDED.`
	
	`DIVIDE value_var INTO subtotal`
	`GIVING result ROUNDED`
	`REMAINDER remainder.`
	
	Variant 3:
	`DIVIDE subtotal BY value_var GIVING result.`
	
	`DIVIDE subtotal BY value_var GIVING result ROUNDED.`
	
	`DIVIDE subtotal BY value_var GIVING result ROUNDED`
	`REMAINDER remainder.`

See Also:	COMPUTE ADD SUBTRACT MULTIPLY
Sample Program:	DIVIDE.CBL

EXEC SQL

Command:	**EXEC SQL (*CobolScript Professional Edition Only*)**
Syntax:	EXEC SQL <sql-statement> END-EXEC.

Description:	This LinkMaker™ command executes a single SQL statement *sql-statement*. A connection must be established to the data source with the OPENDB command before this command can be used. See Appendix H for further explanation and examples of how to use this command. See Appendix G for more information about configuring data sources.
	
	An SQL communications area is required when working with a LinkMaker™ data source. In CobolScript, this area of memory is allocated by defining the variable *sql-return-codes*. You should include this definition in any of your programs that use LinkMaker™; all of these variables are all standard ODBC return code variables:
	
	`1 sql-return-codes.`
	`  5 sqlstate  PIC X(5).`
	`  5 sqlnativeerror  PIC S9(6).`
	`  5 sqlerrormessage  PIC X(500).`
	`  5 sqlstatement  PIC X(500).`
	
	After an SQL statement has been executed, these variables contain information that was returned from the data source. The variable *sqlstate* will contain the ODBC SQLSTATE returned from the data source; *sqlnativeerror* will contain a data source-specific return code; *sqlerrormessage* will contain text describing an error, if one occurred; and *sqlstatement* will contain a copy of the SQL that was passed to the data source. These return values are provided to assist with database application debugging. It is important to remember, however, that these return values come from the data source, and are therefore specific to that data source. Consult your data source's documentation for specific information about the values returned to these variables.

Example Usage:	`EXEC SQL`
	`  insert into customer`
	`  values ('Jane','Doe', :host_var_balance)`
	`END-EXEC.`

See Also:	OPENDB, CLOSEDB
Sample Program:	SQL.CBL

EXECUTE

Command:	**EXECUTE**
Syntax:	EXECUTE <code-component-1> <code-component-2> …
Description:	EXECUTE dynamically interprets a program statement contained inside *code-component* literal(s) or variable(s), either elementary or group item. Literal keywords such as ACCENT are also permitted as arguments to EXECUTE. EXECUTE is useful when some program logic component is undetermined prior to program execution. See the section titled Dynamic Statement Creation and Execution in Chapter 8 for practical examples of EXECUTE usage. An unusual form of recursion is possible by using EXECUTE to call other EXECUTE statements, e.g.: `EXECUTE `EXECUTE statement_var`.` Although this type of recursion may be difficult to conceptualize and use for normal programming, it is supported. The maximum permitted number of nested recursive calls of this nature is 500; bypassing this limit will cause CobolScript to generate a normal error message specific to this recursion.

Command:	**EXECUTE**
	Moderate caution should be exercised when using EXECUTE to process user input; naturally, it is inadvisable to accept unauthorized user input in the form of a whole code statement for use as an EXECUTE argument; however, since one EXECUTE statement can only process a single code statement, allowing user input for portions of a statement may be appropriate, depending on your objective. The level of flexibility that you permit in user input is directly constrained by how much you wish to restrict user actions; this is therefore your decision to make.

Example Usage:	`1 test_var PIC X(n) VALUE `Hello, `.`
	`1 execute_group.`
	`  5 `DISPLAY`.`
	`  5 ` test_var`.`
	
	`EXECUTE execute_group  `&`  ACCENT  `world.`  ACCENT.`

Sample Program:	EXECUTE.CBL

FD

Command:	**FD**
Syntax:	FD <filename> RECORD IS <bytes-length> BYTES.
Description:	The FD statement describes a data file's location and its record length to CobolScript. This statement is a necessary precursor to all flat (text) file data processing work. The *filename* is a literal or variable that includes the name of the data file as well as any path information, which is necessary if the file is not in the current working directory of the program. The *bytes-length* is a numeric variable or literal that indicates the record length, in bytes, of the file record. The *bytes-length* value should account for any delimiters that are in the record but should *not* account for end-of-line characters; these end-of-line characters vary between Windows and Unix platforms, and this variation is automatically accounted for by CobolScript. The *bytes-length* value must be exact for statements that rely on this value, such as POSITION, to work correctly. Once a data file has been described, it may be opened and further processed. For further information on describing files, see the Data and Copybook Files section of Chapter 3, *CobolScript Language Constructs*. For more information on data file processing, see Chapter 4, *File Processing and I/O*.

Command:	**FD**

Example Usage:	**Example with literal arguments:**
	`FD `test.dat` RECORD IS 50 BYTES.`
	
	Example with variable arguments, which are defined prior to the FD:
	`1 test_file PIC X(n) VALUE `test.dat`.`
	`1 bytes_length PIC 99 VALUE 50.`
	`FD test_file RECORD IS bytes_length BYTES.`
	
	Example that includes path information for a Windows® machine:
	`1 test_file PIC X(n) VALUE`
	`` `c:\windows\desktop\test.dat`.``
	`1 bytes_length PIC 99 VALUE 50.`
	`FD test_file RECORD IS bytes_length BYTES.`
	
	Example that includes path information for a Unix machine:
	`1 test_file PIC X(n) VALUE `/usr/cscript/test.dat`.`
	`1 bytes_length PIC 99 VALUE 50.`
	`FD test_file RECORD IS bytes_length BYTES.`

See Also:	CLOSE OPEN POSITION READ REWRITE WRITE
Sample Program:	FTP.CBL

FTPASCII

Command:	**FTPASCII**
Syntax:	FTPASCII.
Description:	The FTPASCII command sets the FTP file transfer mode to ASCII mode (as opposed to binary mode – see FTPBINARY command below). ASCII file transfer mode should be used when the file to be transferred is an ASCII text file. The FTPASCII command should generally be used immediately before a statement that uses the FTPPUT or FTPGET commands. An open FTP connection must be established with FTPCONNECT prior to issuing the FTPASCII command. The TCP/IP return code and return message variables are populated with standard FTP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Transferring Files Using FTP section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`FTPASCII.`
	
	The FTPASCII command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	FTPBINARY, FTPGET, FTPPUT
Sample Program:	FTP.CBL

FTPBINARY

Command:	**FTPBINARY**
Syntax:	FTPBINARY.
Description:	The FTPBINARY command sets the FTP file transfer mode to binary mode (as opposed to ASCII mode – see FTPASCII command above). Binary file transfer mode should be used when the file to be transferred is a non-text file (any proprietary format file or executable). The FTPBINARY command should generally be used immediately before a statement that uses the FTPPUT or FTPGET commands. An open FTP connection must be established with FTPCONNECT prior to issuing the FTPBINARY command. The TCP/IP return code and return message variables are populated with standard FTP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Transferring Files Using FTP section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`FTPBINARY.`
	
	The FTPBINARY command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	FTPASCII, FTPGET, FTPPUT
Sample Program:	FTP.CBL

FTPCD

Command:	**FTPCD**
Syntax:	FTPCD USING <directory-name>.
Description:	The FTPCD command changes the working FTP directory on a remotely-connected machine to the directory name contained in the variable or literal *directory-name*. An open FTP connection to a remote machine must first be successfully established with FTPCONNECT before FTPCD can be used. The TCP/IP return code and return message variables are populated with standard FTP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Transferring Files Using FTP section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`FTPCD USING `\ftp`.`
	
	`FTPCD USING ftp_dir.`
	
	The FTPCD command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`

Command:	**FTPCD**
	`5 TCPIP-RETURN-MESSAGE PIC X(255).` Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.
See Also:	FTPPUT, FTPGET
Sample Program:	FTP.CBL

FTPCLOSE

Command:	**FTPCLOSE**
Syntax:	FTPCLOSE.
Description:	The FTPCLOSE command closes an FTP connection that has been made with the FTPCONNECT command. The TCP/IP return code and return message variables are populated with standard FTP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Transferring Files Using FTP section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`FTPCLOSE.`
	
	The FTPCLOSE command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	FTPCONNECT
Sample Program:	FTP.CBL

FTPCONNECT

Command:	**FTPCONNECT**
Syntax:	FTPCONNECT USING <hostname> <user-id> <password>.
Description:	The FTPCONNECT command attempts to establish an FTP connection with a remote machine at *hostname* using *user-id* and *password*. The TCP/IP return code and return message variables are populated with standard FTP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Transferring Files Using FTP section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`FTPCONNECT USING `ftp.deskware.com` `anonymous` `info@deskware.com`.`
	
	`FTPCONNECT USING server_var`
	`user_id_var`
	`password_var.`
	
	The FTPCONNECT command requires that the following variable definitions be included in your program:

Command:	**FTPCONNECT**

	1 TCPIP-RETURN-CODES.
	  5 TCPIP-RETURN-CODE PIC 9(07).
	  5 TCPIP-RETURN-MESSAGE PIC X(255).
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	FTPCLOSE
Sample Program:	FTP.CBL

FTPGET

Command:	**FTPGET**
Syntax:	FTPGET USING <filename>.
Description:	The FTPGET command downloads a file *filename* from a connected remote machine via the FTP protocol. An open FTP connection to a remote machine must first be successfully established with FTPCONNECT before FTPGET can be used. The file transfer type is either ASCII or binary, and this can be set prior to calling FTPGET by using the FTPASCII and FTPBINARY commands. The TCP/IP return code and return message variables are populated with standard FTP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Transferring Files Using FTP section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	FTPGET USING `test.dat`.
	
	FTPGET USING test_file.
	
	The FTPBINARY command requires that the following variable definitions be included in your program:
	
	1 TCPIP-RETURN-CODES.
	  5 TCPIP-RETURN-CODE PIC 9(07).
	  5 TCPIP-RETURN-MESSAGE PIC X(255).
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	FTPPUT, FTPASCII, FTPBINARY
Sample Program:	FTP.CBL

FTPPUT

Command:	**FTPPUT**
Syntax:	FTPPUT USING <filename>
Description:	The FTPPUT command uploads a file *filename* to a remote machine via the FTP protocol. An open FTP connection to a remote machine must first be successfully established with FTPCONNECT before FTPPUT can be used. The file transfer type is either ASCII or binary, and this can be set prior to calling FTPPUT by using the FTPASCII and FTPBINARY commands.

Command:	**FTPPUT**
	The TCP/IP return code and return message variables are populated with standard FTP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Transferring Files Using FTP section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`FTPPUT USING `upload.dat`.`
	
	`FTPPUT USING test_file.`
	
	The FTPBINARY command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	FTPGET, FTPASCII, FTPBINARY
Sample Program:	FTP.CBL

GETBANNER

Command:	**GETBANNER**
Syntax:	GETBANNER USING <banner-input> <banner-character-input> <banner-target-variable>.

Description:	GETBANNER places a Unix-style banner into a group item variable. The contents of *banner-input* are the large characters of the banner; the contents of *banner-character-input* are the component characters of the banner, which are the small characters used to make the banner letters. If *banner-character-input* is equal to a single space (`` or the SPACE keyword), the component character of each large letter will be a smaller version of itself, e.g.,
	
	`GETBANNER USING `TEST` SPACE banner_target_variable`
	
	will generate the following output for *banner-target-variable* population:
	
	`TTTTTTT EEEEEEE  SSSSS  TTTTTTT`
	`  T  E  S  S  T`
	`  T  E  S  T`
	`  T  EEEEE  SSSSS  T`
	`  T  E  S  T`
	`  T  E  S  S  T`
	`  T  EEEEEEE  SSSSS  T`
	
	To work properly, GETBANNER requires that the *banner-target-variable* be defined as a group item with 8 elementary items. See example below.

Example Usage:	`1 text_banner_char PIC X VALUE `#`.`
	`1 banner_group.`
	`  5  banner_line1  PIC X(35).`
	`  5  banner_line2  PIC X(35).`
	`  5  banner_line3  PIC X(35).`
	`  5  banner_line4  PIC X(35).`
	`  5  banner_line5  PIC X(35).`
	`  5  banner_line6  PIC X(35).`
	`  5  banner_line7  PIC X(35).`
	`  5  banner_line8  PIC X(35).`
	
	`GETBANNER USING `TEST`  `#`  banner_group.`

Command:	**GETBANNER**

	`DISPLAYLF banner_group.`
	
	`GETBANNER USING text banner_char banner_group.`
	`DISPLAYLF banner_group.`

See Also:	BANNER
Sample Program:	GETBAN.CBL

GETCALENDAR

Command:	**GETCALENDAR**
Syntax:	GETCALENDAR USING <year-input> <month-input> <calendar-target-variable>.
Description:	The GETCALENDAR command places a calendar for a given year *year-input* and a given month *month-input* into a target group item variable *calendar-target-variable*. The *year-input* and *month-input* should be numeric values; if they are variables, their variable declarations must have numeric picture clauses. Any fractional component to *year-input* or *month-input* will be ignored, e.g., a *month-input* of 11.88 will be processed as 11. GETCALENDAR does not support pre-Julian calendar dates, i.e., any date prior to August 1752. To work properly, GETCALENDAR requires that the *calendar-target-variable* be defined as a group item with 8 elementary items. See the Example Usage below.

Example Usage:	`1 year_var  PIC 9(4) VALUE 2001.`
	`1 month_var PIC 99 VALUE 1.`
	
	`1 calendar_group.`
	`  5  calendar_line1   PIC X(30).`
	`  5  calendar_line2   PIC X(30).`
	`  5  calendar_line3   PIC X(30).`
	`  5  calendar_line4   PIC X(30).`
	`  5  calendar_line5   PIC X(30).`
	`  5  calendar_line6   PIC X(30).`
	`  5  calendar_line7   PIC X(30).`
	`  5  calendar_line8   PIC X(30).`
	
	`GETCALENDAR USING 2001 1 calendar_group.`
	`DISPLAYLF calendar_group.`
	
	`GETCALENDAR USING year_var month_var calendar_group.`
	`DISPLAYLF calendar_group.`

See Also:	CALENDAR
Sample Program:	GETCAL.CBL

GETCMDLINE

Command:	**GETCMDLINE**
Syntax:	GETCMDLINE <num-args> <args>
Description:	The GETCMDLINE command will retrieve the input parameters supplied to the program upon execution.
Example Usage:	`GETCMDLINE WS-NUM-ARGS WS-ARGS`
See Also:	CALL
Sample Program:	GETCMDLINE.CBL

GETENV

Command:	**GETENV**
Syntax:	GETENV USING <environmental-variable> <cobolscript-variable>.
Description:	The GETENV command accepts a literal or variable whose contents are an operating system environmental variable, *environmental-variable*, from the operating system environment and copies the value to the variable *cobolscript-variable*. Environmental variables are values that are set by the operating system and provide information about the current operating environment. This command can be used in CobolScript internet programs that need to get information about their web server environment. See Chapter 7 for a list of the environmental variables that are made available by a web server.

Example Usage:	**Example that uses a literal as the environmental variable argument:**
	`GETENV USING `CONTENT_LENGTH` content_length_var.`
	
	Example that uses a variable as the environmental variable argument:
	`1 env_variable PIC X(n) VALUE `CONTENT_LENGTH`.`
	`GETENV USING env_variable content_length_var.`

See Also:	ACCEPT
Sample Program:	GETENV.CBL

GETHOSTBYNAME

Command:	**GETHOSTBYNAME**
Syntax:	GETHOSTBYNAME USING <hostname>.
Description:	The GETHOSTBYNAME command resolves a *hostname* and returns detailed information about the host. The *hostname* that is supplied can either be the name of the host or an IP address. The information returned about a host is stored in the TCPIP-HOSTENT group-level data item variable (see below). It contains all of the aliases for this IP address, other IP addresses associated with this host, the address type, and the address length. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using DNS commands.

Example Usage:	`GETHOSTBYNAME USING `deskware.com`.`
	
	`GETHOSTBYNAME USING `206.228.224.17`.`
	
	`GETHOSTBYNAME USING ip_variable.`
	
	The GETHOSTBYNAME command requires that the following TCP/IP variable definitions be included in your program:
	
	`1 TCPIP-HOSTENT.`
	`  5 TCPIP-HOSTENT-HOSTNAME PIC X(255).`
	`  5 TCPIP-HOSTENT-NUM-ALIASES PIC X(01).`
	`  5 TCPIP-HOSTENT-ALIASES OCCURS 8 TIMES.`
	`  10 TCPIP-HOSTENT-ALIAS PIC X(255).`
	`  5 TCPIP-HOSTENT-ADDRESS-TYPE PIC 9(07).`
	`  5 TCPIP-HOSTENT-ADDRESS-LENGTH PIC 9(07).`
	`  5 TCPIP-HOSTENT-NUM-ADDRESSES PIC X(01).`
	`  5 TCPIP-HOSTENT-ADDRESSES OCCURS 8 TIMES.`
	`  10 TCPIP-HOSTENT-ADDRESS PIC X(255).`

Command:	**GETHOSTBYNAME**

	1 TCPIP-RETURN-CODES.
	  5 TCPIP-RETURN-CODE PIC 9(07).
	  5 TCPIP-RETURN-MESSAGE PIC X(255).
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	GETHOSTNAME
Sample Program:	GETHN.CBL, DNS.CBL

GETHOSTNAME

Command:	**GETHOSTNAME**
Syntax:	GETHOSTNAME USING <hostname-variable>.
Description:	GETHOSTNAME places the hostname of the current machine (the one on which CobolScript is installed) in the target variable *hostname-variable*. The hostname is a machine-specific parameter that generally is derived from the /etc/hosts file on Unix machines, and from the registry on Windows machines. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using DNS commands.

Example Usage:	GETHOSTNAME USING hostname_var.
	
	The GETHOSTNAME command requires that the following TCP/IP variable declarations be included in your program:
	
	1 TCPIP-HOSTENT.
	  5 TCPIP-HOSTENT-HOSTNAME PIC X(255).
	  5 TCPIP-HOSTENT-NUM-ALIASES PIC X(01).
	  5 TCPIP-HOSTENT-ALIASES OCCURS 8 TIMES.
	  10 TCPIP-HOSTENT-ALIAS PIC X(255).
	  5 TCPIP-HOSTENT-ADDRESS-TYPE PIC 9(07).
	  5 TCPIP-HOSTENT-ADDRESS-LENGTH PIC 9(07).
	  5 TCPIP-HOSTENT-NUM-ADDRESSES PIC X(01).
	  5 TCPIP-HOSTENT-ADDRESSES OCCURS 8 TIMES.
	  10 TCPIP-HOSTENT-ADDRESS PIC X(255).
	
	1 TCPIP-RETURN-CODES.
	  5 TCPIP-RETURN-CODE PIC 9(07).
	  5 TCPIP-RETURN-MESSAGE PIC X(255).
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	GETHOSTBYNAME
Sample Program:	GETHN.CBL

GETMAIL

Command:	**GETMAIL**
Syntax:	GETMAIL USING <email-address> <password> <email-number> <email-filename> <smtp-server>.

Command:	**GETMAIL**
Description:	The GETMAIL command connects to *smtp-server* using *email-address* and *password*, and retrieves the email message whose number is *email-number*. The email message is appended to the file *email-filename*. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using Email Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on email commands.

Example Usage:	**Literal argument example:**
	`GETMAIL USING `info@deskware.com` `12jkd` 1 `MAIL.TXT` `deskware.com`.`
	
	Variable argument example:
	`GETMAIL USING email_address`
	`password`
	`number_of_mail_to_get`
	`mail_file`
	`smtp_server.`
	
	The GETMAIL command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`5 TCPIP-RETURN-CODE PIC 9(07).`
	`5 TCPIP-RETURN-MESSAGE PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	SENDMAIL, GETMAILCOUNT
Sample Program:	MAIL.CBL

GETMAILCOUNT

Command:	**GETMAILCOUNT**
Syntax:	GETMAILCOUNT USING <email-address> <password> <count-variable> <smtp-server>.
Description:	The GETMAILCOUNT command connects to *smtp-server* using *email-address* and *password*, determines the number of emails that are in the account for *email-address*, and populates *count-variable* with this number. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using Email Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on email commands.

Example Usage:	**Example with literal arguments for email address, password, and smtp server:**
	`GETMAILCOUNT USING `info@deskware.com` `12F3g` email_count `deskware.com`.`
	
	Example with variable arguments:
	`GETMAILCOUNT USING email_address`
	`password`
	`email_count`
	`smtp_server.`
	
	The GETMAILCOUNT command requires that the following variable definitions be included in your program:

Command:	**GETMAILCOUNT**

	1 TCPIP-RETURN-CODES.
	  5 TCPIP-RETURN-CODE PIC 9(07).
	  5 TCPIP-RETURN-MESSAGE PIC X(255).
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	GETMAIL
Sample Program:	MAIL.CBL

GETMAILSIZE

Command:	**GETMAILSIZE**
Syntax:	GETMAILSIZE USING <email address> <email password> <email number> <pop3 server> <size of emails>
Description:	The GETMAILSIZE command is used to determine the size in bytes of an email message.
Example Usage:	`GETMAILSIZE USING WS-EMAIL WS-PASS WS-NUM WS-PO3 WS-EMAIL-SIZE.`
See Also:	GM.CBL, SM.CBL
Sample Program:	SIZE.CBL

GETSHMPOOL

Command:	**GETSHMPOOL**
Syntax:	GETSHMPOOL <share memory pool id> <data>
Description:	The GETSHMPOOL command will read the contents of a share memory pool and populate a variable with the contents.
Example Usage:	`GETSHMPOOL WS-SHM-ID WS-DATA`
See Also:	PUTSHMPOOL, CREATESHMPOOL, DETACHSHMPOOL
Sample Program:	SHM1.CBL, SHM2.CBL

GETTIMEFROMSERVER

Command:	**GETTIMEFROMSERVER**
Syntax:	GETTIMEFROMSERVER USING <hostname> <server-time-variable>.
Description:	GETTIMEFROMSERVER contacts a server *hostname*, and retrieves and stores the local time from that machine in a variable *server-time-variable*. The variable can be either the name of the host or the IP address. Note that currently the GETTIMEFROMSERVER command will only work successfully if a time daemon is running on the *hostname* server; if a time daemon is not running on *hostname*, the GETTIMESERVER command will wait indefinitely for a response from the server. If this happens, the process must be killed manually to properly terminate execution of the CobolScript program. Generally, you should only

Command:	**GETTIMEFROMSERVER**
	use GETTIMEFROMSERVER when you are certain that a time daemon is running on *hostname*. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes.

Example Usage:	`GETTIMEFROMSERVER USING `purdue.edu` server_time.`
	
	`GETTIMEFROMSERVER USING server_var server_time.`
	
	The GETTIMEFROMSERVER command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	GETHOSTNAME, GETHOSTBYNAME
Sample Program:	IPTIME.CBL

GETWEBPAGE

Command:	**GETWEBPAGE**
Syntax:	GETWEBPAGE <hostname> <webpage-path> <webpage-filename>.
Description:	The GETWEBPAGE command connects to *hostname* using the HTTP protocol, and retrieves the webpage at location *webpage-path*. This webpage is then written to the file *webpage-filename*, replacing any previous contents of *webpage-filename*. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes.

Example Usage:	`GETWEBPAGE `www.deskware.com`  `/index.htm`  `DESK.TXT`.`
	
	`GETWEBPAGE server_var path_var filename_var.`
	
	The GETWEBPAGE command requires that the following variable definitions be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program. This copybook includes these variable definitions.

See Also:	GETHOSTNAME, GETHOSTBYNAME
Sample Program:	WEB.CBL

GOBACK

Command:	**GOBACK**
Syntax:	GOBACK.
Description:	The GOBACK command ends the execution of a program. No commands following GOBACK will be executed. There is no material difference between GOBACK and STOP RUN in CobolScript.

Command:	**GOBACK**
	For COBOL programmers, note that GOBACK is *not* the equivalent of the COBOL GOBACK command.
Example Usage:	`GOBACK.`
See Also:	STOP RUN
Sample Program:	GOBACK.CBL

IF

Command:	**IF**
Syntax:	IF <condition> [THEN] <statement> : [ELSIF <elsif-condition> :] [ELSIF <elsif-condition-2> :] . . [ELSE <statement> :] END-IF
Description:	The IF statement is a basic programming construct; it controls program flow based on whether a condition evaluates to TRUE or FALSE. IF first evaluates *condition*, and if *condition* is TRUE, executes the statement(s) following *condition* (or after the optional THEN keyword) and then leaves the IF clause by passing control to the statement following the END-IF keyword. If *condition* is FALSE, control passes to the next ELSIF clause or ELSE keyword, if one or these exists. If an ELSIF clause exists, *elsif-condition* is evaluated. If *elsif-condition* is TRUE, the statements following the ELSIF clause are executed, and control is passed to the statement following the ELSIF keyword. If *elsif-condition* is FALSE, control passes to the next ELSIF or ELSE, if one exists. If an ELSE is reached and all prior conditions and ELSIF conditions have evaluated to FALSE, the statement(s) after the ELSE keyword are executed. For this reason, if you specify an ELSE clause it should always be the last part of your IF statement. There is no imposed limit to the number of ELSIF clauses that may be specified. Practical limits do exist due to program size limits, but you should not encounter these limits in normal programming. ELSIF clauses should always be placed in the order that you want each ELSIF condition evaluated, if the order is relevant. Generally, the use of ELSIF clauses will necessitate the use of an ELSE to cover all other cases; good programming practice warrants the use of an ELSE when using ELSIFs even if no action should be taken in the ELSE case. This can be done by using the CONTINUE statement, which acts as a placeholder or 'do nothing' statement, as in the following: `IF var > 1` `  DISPLAY `Greater than one`` `ELSIF var =1` `  DISPLAY `Equal to one`` `ELSIF var < 0` `  DISPLAY `Less than zero`` `ELSE` `  CONTINUE`

Command:	**IF**

	END-IF.
	
	Condition and *elsif-condition* are any normal expressions that evaluate to a number; typically, conditions are statements of fact, and therefore can only evaluate to 1 (TRUE) or 0 (FALSE), as in the following cases:
	
	IF var >= 1
	IF letter IS ALPHABETIC THEN
	IF ALPHABETIC(letter)
	IF a = 1 OR a = 2 THEN
	IF (x + y + z) IS NOT GREATER THAN 6 AND y = 4
	
	In the above cases, all TRUE-evaluating conditions have an integer value of 1. However, in CobolScript, any nonzero condition result is considered TRUE, and only zero results are considered FALSE. Therefore, the following type of conditions are also possible in CobolScript:
	
	IF (-5) THEN
	IF var
	IF NOT(var)
	IF x + y + z
	
	For COBOL programmers, note that CobolScript enforces C-like rules for expression construction. COBOL constructs such as implied subjects and implied operators encourage poor programming practices and are not permitted in CobolScript – all conditions must be completely and explicitly defined.
	
	For more information on conditions and expressions, refer to the Expressions and Conditions section in Chapter 3, *CobolScript Language Constructs*.

Example Usage:	IF var1 > var2
	  DISPLAY `var1 is greater than var2`
	ELSIF var < var2
	  DISPLAY `var1 is less than var2`
	ELSE
	  DISPLAY `var1 is equal to var2`
	END-IF

Sample Program:	IF.CBL

INCLUDE

Command:	**INCLUDE**
Syntax:	INCLUDE <copybook-literal>.
Description:	INCLUDE loads the file named by the literal value *copybook-literal* into a CobolScript program. The code that is in the copybook file is loaded and executed as if it were part of the loading program, exactly in the position of the COPY statement. In CobolScript, there is no material difference between INCLUDE and COPY.
Example Usage:	INCLUDE `COPYBOOK.INC`. INCLUDE copybook_var.
See Also:	COPY
Sample Program:	COPY.CBL

INITIALIZE

Command:	**INITIALIZE**
Syntax:	INITIALIZE <init-variable>.
Description:	The INITIALIZE command moves SPACES or ZEROS to variable *init-variable*; SPACES are moved to the variable if it is defined as alphanumeric (PIC X) and ZEROS if it has been defined as numeric (PIC 9). Note that CobolScript automatically initializes all variables that have VALUE clauses; for this reason, using a VALUE clause is normally preferred to using the INITIALIZE statement.
Example Usage:	`INITIALIZE var1.`
Sample Program:	INIT.CBL

LISTENTOSOCKET

Command:	**LISTENTOSOCKET**
Syntax:	LISTENTOSOCKET USING <socket-number> <backlog-queue-length>.
Description:	The LISTENTOSOCKET command prepares a socket *socket-number* to accept an incoming connection. The *backlog-queue-length* is the number of incoming connection requests permitted to queue while accepted connections are processed. LISTENTOSOCKET should be called prior to using ACCEPTFROMSOCKET. This command is conventionally used only on the machine that is considered to be the server in two-way socket connections. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`LISTENTOSOCKET USING socket_num_var backlog_num_var.`
	
	The LISTENTOSOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET BINDSOCKET CLOSESOCKET CONNECTTOSOCKET CREATESOCKET RECEIVESOCKET SENDSOCKET SHUTDOWNSOCKET
Sample Program:	SERV.CBL

LOWER

Command:	**LOWER**
Syntax:	LOWER <variable>
Description:	The LOWER command converts the contents of <variable> to lower case.
Example Usage:	`LOWER WS-NAME`

Command:	**LOWER**
See Also:	UPPER
Sample Program:	LOWER.CBL

LTRIM

Command:	**LTRIM**
Syntax:	LTRIM <variable>
Description:	The LTRIM command removes leading space characters from <variable>.
Example Usage:	`LTRIM WS-NAME`
See Also:	TRIM, RTRIM
Sample Program:	TRIM.CBL

MOVE

Command:	**MOVE**
Syntax:	MOVE <source-data> TO <target-variable>.

Description:	The MOVE statement copies the contents of a literal or variable, *source-data*, to the contents of the *target-variable*.
	
	In the cases of an alphanumeric to alphanumeric, an alphanumeric to numeric, or a numeric to alphanumeric MOVE, if the length of the *source-data* contents is greater than the length of *target-variable*, *target-variable* is populated with the *source-data* characters from left to right, and the remaining source characters are discarded.
	
	In the case of a numeric to numeric MOVE, if the length of the contents of *source-data* is greater than the length of *target-variable*, *target-variable* is populated as follows:
	• Digits to the right of the decimal point are populated in *target-variable* from left to right, and remaining digits in the *source-data* decimal are discarded, for example:
	If var1 is defined as PIC 9.99,
	`MOVE 5.432 TO var1`
	will place 5.43 in var1.
	• Digits to the left of the decimal point are populated in *target-variable* from right to left, and remaining higher digits in the *source-data* are discarded, for example:
	If var1 is defined as PIC 9.99,
	`MOVE 65.432 TO var1`
	will place 5.43 in var1.
	
	Besides simple moves, MOVE also allows a group item to be moved to another group item, or a group item to be moved to an elementary item. MOVE also permits both source and target variables to use positional string referencing; refer to the section titled Manipulating CobolScript Variables in Chapter 8 for further details.

Example Usage:	**Simple MOVE:**
	`MOVE var1 TO var2.`
	
	MOVE with positional referencing of source variable:
	`MOVE var1(1:2) TO var3.`
	
	MOVE with positional referencing of target variable:
	`MOVE `test` TO var5(start_position:length).`

See Also:	SET
Sample Program:	MOVE.CBL

MULTIPLY

Command:	**MULTIPLY**
Syntax:	**Variant 1:** MULTIPLY <number or variable> … BY <target-variable> [ROUNDED] **Variant 2:** MULTIPLY <number or variable> … BY <number or variable> GIVING <target-variable> [ROUNDED]
Description:	**Variant 1** of MULTIPLY is used to multiply one or more numeric literals and/or numeric variables together, storing the result in the numeric *target-variable*. All literals and variables are multiplied together to produce the result, including the value in *target-variable* prior to the multiplication. **Variant 2** of MULTIPLY is used to multiply one or more numeric literals and/or variables together, with the result stored in *target-variable*, whose original contents are not considered in the multiplication. Thus, if VAR has an initial value of 3, performing the operation MULTIPLY 2 BY 2 GIVING VAR will place a value of 4, not 12, into VAR. Both forms of MULTIPLY permit the use of the ROUNDED keyword, which rounds the target variable (after computation) to the nearest integer.

Example Usage:	**Variant 1:**
	`MULTIPLY 2 BY num.`
	`MULTIPLY 2  3 BY num.`
	`MULTIPLY value BY total.`
	`MULTIPLY 1.11  2  value BY total ROUNDED.`
	
	Variant 2:
	`MULTIPLY value BY subtotal GIVING total.`
	`MULTIPLY 2 BY 3 GIVING total ROUNDED.`

See Also:	COMPUTE ADD SUBTRACT DIVIDE
Sample Program:	MULTIPLY.CBL

OPEN

Command:	**OPEN**
Syntax:	OPEN <filename> FOR READING [DELIMITED WITH <delimiter-character>]. OPEN <filename> FOR WRITING [DELIMITED WITH <delimiter-character>]. OPEN <filename> FOR APPENDING [DELIMITED WITH <delimiter-character>]. OPEN <filename> FOR UPDATING [DELIMITED WITH <delimiter-character>].
Description:	OPEN is used to open a text data file named by the literal or variable *filename* for READING, UPDATING, WRITING (which positions the disk head at the beginning of the file), or APPENDING (which positions the disk head at the end of the file). The FOR UPDATING clause allows the update records in an existing data file. Use it in conjunction with the REWRITE statement. The DELIMITED WITH option treats the *delimiter-character* (which must be a single character literal value or variable, or a character keyword such as TAB or SPACE) as the separator between fields, rather than relying on field lengths to define where record fields begin and end inside the file (as is the case when DELIMITED WITH is omitted). The delimiter can be any character that is in the ASCII character set, but remember that

Command:	**OPEN**
	no delimiter characters may appear inside any of the record fields; otherwise, an unintended field separation will occur. For more information on file manipulation, refer to Chapter 4, *File Processing and I/O*.

Example Usage:	`OPEN test_file_var FOR READING.`
	
	`OPEN `TEST.DAT` FOR READING DELIMITED WITH `,`.`
	
	`OPEN `TEST.DAT` FOR UPDATING DELIMITED WITH TAB.`
	
	`OPEN test_file_var FOR WRITING.`
	
	`OPEN test_file_var FOR APPENDING DELIMITED WITH `|`.`
	
	`OPEN test_file_var FOR UPDATING DELIMITED WITH delim_var.`

See Also:	CLOSE FD POSITION READ REWRITE WRITE
Sample Program:	IO.CBL

OPENDB

Command:	**OPENDB (*CobolScript Professional Edition Only*)**
Syntax:	OPENDB USING <data-source-name> <user-id> <password> <return-code-variable>.
Description:	The OPENDB command opens a LinkMaker™ connection to a data source *data-source-name* using *user-id* and *password*. Upon completion, OPENDB populates *return-code-variable* with an integer value of 1 (success) or 0 (failure). For OPENDB to work correctly, an ODBC driver for the specific data source must be installed, and a DSN (Data Source Name) must be defined. On Unix platform machines, UnixODBC must also be installed prior to using any LinkMaker commands. See Appendix G for more information about configuring LinkMaker™ data sources and installing and configuring UnixODBC on Unix platform machines.
Example Usage:	`OPENDB USING data_source user_id password ret_code.`
See Also:	CLOSEDB EXEC SQL
Sample Program:	SQL.CBL

PERFORM

Command:	**PERFORM**

Syntax:	**Variant 1, Standard PERFORM:**
	PERFORM <module-name>.
	
	Variant 2, PERFORM .. UNTIL:
	PERFORM <module-name> UNTIL <condition>.
	
	Variant 3, Inline PERFORM:
	PERFORM UNTIL <condition>
	  :
	  :
	END-PERFORM

Description:	The basic PERFORM statement has three variants in CobolScript:

Command:	**PERFORM**
	Variant 1, Standard PERFORM: The standard PERFORM passes program control to a program module *module-name* a single time, and then returns control to the statement following the PERFORM. When the PERFORM is encountered during program execution, control passes immediately to the first line of code within *module-name*. The code within *module-name* then executes; after the last statement in *module-name* has been processed, control is returned to the line immediately following the PERFORM statement, and program execution continues normally. **Variant 2, PERFORM .. UNTIL:** PERFORM .. UNTIL is used to pass program control to a program module *module-name* multiple times, until *condition* is satisfied. Execution of the code within *module-name* is similar to the standard PERFORM. When a PERFORM .. UNTIL statement is encountered during program execution, *condition* is immediately evaluated; if it evaluates to FALSE, *module-name* is executed, and control returns to the beginning of the PERFORM .. UNTIL statement, so that *condition* can be evaluated again. If *condition* evaluates to TRUE, *module-name* is not executed, and control passes to the statement following the PERFORM .. UNTIL. There are two important points to keep in mind when using PERFORM .. UNTIL: • First, remember that *condition* must evaluate to TRUE in order for control to be passed to the statement following the PERFORM .. UNTIL; if condition always evaluates to FALSE, the program will be caught in an endless loop, repeatedly performing the code in *module-name*. To avoid this, some of the code within *module-name* must change some component of *condition*, so that *condition* will eventually be TRUE. • Second, remember that *condition* is always evaluated prior to the execution of *module-name*. Therefore, if *condition* evaluates to TRUE the first time that the PERFORM .. UNTIL is encountered, the code in *module-name* will never be performed. More information on conditions, condition evaluation, and permitted condition syntax is available in the Command Reference entry for IF, and in the Expressions and Conditions section in Chapter 3, *CobolScript Language Constructs*. **Variant 3, Inline PERFORM:** The Inline PERFORM is simply a variation of PERFORM .. UNTIL. Instead of performing a separate module, however, it executes the code that is between the PERFORM and END-PERFORM statements multiple times, until *condition* is satisfied.

Example Usage:	**Variant 1:**
	`PERFORM INIT.`
	
	Variant 2:
	`PERFORM PROCESSING UNTIL counter = 5.`
	
	Variant 3:
	`PERFORM UNTIL counter = 5`
	`  ADD 1 TO counter`
	`  DISPLAY `counter = ` & counter`
	`END-PERFORM`

See Also:	IF (for explanation of condition evaluation, PERFORM .. VARYING
Sample Program:	PERFORM.CBL

PERFORM .. VARYING

Command:	**PERFORM .. VARYING**
Syntax:	**Variant 1, Standard PERFORM .. VARYING:**

Command:	**PERFORM .. VARYING**
	PERFORM <module-name> VARYING <varying-variable> FROM <from-amount> BY <increment-amount> UNTIL <condition>. **Variant 2, Inline PERFORM VARYING:** PERFORM VARYING <varying-variable> FROM <from-amount> BY <increment-amount> UNTIL <condition> : : END-PERFORM
Description:	PERFORM .. VARYING has two variants in CobolScript: **Variant 1, Standard PERFORM .. VARYING:** The standard PERFORM .. VARYING is used to pass program control to a program module *module-name* multiple times, until *condition* is satisfied, while also incrementing *varying-variable* with each call to *module-name*. Condition evaluation, and the execution of *module-name*, are handled in the same way as PERFORM .. UNTIL; see Variant 2 in the PERFORM command description above for details. In a PERFORM .. VARYING, the *varying-variable* is initialized on the first loop pass, or incremented for every pass other than the first, then *condition* is evaluated, then *module-name* is performed, in that order. This happens as follows: • On the first pass through the PERFORM .. VARYING statement, the *varying-variable* is first initialized to *from-amount*; then, if *condition* evaluates to FALSE, the code in *module-name* is executed, and control returns to the beginning of the PERFORM .. VARYING. If *condition* evaluates to TRUE on the first pass, *module-name* is not performed. • From the second pass through the PERFORM .. VARYING and all subsequent passes, *varying-variable* is first incremented by *increment-amount*; if *condition* evaluates to FALSE, *module-name* is performed, and control returns to the beginning of the PERFORM .. VARYING. If *condition* evaluates to TRUE, control passes to the statement following the PERFORM .. VARYING. *Increment-amount* can be any nonzero number or numeric variable; to decrement the *varying-variable* rather than increment it, use a negative value for *increment-amount*. More information on conditions, condition evaluation, and permitted condition syntax is available in the Command Reference entry for IF, and in the Expressions and Conditions section in Chapter 3, *CobolScript Language Constructs*. **Variant 2, Inline PERFORM VARYING:** The Inline PERFORM VARYING is a variation of PERFORM .. VARYING. Instead of performing a separate module, however, it executes the code that is between the PERFORM and END-PERFORM statements multiple times, until *condition* is satisfied.

Example Usage:	**Variant 1:**
	`PERFORM PROCESSING`
	`  VARYING varying_nbr`
	`  FROM 5 BY -1`
	`  UNTIL varying_nbr = 0.`
	
	Variant 2:
	`PERFORM VARYING varying_nbr`
	`  FROM 10 BY 2 UNTIL SQRT(varying_nbr)>=4`
	`  DISPLAY `varying_nbr = ` & varying_nbr`
	`END-PERFORM`

Command:	**PERFORM .. VARYING**
See Also:	IF (for explanation of condition evaluation), PERFORM.
Sample Program:	PERFORM.CBL

POSITION

Command:	**POSITION**
Syntax:	**Variant 1, Absolute POSITION:** POSITION <filename> AT RECORD <record-number>. **Variant 2, Relative POSITION:** POSITION <filename> RELATIVE OFFSET <number-of-records>.

Description:	The POSITION statement positions the file pointer in *filename* at the beginning of a particular record within a text data file in a single step.
	
	POSITION can be used to simulate an indexing system within flat files; if a data file uses a sequential numeric value as the record key value, a record within the file can be randomly (directly) accessed given that key value. This functionality is similar to COBOL relative file processing.
	
	When using the POSITION statement, the number of bytes specified in the BYTES clause of the FD statement for your file must exactly match the number of bytes in the data file record; this value is used to reposition the file pointer, and a BYTES value that is larger or smaller than the actual data record size will cause the file pointer to be incorrectly positioned.
	
	Variant 1, Absolute POSITION:
	The absolute POSITION moves the file pointer directly to the beginning of the record at *record-number*, which must be a numeric literal or variable. The first record in the file is considered to be record number 1; therefore, *record-number* must be a positive integer, and its value must fall within the range:
	
	(1 <= *record-numbe*r <= total number of records in file)
	
	The *record-number* value (and hence the number of records in your data file) cannot exceed 2,147,483,647 (2.1 billion).
	
	Variant 2, Relative POSITION:
	The relative POSITION moves the file pointer relative to its current position. *Number-of-records* must be an integer-valued numeric literal or variable. This value indicates the number of records, counting from the current record, that the file pointer should be moved. Thus, a value of 1 will shift the file pointer one record forward in the data file; a value of –1 will shift the file pointer one record back. The value of *number-of-records* must fall within the absolute range:
	
	(-2,147,483,647 <= *number-of-records* <= 2,147,483,647)
	
	A *number-of-records* value that causes the file pointer to be positioned before the beginning of the data file or after the end of the data file will cause a CobolScript error.
	
	When using relative POSITION, keep in mind that certain file operations such as READ, WRITE, and REWRITE will advance the file pointer by one record. Thus, in the following code, the second READ statement will read the eighth record in the file, not the seventh, because the first READ and the second POSITION advance the file pointer by one record each:
	
	`POSITION file_name AT RECORD 6.`
	`READ file_name INTO record_var.`

Command:	**POSITION**

	`POSITION file_name RELATIVE OFFSET 1.`
	`READ file_name INTO record_var.`
	
	For more information on using POSITION, see the Relative and Absolute File Positioning section of Chapter 4, *File Processing and I/O*.

Example Usage:	**Variant 1:**
	`POSITION acct_file AT RECORD 5000.`
	
	`POSITION acct_file AT RECORD record_num_var.`
	
	Variant 2:
	`POSITION cust_file RELATIVE OFFSET rel_ofs_var.`
	
	`POSITION `cust.dat` RELATIVE OFFSET 100.`

See Also:	CLOSE FD OPEN READ REWRITE WRITE
Sample Program:	POSITION.CBL

PUTSHMPOOL

Command:	**PUTSHMPOOL**
Syntax:	PUTSHMPOOL <shared memory pool id> <data>.
Description:	The PUTSHMPOOL command copies the contents of a variable into a share memory pool.
Example Usage:	`PUTSHMPOOL WS-SHM-ID WS-DATA`
See Also:	GETSHMPOOL, CREATESHMPOOL, DETACHSHMPOOL
Sample Program:	SHM1.CBL, SHM2.CBL

READ

Command:	**READ**
Syntax:	READ <filename> INTO <record-variable> [AT END <imperative-statement>].
Description:	READ is used to read a single record from a text data file *filename* into a variable *record-variable*. Generally, *record-variable* should be defined as a group item, with an elementary item declared for each individual field within the record. The AT END clause specifies an *imperative-statement* to execute when the end of the file is reached. AT END is an error-trapping clause and should be used whenever multiple records are read using a single READ statement, or whenever it is unclear whether the end of file could be encountered with a particular READ. *Imperative-statement* should be a single-statement command only (rather than an IF clause or an inline PERFORM), such as MOVE, DISPLAY, ACCEPT, COMPUTE, or a simple PERFORM (PERFORM <module-name>). The maximum allowed size of a data file record in CobolScript is 10,000 bytes. Data beyond the 10,000th byte in an individual record in a data file will be ignored.

Command:	**READ**
	For more information on file manipulation, refer to Chapter 4, *File Processing and I/O*.

Example Usage:	`READ `TEST.DAT` INTO input_record`
	`   AT END MOVE 1 TO eof.`
	
	`READ test_file INTO input_record`
	`   AT END PERFORM END-READ-MODULE.`
	
	`READ test_file INTO input_record`
	`   AT END DISPLAY `Made it to end of file`.`

See Also:	CLOSE FD OPEN POSITION REWRITE WRITE
Sample Program:	READ.CBL

READBLOCK

Command:	**READBLOCK**
Syntax:	READBLOCK <byte_size to read> BYTES FROM <filename> INTO <input record>
Description:	The READBLOCK commands reads a specified number of bytes from an input file.
Example Usage:	`READBLOCK byte_size BYTES FROM filename_var INTO input_record`
See Also:	WRITEBLOCK
Sample Program:	RB.CBL

RTRIM

Command:	**RTRIM**
Syntax:	RTRIM <variable>
Description:	The RTRIM command removes trailing space characters from <variable>.
Example Usage:	`RTRIM WS-NAME`
See Also:	LTRIM, TRIM
Sample Program:	TRIM.CBL

RECEIVESOCKET

Command:	**RECEIVESOCKET**
Syntax:	RECEIVESOCKET USING <socket-number> <receiving-variable>.
Description:	The RECEIVESOCKET command receives the data in *receiving-variable* from a remotely transmitting machine, over an open socket connection using socket *socket-number*. For RECEIVESOCKET to work properly, the transmitting (remote) machine must transmit the data using SENDSOCKET or an equivalent command. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes.

Command:	**RECEIVESOCKET**
	See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`RECEIVESOCKET USING socket_num_var receive_string.`
	
	The RECEIVESOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET, BINDSOCKET, CLOSESOCKET, CONNECTTOSOCKET, CREATESOCKET, LISTENTOSOCKET, SENDSOCKET, SHUTDOWNSOCKET
Sample Program:	SERV.CBL

REWRITE

Command:	**REWRITE**
Syntax:	REWRITE <record-value> TO <filename>.
Description:	The REWRITE command writes a variable or literal *record-value* to a data file *filename*. REWRITE can be used to update a record in an existing data file without having to read through the file twice. It is essentially the equivalent of backing the file pointer up one record, and then performing a WRITE. See the Example Usage below. For more information on using REWRITE, see the section titled Writing to a File by Updating Existing Records in Chapter 4, *File Processing and I/O*.

Example Usage:	**Rewrite inside an inline PERFORM:**
	`PERFORM UNTIL (at_end_test OR record_updated)`
	`  READ filename_var INTO record_variable`
	`  AT END MOVE 1 TO at_end_test`
	`  IF record_variable(1:4) = key_val`
	`  REWRITE record_variable TO filename_var`
	`  MOVE 1 TO record_updated`
	`  END-IF`
	`END-PERFORM.`
	
	In the example above, REWRITE is used to update an individual file record whose first four characters match the contents of the variable key_val.

See Also:	CLOSE, FD, OPEN, POSITION, READ, WRITE
Sample Program:	REWRITE.CBL

SENDMAIL

Command:	**SENDMAIL**
Syntax:	SENDMAIL USING <to-address> <from-address> <subject> <message> <smtp-server>.
Description:	The SENDMAIL command connects to *smtp-server*, and uses this connection to send an email message to *to-address* using *from-address* as the sender, with *subject* as the subject of the email and *message* as the message body. It should be noted that CobolScript SENDMAIL is *not* the same as the Unix shell's sendmail command; this is an important distinction because potential security holes and

Command:	**SENDMAIL**
	'backdoors' that are present in Unix sendmail do not exist in CobolScript SENDMAIL, making CobolScript SENDMAIL a safer command to use. SENDMAIL permits a group item to be used as the message argument or as the destination address argument. This eliminates the 2000-byte restriction that exists when using elementary items as arguments, and also allows a single message to be sent to multiple recipients. Also, the SENDMAIL destination address allows aliasing – this can be done by enclosing the destination address in < and > and preceding this enclosed address with an alias. See Example 2 below for a demonstration of SENDMAIL used with a gldi message and *to-address* argument, and various forms of destination addresses. The TCP/IP return code and return message data structures are populated with standard TCP/IP The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using Email Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on email commands.
Example Usage:	**Example 1 (basic SENDMAIL):** `SENDMAIL USING `test@deskware.com` `info@deskware.com` `SENDMAIL TEST` email_message.` **Example 2 (SENDMAIL with some gldi arguments, aliasing in destination addresses):** `1 message_variable.` `5 intro_text PIC X(50) value `Dear Mr. Thomas,`.` `5 FILLER PIC X(1000).` `5 FILLER PIC X(1000).` `COPY `TCPIP.CPY`.` `1 from_address PIC X(n) VALUE `info@deskware.com`.` `1 to_address.` `5 FILLER PIC X(n) VALUE `<nobody1@ttttt.com>`.` `5 FILLER PIC X(n) VALUE `Nobody <nobody2@ttttt.com>`.` `5 FILLER PIC X(n) VALUE `nobody3@ttttt.com`.` `1 subject PIC X(n) VALUE `Test`.` `1 smtp_server PIC X(n) VALUE `ttttt.com`.` `SENDMAIL USING to_address from_address` `subject message_variable smtp_server.` The SENDMAIL command requires that the following variable definitions be included in your program: `1 TCPIP-RETURN-CODES.` `5 TCPIP-RETURN-CODE PIC 9(07).` `5 TCPIP-RETURN-MESSAGE PIC X(255).` Alternatively, include the sample file TCPIP.CPY in your program, as in Example 2 above. This copybook includes these variable definitions.

Command:	**SENDMAIL**
See Also:	GETMAILCOUNT, GETMAIL
Sample Program:	MAIL.CBL

SENDSOCKET

Command:	**SENDSOCKET**
Syntax:	SENDSOCKET USING <socket-number> <send-string>.
Description:	The SENDSOCKET command transmits the data in *send-string* over an open TCP/IP socket connection using socket *socket-number*. For SENDSOCKET to work properly, the receiving (remote) machine must receive the data using RECEIVESOCKET or an equivalent command. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`SENDSOCKET USING connected_socket_num send_string.`
	
	The SENDSOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET BINDSOCKET CLOSESOCKET CONNECTTOSOCKET CREATESOCKET LISTENTOSOCKET RECEIVESOCKET SHUTDOWNSOCKET
Sample Program:	SERV.CBL

SET

Command:	**SET**
Syntax:	SET <target-variable> TO <source-data>.
Description:	The SET command sets the contents of *target-variable* equal to the contents of *source-data*. In CobolScript, MOVE is preferred to SET. For COBOL programmers, note that CobolScript SET is *not* equivalent to the COBOL SET command.
Example Usage:	`SET var1 TO var2.`
See Also:	MOVE
Sample Program:	SET.CBL

SHUTDOWNSOCKET

Command:	**SHUTDOWNSOCKET**
Syntax:	SHUTDOWNSOCKET USING <socket-number> <shutdown-method>.
Description:	The SHUTDOWNSOCKET command prepares an open socket connection *socket-number* to be closed. SHUTDOWNSOCKET should be used prior to calling CLOSESOCKET, to ensure a graceful termination. The *shutdown-method* is a numeric variable or literal flag that describes how the socket will be shut down. The allowed values for *shutdown-method* are: 0: Receives are no longer allowed 1: Sends are no longer allowed 2: Sends and receives are no longer allowed Normally, a *shutdown-method* of 1 is preferred; by using 1, the local machine will alert the remote machine that the local machine is no longer transmitting data packets, which initiates a graceful termination of the socket connection. The TCP/IP return code and return message variables are populated with standard TCP/IP return codes and messages after execution of this command. They can be examined after command execution for error-trapping purposes. See the Using TCP/IP Commands section of Chapter 6, *Network and Internet Programming Using CobolScript*, for more information on using socket commands.

Example Usage:	`SHUTDOWNSOCKET USING socket_num 1.`
	
	`SHUTDOWNSOCKET USING socket_num shutdown_method.`
	
	The SHUTDOWNSOCKET command requires that the following TCP/IP variable declarations be included in your program:
	
	`1 TCPIP-RETURN-CODES.`
	`  5 TCPIP-RETURN-CODE  PIC 9(07).`
	`  5 TCPIP-RETURN-MESSAGE  PIC X(255).`
	
	Alternatively, include the sample file TCPIP.CPY in your program with a COPY or INCLUDE statement. This copybook includes these variable definitions.

See Also:	ACCEPTFROMSOCKET BINDSOCKET CLOSESOCKET CONNECTTOSOCKET CREATESOCKET LISTENTOSOCKET RECEIVESOCKET SENDSOCKET
Sample Program:	SERV.CBL

SUBTRACT

Command:	**SUBTRACT**
Syntax:	**Variant 1:** SUBTRACT <number or variable> … FROM <target-variable> [ROUNDED] **Variant 2:** SUBTRACT <number or variable> … FROM <number or variable> GIVING <target-variable> [ROUNDED
Description:	**Variant 1** of SUBTRACT is used to subtract one or more numeric literals and/or numeric variables from a numeric *target-variable*, with the result stored in the *target-variable*. All literals and variables to the left of the FROM keyword are subtracted from the *target-variable* in order to determine its new value. **Variant 2** of SUBTRACT is used to subtract one or more numeric literals and/or numeric variables from a numeric literal or variable, with the result stored in the *target-variable*. All literals and variables to the left of the FROM keyword are subtracted from the literal or variable that is between the FROM and GIVING keywords in order to arrive

Command:	**SUBTRACT**
	at the new value for *target-variable*. Thus, if num_var has an initial value of 1, performing the operation: `SUBTRACT 1 FROM 3 GIVING num_var` will place a value of 2 into num_var. Both forms of SUBTRACT permit the use of the ROUNDED keyword, which rounds the target variable (after computation) to the nearest integer.

Example Usage:	**Variant 1:**
	`SUBTRACT 1 FROM num.`
	`SUBTRACT 1  2  3 FROM num.`
	`SUBTRACT value FROM total.`
	`SUBTRACT 1.11  2  value FROM total ROUNDED.`
	
	Variant 2:
	`SUBTRACT value FROM subtotal GIVING total.`
	`SUBTRACT 9.99  value FROM subtotal GIVING total ROUNDED.`

See Also:	COMPUTE ADD MULTIPLY DIVIDE
Sample Program:	SUBTRACT.CBL

STOP RUN

Command:	**STOP RUN**
Syntax:	STOP RUN
Description:	The STOP RUN command ends the execution of a program. No commands following STOP RUN will be executed. There is no material difference between GOBACK and STOP RUN in CobolScript.
Example Usage:	`STOP RUN.`
See Also:	GOBACK
Sample Program:	STOPRUN.CBL

TOK

Command:	**TOK**
Syntax:	TOK <source variable> <target variable> <delimiter>
Description:	The TOK command tokenizes <source variable> and places the contents in <target variable> base on the supplied <delimiter>
Example Usage:	`TOK WS-NAME WS-TOKENS ","`
See Also:	TRIM, LTRIM, RTRIM
Sample Program:	TOK.CBL

TRIM

Command:	**TRIM**
Syntax:	TRIM <variable>
Description:	The TRIM command removes leading and trailing space characters from the contents of <variable>.
Example Usage:	`TRIM WS-NAME`
See Also:	LTRIM, RTRIM
Sample Program:	TRIM.CBL

UPPER

Command:	**UPPER**
Syntax:	UPPER <variable>
Description:	The UPPER command converts the contents of <variable> to Upper Case.
Example Usage:	`UPPER WS-NAME`
See Also:	LOWER
Sample Program:	UPPER.CBL

WRITE

Command:	**WRITE**
Syntax:	WRITE <record-value> TO <filename>.
Description:	The WRITE command writes a variable or literal *record-value* to a text data file *filename*. Each individual call to WRITE causes a new record, terminated with the appropriate linefeed character sequence (depending on your platform), to be written to the data file specified. WRITE should be used to write out whole records at a time; populate elementary item variables that comprise a group item variable, then use the group item as the argument to WRITE. See the Example Usage. For details on how to do record updates using READ and WRITE refer to Chapter 4, *File Processing and I/O*.

Command:	**WRITE**

Example Usage:	**WRITE of a literal argument:**
	`WRITE `1234` TO `TEST.DAT`.`
	
	WRITE of a group item variable argument:
	`1 record.`
	`  5  first_variable   PIC X(10) VALUE `TEST DATA `.`
	`  5  second_variable  PIC 99.99 VALUE 15.31.`
	`  5  last_variable    PIC XXXXX VALUE `ABCD`.`
	
	`WRITE record TO `TEST.DAT`.`

See Also:	CLOSE FD OPEN POSITION READ REWRITE
Sample Program:	WRITE.CBL

WRITEBLOCK

Command:	**WRITEBLOCK**
Syntax:	WRITEBLOCK <output record> TO <output filename>
Description:	WRITEBLOCK sends the contents of a variable to an output file.
Example Usage:	`WRITEBLOCK output_record TO output_filename_var`
See Also:	READBLOCK
Sample Program:	RB.CBL

Appendix

B

Function Reference

CobolScript comes with a number of different mathematical and test functions to simplify computational work. All functions currently implemented in CobolScript return numeric values and can each be used in the COMPUTE, PERFORM .. UNTIL, and IF statements except in certain cases, where usage is restricted to a subset of these commands as noted.

Function arguments can be in the form of literals, variables, or expressions (See Chapter 3 for a detailed discussion of expression syntax). Examples of each form appear in the function reference below. Using inappropriate or out-of-range function arguments will cause program termination and a corresponding CobolScript error will be displayed. See Appendix I for a detailed description of error messages.

Function usage in CobolScript programs is flexible. A function can act as a standalone expression, as in the following two examples:

```
COMPUTE x = ABS(x).

IF ALPHABETIC(y) THEN
  DISPLAY `y IS ALPHABETIC`
END-IF.
```

A function can also be one component of a longer expression, as in these two examples:

```
COMPUTE x = x + ABS (x).

IF (x+ABS(x)) > 5 THEN
  DISPLAY `> CASE`
END-IF.
```

The example usage in this appendix contains only the function, with arguments, as it would be used inside a COMPUTE or conditional statement. To use the function inside a statement, the remainder of the statement syntax must be in place. For instance, if the example usage for function is:

```
ABS(-5)
```

then, to use this example inside a COMPUTE statement, a variable assignment must occur, as in the following:

```
COMPUTE abs_val = ABS(-5).
```

To use the ABS example inside a conditional statement, some form of conditional test must occur. At its simplest, the return value of the function can be tested to see whether it is false (equal to zero) or true (nonzero). The condition below will evaluate to a nonzero value (true), and the word TRUE will:

```
IF ABS(-5) THEN
   DISPLAY `TRUE`
ELSE
   DISPLAY `FALSE`
END-IF.
```

More complex conditions are also possible (for more on this, see the section in Chapter 3 dealing with conditions and expressions). The condition below evaluates to false, and the word FALSE will display:

```
IF ABS(-5) > 5 OR ABS(-5) < 0 THEN
   DISPLAY `TRUE`
ELSE
   DISPLAY `FALSE`
END-IF.
```

Following is the CobolScript function list, in alphabetical order. Each function entry contains a description, function syntax, example function usage, and the numeric return value of the example.

ABS

Function:	**ABS**
Function Name:	Absolute value
Syntax:	ABS(n)
Description:	Returns the absolute value of the argument. If n < 0, returns (-n), if n >= 0, returns n.
Example Usage:	ABS(-5)
Return Value:	5

ACOS

Function:	**ACOS**
Function Name:	Arccosine
Syntax:	ACOS(n)
Description:	Returns a value, in radians, that approximates $\cos^{-1}(n)$, where $-1 <= n <= 1$. The result will be in the range $0 <= \cos^{-1}(n) <= \pi$.
Example Usage:	ACOS(0)
Return Value:	1.570796327 (approximates $\pi/2$)

ACOSH

Function:	**ACOSH**
Function Name:	Inverse hyperbolic cosine
Syntax:	ACOSH(n)
Description:	Returns $\cosh^{-1}(n)$, where n >= 1, and cosh(n) is equivalent to the expression $\frac{(e^n + e^{-n})}{2}$

Function:	**ACOSH**
Example Usage:	ACOSH(5.25)
Return Value:	2.3421 (for PIC format 9.9999)

ALPHABETIC

Function:	**ALPHABETIC**
Function Name:	Alphabetic test function
Syntax:	x IS ALPHABETIC *or* ALPHABETIC(x)
Description:	This function tests a value to determine if it is alphabetic or not. It can only be used in the conditions of IF and PERFORM .. UNTIL statements.
Example Usage (1):	BUFFER IS ALPHABETIC
Return Value (1):	For case where BUFFER = `ABCdef`, return value is 1 (TRUE) For case where BUFFER = `ABC123`, return value is 0 (FALSE)
Example Usage (2):	ALPHABETIC(`ABC123`)
Return Value (2):	0 (FALSE)

ANNUITY

Function:	**ANNUITY**
Function Name:	Annuity given present value
Syntax:	ANNUITY(present value, interest rate per period, number of periods)
Description:	Calculate an annuity amount given present value, interest rate per period, and number of periods. It is assumed that the first annuity payment is made at the end of the first period, not at the beginning.
Example Usage:	ANNUITY(1000, .05, 5)
Return Value:	380.40

ANNUITYFV

Function:	**ANNUITYFV**
Function Name:	Annuity given future value
Syntax:	ANNUITYFV(future value, interest rate, number of payments)
Description:	Calculates an annuity amount given a future value, interest rate per period, and number of periods. It is assumed that the first annuity payment is made at the end of the first period, not at the beginning.
Example Usage:	ANNUITYFV(20000,0.00833,48)
Return Value:	340.61

ASIN

Function:	**ASIN**
Function Name:	Arcsine
Syntax:	ASIN(n)
Description:	Returns a value, in radians, that approximates $\sin^{-1}(n)$, where $-1 <= n <= 1$. The result will be in the range $-\pi/2 <= \sin^{-1}(n) <= \pi/2$.
Example Usage:	ASIN(1)
Return Value:	1.570796327 (approximates $\pi/2$)

ASINH

Function:	**ASINH**

Function Name:	Inverse hyperbolic sine
Syntax:	ASINH(n)
Description:	Returns $\sinh^{-1}(n)$, where n >= 1, and sinh(n) is equivalent to the expression $\frac{(e^n - e^{-n})}{2}$
Example Usage:	ASINH(5.25)
Return Value:	2.3603 (for PIC format 9.9999)

ATAN

Function:	**ATAN**
Function Name:	Arctangent
Syntax:	ATAN(n)
Description:	Returns a value, in radians, that approximates $\tan^{-1}(n)$, where $-\infty <= n <= \infty$. The result will be in the range $-\pi <= \tan^{-1}(n) <= \pi$.
Example Usage:	ATAN(1)
Return Value:	0.7853981634 (approximates $\pi/4$)

ATAN2

Function:	**ATAN2**
Function Name:	Arctangent of y/x
Syntax:	ATAN2(x, y)
Description:	Returns a value, in radians, that approximates $\tan^{-1}(y/x)$, where $-\infty <= n <= \infty$. The result will be in the range $-\pi <= \tan^{-1}(n) <= \pi$. ATAN2 allows a zero-valued argument for x.
Example Usage:	ATAN2(0, 1)
Return Value:	0

ATANH

Function:	**ATANH**
Function Name:	Inverse hyperbolic tangent
Syntax:	ATANH(n)
Description:	Returns $\tanh^{-1}(n)$, where $-1 < n < 1$, and tanh(n) is equivalent to the expression $\frac{(e^n - e^{-n})}{(e^n + e^{-n})}$
Example Usage:	ATANH(.999999999)
Return Value:	10.73422678

CALTOJ

Function:	**CALTOJ**
Function Name:	Calories to joules
Syntax:	CALTOJ(n)
Description:	Returns joules given calories, or kilojoules given Calories (kcal).
Example Usage:	CALTOJ(1)
Return Value:	4.185500000

CCTOCIN

Function:	**CCTOCIN**
Function Name:	Cubic centimeters to cubic inches
Syntax:	CCTOCIN(n)

Description:	Returns cubic inches, given cubic centimeters.
Example Usage:	CCTOCIN(5200)
Return Value:	317.3234693

CEILING

Function:	**CEILING**
Function Name:	Ceiling
Syntax:	CEILING(n)
Description:	Returns the ceiling of a given value. The ceiling is the next integer value larger than the fractional argument. If an integer value is specified as the argument to CEILING, the return value will equal the argument.
Example Usage:	CEILING(1.2)
Return Value:	2

CHOOSE

Function:	**CHOOSE**
Function Name:	Choose
Syntax:	CHOOSE(n, r)
Description:	Returns the result of performing the stochastic operation *n choose r*. *n choose r* is the number of ways that r objects can be selected from n unique objects, where the order of selection of the r objects is not relevant. *n choose r* is equivalent to $\frac{n!}{r!(n-r)!}$ Both arguments to CHOOSE must be positive integer values.
Example Usage:	CHOOSE(9, 4)
Return Value:	126

CINTOCC

Function:	**CINTOCC**
Function Name:	Cubic inches to cubic centimeters
Syntax:	CINTOCC(n)
Description:	Returns cubic centimeters, given cubic inches.
Example Usage:	CINTOCC(318)
Return Value:	5211.086352

CMTOIN

Function:	**CMTOIN**
Function Name:	Centimeters to inches
Syntax:	CMTOIN(n)
Description:	Returns inches, given centimeters.
Example Usage:	CMTOIN(10)
Return Value:	3.937007874

COS

Function:	**COS**
Function Name:	Cosine
Syntax:	COS(θ)

Description:	Returns the cosine of an angle θ, when θ is specified in radians. The cosine of an angle is equivalent to the ratio base/hypotenuse, when a right triangle is formed using θ (or the complement of θ, as appropriate), and the intersection point of the angle θ is taken to be the x, y coordinate (0, 0). Negative return values for COS apply for $\pi/2 < \theta < 3\pi/2$. Cosine is a periodic function with a period of 2π.
Example Usage:	COS(PI(0))
Return Value:	-1

COSH

Function:	**COSH**
Function Name:	Hyperbolic cosine
Syntax:	COSH(n)
Description:	Returns cosh(n), where cosh(n) is equivalent to the expression $\frac{(e^n + e^{-n})}{2}$
Example Usage:	COSH(LN(VAR))
Return Value:	Result is equivalent to ½ * (VAR + 1 /VAR) For example, if VAR = 2, result is 1.25

CTOFAHR

Function:	**CTOFAHR**
Function Name:	Celsius to Fahrenheit
Syntax:	CTOFAHR(n)
Description:	Returns degrees Fahrenheit, given degrees Celsius.
Example Usage:	CTOFAHR(0)
Return Value:	32

DDBAMT

Function:	**DDBAMT**
Function Name:	Double Declining Balance amount
Syntax:	DDBAMT(cost, life, period, salvage value)
Description:	Returns a depreciation amount using the Double Declining Balance (DDB) method, given an initial cost, asset life (in periods), the number of the current period for which the amount should be calculated, and a salvage value (frequently zero).
Example Usage:	DDBAMT(10000,5,2,0)
Return Value:	2400

EXP

Function:	**EXP**
Function Name:	e to the power of n
Syntax:	EXP(n)
Description:	Returns e, the natural number, raised to the power of n
Example Usage:	EXP(1)
Return Value:	2.718281828

FACT

Function:	**FACT**
Function Name:	Factorial (n!)

Syntax:	FACT(n)
Description:	Returns n!, the factorial of n. *n!* is equivalent to the multiplicative series n * (n – 1) * … * 2 * 1 Factorials are commonly used in calculating probabilities and permutations.
Example Usage:	FACT(5)
Return Value:	120

FAHRTOC

Function:	**FAHRTOC**
Function Name:	Fahrenheit to Celsius
Syntax:	FAHRTOC(n)
Description:	Returns degrees Celsius, given degrees Fahrenheit.
Example Usage:	FAHRTOC(212)
Return Value:	100

FLOOR

Function:	**FLOOR**
Function Name:	Floor
Syntax:	FLOOR(n)
Description:	Returns the floor of a value. The floor is the next integer value smaller than the fractional argument. If an integer value is specified as the argument to FLOOR, the return value will equal the argument.
Example Usage:	FLOOR(5.9)
Return Value:	5

FTOM

Function:	**FTOM**
Function Name:	Feet to meters
Syntax:	FTOM(n)
Description:	Returns meters, given a measurement in English feet.
Example Usage:	FTOM(110*3)
Return Value:	100.5840000

FV

Function:	**FV**
Function Name:	Future value
Syntax:	FV(present value, interest rate per period, number of periods)
Description:	Returns a future value of an amount, given a present value, an interest rate per period, and the number of periods.
Example Usage:	FV(1000, .083333, 48) -- Corresponds to 10% annual interest rate, compounded monthly, for 48 months.
Return Value:	1489.35

FVANNUITY

Function:	**FVANNUITY**
Function Name:	Future value of an annuity
Syntax:	FVANNUITY(annuity payment, interest rate, number of periods)

Description:	Returns the future value of an annuity given annuity payment, interest rate per annuity period, and the number of periods.
Example Usage:	FVANNUITY(200, .0083333, 36)
Return Value:	8356.359

GALTOL

Function:	**GALTOL**
Function Name:	Gallons to liters
Syntax:	GALTOL(n)
Description:	Returns liters, given gallons.
Example Usage:	GALTOL(10)
Return Value:	37.85411784

GMTOOZ

Function:	**GMTOOZ**
Function Name:	Grams to ounces
Syntax:	GMTOOZ
Description:	Returns ounces, given grams.
Example Usage:	GMTOOZ(28)
Return Value:	0.9876709346

HPTOKW

Function:	**HPTOKW**
Function Name:	Horsepower to kilowatts
Syntax:	HPTOKW(n)
Description:	Returns kilowatts, given horsepower.
Example Usage:	HPTOKW(100)
Return Value:	74.5699871600

INTOCM

Function:	**INTOCM**
Function Name:	Inches to centimeters
Syntax:	INTOCM(n)
Description:	Returns centimeters, given inches.
Example Usage:	INTOCM(12)
Return Value:	30.48000000

JTOCAL

Function:	**JTOCAL**
Function Name:	Joules to calories
Syntax:	JTOCAL(n)
Description:	Returns calories given joules, or Calories (kcal) given kilojoules.
Example Usage:	JTOCAL(1)
Return Value:	0.2389200812

KGTOPD

Function:	**KGTOPD**
Function Name:	Kilograms to pounds
Syntax:	KGTOPD(n)
Description:	Returns pounds, given kilograms
Example Usage:	KGTOPD(100)
Return Value:	220.4622622

KMTOML

Function:	**KMTOML**
Function Name:	Kilometers to miles
Syntax:	KMTOML(n)
Description:	Returns miles, given kilometers.
Example Usage:	KMTOML(10)
Return Value:	6.213711922

KWTOHP

Function:	**KWTOHP**
Function Name:	Kilowatts to horsepower
Syntax:	KWTOHP(n)
Description:	Returns horsepower, given kilowatts
Example Usage:	KWTOHP(100)
Return Value:	134.1022090

LN

Function:	**LN**
Function Name:	Natural log
Syntax:	LN(n)
Description:	Returns the natural log of a value.
Example Usage:	LN(2)
Return Value:	0.6931471806

LOG

Function:	**LOG**
Function Name:	Logarithm
Syntax:	LOG(n)
Description:	Returns the base-10 logarithm of the given value.
Example Usage:	LOG(2)
Return Value:	0.3010299957

LTOGAL

Function:	**LTOGAL**
Function Name:	Liters to gallons
Syntax:	LTOGAL(n)
Description:	Returns gallons, given liters
Example Usage:	LTOGAL(100)
Return Value:	26.41720524

MLTOKM

Function:	**MLTOKM**
Function Name:	Miles to kilometers
Syntax:	MLTOKM(n)
Description:	Returns kilometers, given miles.
Example Usage:	MLTOKM(100)
Return Value:	160.9344000

MTOF

Function:	**MTOF**
Function Name:	Meters to feet
Syntax:	MTOF(n)
Description:	Returns English feet, given meters
Example Usage:	MTOF(10)
Return Value:	32.80839895

NUMERIC

Function:	**NUMERIC**
Function Name:	Numeric test function
Syntax:	x IS NUMERIC *or* NUMERIC(x)
Description:	This function tests a value to determine if it is numeric or not. It can only be used in the conditions of IF and PERFORM .. UNTIL statements.
Example Usage (1):	BUFFER IS NUMERIC
Return Value (1):	For case where BUFFER = `123456`, return value is 1 (TRUE) For case where BUFFER = `ABC123`, return value is 0 (FALSE)
Example Usage (2):	NUMERIC(`ABC`)
Return Value (2):	0 (FALSE)

NUMPMTS

Function:	**NUMPMTS**
Function Name:	Number of payments
Syntax:	NUMPMTS(present value, payment amount, interest rate per payment period)
Description:	Returns the number of payments given a present value, payment amount, and interest rate per payment period. Returns –1 if solution is infeasible, i.e., no number of payments will equal present value, given interest rate.
Example Usage:	NUMPMTS(10000,500, .0083333)
Return Value:	21.96961262

OZTOGM

Function:	**OZTOGM**
Function Name:	Ounces to grams
Syntax:	OZTOGM(n)
Description:	Returns grams, given ounces.
Example Usage:	OZTOGM(1)
Return Value:	28.34952313

PDTOKG

Function:	**PDTOKG**
Function Name:	Pounds to kilograms
Syntax:	PDTOKG(n)
Description:	Returns kilograms, given pounds.
Example Usage:	PDTOKG(10)
Return Value:	4.535923700

PERMUTAT

Function:	**PERMUTAT**
Function Name:	Permutate
Syntax:	PERMUTAT(n,r)
Description:	Returns the result of performing the stochastic operation *n permutate r*. *n permutate r* is the number of ways that r objects can be selected from n unique objects, where the order of selection of the r objects is considered relevant. *n permutate r* is equivalent to $\frac{n!}{(n-r)!}$ Both arguments to PERMUTAT must be positive integer values.
Example Usage:	PERMUTAT(5, 3)
Return Value:	60

PI

Function:	**PI**
Function Name:	Pi (π)
Syntax:	PI(0)
Description:	Returns the ratio π that defines the relationship between a circle's circumference and its diameter, i.e., π = circumference / diameter. An argument of 0 (zero) should always be specified for PI.
Example Usage:	PI(0)
Return Value:	3.141592654

PV

Function:	**PV**
Function Name:	Net present value
Syntax:	PV(future value, interest rate, number of periods)
Description:	Returns the net present value of a specified future value, given interest rate per period and number of periods.
Example Usage:	PV(10000, .0083333, 36)
Return Value:	7417.405862

PVANNUITY

Function:	**PVANNUITY**
Function Name:	Present value of an annuity
Syntax:	PVANNUITY(annuity amount, interest rate per period, number of periods)
Description:	Returns the present value of an annuity given the annuity amount, interest rate per period, and number of periods.
Example Usage:	PVANNUITY(100,.0083333,24)
Return Value:	2167.086350

RANDOM

Function:	**RANDOM**
Function Name:	Random number generator
Syntax:	RANDOM(0)
Description:	Returns a pseudo-random number in the range 0 < n < 1. No seeding is necessary for the CobolScript random number generator because the generator is internally seeded with each call to the RANDOM function; however, because of this, it is not possible with CobolScript Standard Edition to repeat a random number series with consecutive runs of the same CobolScript program. Also, because the auto-seeding process is dependent on the processor clock, and seeding is done with the first call to the RANDOM function within any one program, closely timed, periodic runs of a program that makes one call to the random number generator will not necessarily generate a good random profile, or a non-correlative scattering of the plotted random numbers. To achieve good non-correlative scattering, the RANDOM function must either be called multiple times within the same program, or the times at which the program is called must itself be somewhat random, such as with a CGI or other on-demand type of application. An argument of 0 (zero) should always be specified for RANDOM.
Example Usage:	RANDOM(0)
Return Value:	Trial 1: 0.3203833125 Trial 2: 0.0529190954 Trial 3: 0.6378368480 Trial 4: 0.4803613392

ROOT

Function:	**ROOT**
Function Name:	Root function
Syntax:	ROOT(x, n)
Description:	Returns the nth root of x.
Example Usage:	ROOT(27, 3)
Return Value:	3

ROUNDED

Function:	**ROUNDED**
Function Name:	Round function
Syntax:	ROUNDED(n) *or* COMPUTE x ROUNDED = n.
Description:	Rounds the argument to the nearest integer, with a decimal value of .5 rounding to the next highest integer.

Example Usage (1):	COMPUTE x ROUNDED = 1.5.
Return Value (1):	2
Example Usage (2):	ROUNDED(1.49999)
Return Value (2):	1

SIGN

Function:	**SIGN**
Function Name:	Sign function
Syntax:	SIGN(n)
Description:	Returns the sign of the argument – returns –1 if n<0, 0 if x=0, +1 if x>0.
Example Usage:	SIGN(-9)
Return Value:	-1

SIN

Function:	**SIN**
Function Name:	Sine
Syntax:	SIN(θ)
Description:	Returns the sine of an angle θ, when θ is specified in radians. The sine of an angle is equivalent to the ratio height/hypotenuse, when a right triangle is formed using θ (or the complement of θ, as appropriate), and the intersection point of the angle θ is taken to be the x, y coordinate (0, 0). Negative return values for SIN apply for $0 < \theta < \pi$. Sine is a periodic function with a period of 2π.
Example Usage:	SIN(PI(0)/2)
Return Value:	-1

SINH

Function:	**SINH**
Function Name:	Hyperbolic sine
Syntax:	SINH(n)
Description:	Returns sinh(n), where sinh(n) is equivalent to the expression $\frac{(e^n - e^{-n})}{2}$
Example Usage:	SINH(LN(VAR))
Return Value:	Result is equivalent to ½ * (VAR - 1 / VAR) For example, if VAR = 2, result is 0.75

SQRT

Function:	**SQRT**
Function Name:	Square root
Syntax:	SQRT(n)
Description:	Returns the square root of the argument.
Example Usage:	SQRT(625)

Return Value:	25

STRLINEAMT

Function:	**STRLINEAMT**
Function Name:	Straight-line depreciation amount
Syntax:	STRLINEAMT(cost, life, salvage value)
Description:	Returns a depreciation amount using the straight-line depreciation method, given an initial cost, asset life (in periods), and a salvage value (frequently zero). Because the straight-line depreciation amount is equal for each period in which depreciation is calculated, it is not necessary to specify which period is the current one.
Example Usage:	STRLINEAMT(50000,10,0)
Return Value:	5000

SYDAMT

Function:	**SYDAMT**
Function Name:	Sum-of-the-years'-digits depreciation amount
Syntax:	SYDAMT(cost, life, period, salvage value)
Description:	Returns a depreciation amount using the sum-of-the-years'-digits (SYD) method, given an initial cost, asset life (in periods), the number of the current period for which the amount should be calculated, and a salvage value (frequently zero).
Example Usage:	SYDAMT(1000,5, 2, 0)
Return Value:	266.6666667

TAN

Function:	**TAN**
Function Name:	Tangent
Syntax:	TAN(θ)
Description:	Returns the tangent of an angle θ, when θ is specified in radians. The tangent of an angle is equivalent to the ratio height/base (or, alternatively, sin θ / cos θ), when a right triangle is formed using θ (or the complement of θ, as appropriate), and the intersection point of the angle θ is taken to be the x, y coordinate (0, 0). Negative return values for TAN apply for $0 < \theta < \pi/2$. Tangent is a periodic, non-continuous function with a period of π; the non-continuity exists because there are asymptotes in the graph of tan θ as θ approaches π/2. From both the negative and positive side of $\theta = \pi/2$, tan θ approaches infinity (∞). However, the CobolScript TAN function will return a large, arbitrary value for TAN(PI(0)/2) because of the finite size of the constant PI(0). Always keep this in mind when working with the CobolScript TAN function.
Example Usage:	TAN(PI(0)/2)
Return Value:	16331778728383844.0

TANH

Function:	**TANH**

Function Name:	Hyperbolic tangent
Syntax:	TANH(n)
Description:	Returns tanh(n), where tanh(n) is equivalent to the expression $\frac{(e^{n} - e^{-n})}{(e^{n} + e^{-n})}$
Example Usage:	TANH(LN(2))
Return Value:	0.6

Appendix

C

CobolScript® Constraints

In its native state, CobolScript is an interpreted computer language. Because of this, many of the additional steps required when using a compiler to run a program (linking, compiling, etc.) are not necessary with CobolScript. This reduction is steps can be a real timesaver; the time required to change and test code incrementally using CobolScript is significantly less than the time required for the same tasks using a compiler.

Due to the fact that CobolScript is an interpreter, however, some constraints do exist in CobolScript. Constraints such as these are a necessary component of all interpreters, although some interpreted languages may not trap errors that occur when constraints are bypassed, and GPFs or memory errors can result. In CobolScript, bypassing any of the absolute constraints listed below will cause CobolScript run-time errors, not GPFs.

The CobolScript engine constraints are listed on the following page.

Constraint	CobolScript Standard Edition	CobolScript Professional Edition
Maximum permitted lines of code in any one CobolScript program	32,767 lines *-or-* As many lines as computer running CobolScript can load into memory, *whichever is less*	Same as Standard Edition
Maximum number of variables that can be defined in a single program	1,000	10,000
Maximum number of files that can be used by a single program	20	50
Number of OCCURS clause levels (array dimensions) permitted	1	No imposed limit
Maximum statement length, in bytes	500	Same as Standard Edition
Maximum tokens (keywords, literals, expression components, etc.) per line	80	Same as Standard Edition
Maximum token length, in characters	80	Same as Standard Edition
Maximum number of modules permitted in a single program	500	Same as Standard Edition
Maximum program call stack size	300	Same as Standard Edition
Maximum elementary data item variable size, in bytes	2,000	Same as Standard Edition
Maximum record length, in bytes	10,000	Same as Standard Edition
Maximum number of TCP/IP Aliases for GETHOSTBYNAME	8	Same as Standard Edition
Maximum number of TCP/IP Addresses for GETHOSTBYNAME	8	Same as Standard Edition
Maximum TCP/IP hostname size, in bytes	255	Same as Standard Edition
Maximum number of TCP/IP sockets that may be used by a single program	20	Same as Standard Edition
Maximum number of EXECUTE statement recursive calls	500	Same as Standard Edition

Appendix

D

Sample CobolScript® Programs

Each of the sample programs listed below demonstrates a particular command, feature, group of features, or syntax of the CobolScript language. These samples can be used as instructional tools, or as templates for development.

Working with data files requires the proper read and write permissions on directories and files. On Unix systems especially, attempting to run a file manipulation program with insufficient file or directory privileges is a common oversight. Failing to set these permissions correctly will prevent CobolScript file input and output. Make certain that permissions are set correctly before manipulating files with a CobolScript program.

All sample programs are available for download from the Deskware Registered User web site. Refer to your license agreement for information on restrictions governing the redistribution of these programs, or of programs based on these programs.

Command Line Sample Programs

The following sample programs can be run from the command line. To run one of them, at the command prompt type:

```
cobolscript.exe <program-name>
```

where <program-name> is the name of the sample program to be run.

Program Name	Demonstrates…
ACCEPT.CBL	How to get the system date and time, and how to capture standard input.
ARITHMETIC.CBL	Basic arithmetic commands (ADD, SUBTRACT, MULTIPLY, DIVIDE.
BANNER. CBL	How to display a Unix-style banner.
CAL.CBL	How to display a calendar for a given month and year.
CALL.CBL	How to call an external application.
CLIENT.CBL	TCP/IP client example – use with SERV.CBL.
COMPUTE.CBL	Different forms and uses of the COMPUTE statement, and the use of expressions.
CONVFUNCS.CBL	Metric system to English system and English to metric conversion functions.
COPY.CBL	How to include copybooks with the COPY command.
DISPLAY.CBL	How to display different forms of output to the standard output device.
DYNFILE.CBL	Dynamic file creation example.
EXECUTE.CBL	How to use the EXECUTE command to dynamically execute statements.
F_EXEC.CBL	A file processing example that uses the EXECUTE command.
FINANCEFUNCS.CBL	Financial calculation and depreciation functions.
FTP.CBL	File Transfer Protocol commands.
GEOMFUNCS.CBL	Trigonometric functions (sine, cosine, inverses, hyperbolics, etc).
GETBAN.CBL	How to save a Unix-style banner to a variable.
GETCAL.CBL	How to save a calendar for a given month and year to a variable.
GETENV.CBL	How to retrieve environmental variables from the operating system.
GETHN.CBL	Use of GETHOSTNAME command.
GETTIME.CBL	Use of GETTIMEFROMSERVER command.
GOBACK. CBL	How to terminate a program using GOBACK.
HMATHFUNCS.CBL	Higher math functions (logs, natural logs, rounding, roots, etc).
IF.CBL	IF conditions.
INIT.CBL	How to initialize variables.
INPUT.CSV	Input data file for RECCOPY.CBL sample program.
MAIL.CBL	How to send simple emails, retrieve emails, and get count of emails on an SMTP server.
MOVE.CBL	Use of the MOVE statement.
OCCURS.CBL	How to use the OCCURS clause.
OPENCLSE.CBL	How to open and close files.
PERFORM.CBL	Use of the PERFORM statement.
POSITION.CBL	How to use the POSITION statement to position to a particular record in a text data file.
PROBFUNCS.CBL	Probability functions (random number generator, factorials, etc).
PROFOCCR.CBL	Professional Edition OCCURS clause example.
RECCOPY.CBL	A command-line program that demonstrates how to convert delimited files that were created in Excel or other applications to CobolScript delimited files, if record updates are necessary to the data.
READ.CBL	How to read data from files.
REPLICA.CBL	How to use the REPLICA clause.
REWRITE.CBL	How to use the REWRITE statement to update records in a text data file.
SERV.CBL	TCP/IP server example – use with CLIENT.CBL.
SET.CBL	Use of SET statement.
SQL.CBL	Professional Edition SQL example.
SQL.CPY	SQL return variable copybook.
STOPRUN.CBL	How to terminate a program using STOP RUN.
TCPIP.CPY	TCP/IP return variable copybook.
WEB.CBL	How to retrieve web pages and save them to a file.
WRITE. CBL	How to write data to a file.

Web-Based Sample Programs

The following sample programs are meant to be executed from a web browser. A web server must be running on the machine on which CobolScript is installed, and CobolScript and the CobolScript programs must be placed in the web server's cgi-bin directory. On Unix machines, make certain that file and directory permissions allow reading and writing to the cgi-bin directory.

If the web server is running properly and all files are in the correct location, any of the programs below can be run by typing:

http://<ip address>/cgi-bin/cobolscript.exe?<filename>

in the web browser's URL, where <ip address> is the web server's IP address or domain name, and <filename> is the name of the program to be run.

Program Name	Demonstrates...
DEP.CBL	Depreciation calculator.
DNS.CBL	How to obtain information about IP addresses or domain names.
DOWN.CBL	How to construct a download MIME header and use DISPLAYFILE and DISPLAYASCIIFILE.
EMAIL.CBL	Web based form for sending email with CobolScript.
EPRB.CBL	Problem Tracking entry editing.
HELLO.CBL	`Hello world` program.
HELLO1.CBL	Chapter 5 CGI Sample #1.
HELLO2.CBL	Chapter 5 CGI Sample #2.
HELLO3.CBL	Chapter 5 CGI Sample #3.
INPUT.CBL	How to accept CGI data.
OPER.CBL	Mathematical operator example program.
PAGE.CBL	Chapter 7 Sample.
PIC.CBL	CobolScript Picture clause variation.
PRB.CBL	Problem Tracking system example application.
SPRB.CBL	Problem Tracking entry submission.
UTS.CBL	Time Sheet example application.
VPRB.CBL	Problem Tracking entry viewing.
WEBBAN.CBL	How to print a Unix-style banner to a web page.

Appendix

CobolScript® Picture Clauses

The picture clause is a byte-by-byte definition of the format of a variable. It describes the general characteristics and editing requirements of an elementary data item, which can be either numeric or alphanumeric in CobolScript. For example, a picture clause of PIC X(1) represents a variable that has 1 byte of alphanumeric storage, while a picture clause of PIC 9(02) represents a variable that has 2 bytes of numeric storage (the zero in 9(02) is not required).

The general format of a variable definition is:

```
<level-number> <variable> PIC <picture-clause> [VALUE <value-literal>].
```

where *level-number* is the level number of the variable, *variable* is the variable name, *picture-clause* is the numeric or alphanumeric picture clause, and *value-literal* is an initial value for the variable, either an alphanumeric literal, such as `abc`, or a numeric literal, such as 157. Below are some example variable definitions:

```
1 variable_1 PIC X(10) VALUE `abcdefghij`.
1 variable_2 PIC Z,999 VALUE 123.
```

Following are definitions of the allowed components of CobolScript picture clauses.

Alphanumeric Picture Clauses

Alphanumeric picture clauses use an **X** to represent a single byte of storage; a single-byte alphanumeric variable will have a picture clause of PIC X, while a five-byte alphanumeric can be defined as PIC XXXXX, PIC X(5), or PIC X(05).

PIC X(n)

CobolScript also provides a special alphanumeric picture clause – PIC X(n). PIC X(n) can be used for FILLERs, or for any alphanumeric variables with initial values specified in VALUE clauses. PIC X(n), when specified, automatically calculates the length of the value specified in the VALUE clause, and allocates this number of bytes to the FILLER or variable. For example, the following FILLER definition:

```
5 FILLER PIC X(n) VALUE `testing`.
```

will allocate seven bytes of space to an alphanumeric FILLER variable as the variable is assigned the value `testing`. This can also be written in a shorthand as follows, eliminating the keywords from the definition:

```
5 `testing`.
```

The above form is called an *implied filler variable*, and can be used for any PIC X(n) FILLER variable that has a VALUE clause.

Numeric Picture Clauses

Basic numeric picture clauses use a **9** to represent a single byte of storage; a single-digit, single-byte numeric variable will have a picture clause of `PIC 9`, while a five-digit, five-byte numeric variable can be defined as `PIC 99999`, `PIC 9(5)`, or `PIC 9(05)`.

Signed Numeric Variables

Signed numeric variables use an **S** to represent the sign, followed by a normal numeric picture clause. The S indicates that a sign value will be maintained at the leftmost byte position. `PIC S9(05)` is an example of a signed numeric variable.

Implied Decimal Points

An implied decimal point in a numeric picture clause is represented by a **V**. The V indicates that an invisible decimal point will exist between the two digits on either side of the V. All calculations performed on this number will behave as if there is a normal decimal point in the location of the V. For example, a picture clause of `PIC 999V99` represents a five-digit number with an implied decimal; three of the digits are left of the decimal point, and two are post-decimal digits.

Literal Decimal Points

An actual decimal point in a numeric picture clause is represented by a period. The period indicates the position of the literal decimal point; the decimal point will display when the variable is displayed, and all internal calculations will be based on this decimal position. For example, a picture clause of `PIC 999.99` represents a five-digit number with two decimal places that requires six bytes of storage (an additional byte of storage is required for the period).

Numeric Edited Picture Clauses

Numeric edited picture clauses are numeric clauses in which certain symbols have been placed within the number, for purposes of clarity or legibility when the number is displayed. Like a literal decimal point, each edit symbol added to the picture clause requires an additional byte of storage.

Commas

Commas can be placed within a numeric picture clause to clarify number size, as in the numbers 1,000 and 2,345,678. The comma is placed between the digits of the picture clause where it should appear when the number is displayed. For instance,

```
PIC 99,999 VALUE 45678.
```

would display as `45,678`.

Zero Suppression

Leading zeros can be suppressed in numbers by replacing the usual **9** with a **Z** wherever the suppression is desired. Suppression terminates at the first non-zero digit in the number, or at the first 9 in the picture clause, whichever comes first. For instance, if two variables have the following picture clauses:

```
PIC Z,Z99.99 VALUE 123.55.
PIC Z,Z99.99 VALUE 3.55.
```

the numbers will display as `` `  123.55` `` and `` `   03.55` ``, respectively.

Floating Dollar Sign

In addition to suppressing leading zeros like a **Z**, dollar signs in a numeric picture clause will force a **$** to be displayed in place of the rightmost zero that is suppressed. For example, the picture clause:

```
PIC $,$$$.99 VALUE 123.55.
```

will display as `` `  $123.55` ``.

Asterisk Check Protection

The replacement of leading zeros in numeric character positions with asterisks is indicated by the use of the ***** symbol. For example, the picture clause:

```
PIC *,***.99 VALUE 123.55.
```

will display as `` `**123.55` ``.

Floating Plus Sign

In addition to suppressing leading zeros like a **Z**, plus signs in a numeric picture clause will force a **+** to be displayed in place of the rightmost zero that is suppressed if the number is positive, and a **–** if the number is negative. For instance, if two variables have the following picture clauses:

```
PIC +,+++.99 VALUE 123.55.
PIC +,+++.99 VALUE -123.55.
```

the numbers will display as `` ` +123.55` `` and `` ` -123.55` ``, respectively.

Floating Minus Sign

In addition to suppressing leading zeros like a **Z**, minus signs in a numeric picture clause will force a **–** to be displayed in place of the rightmost zero that is suppressed if the number is negative. For instance, if two have the following picture clauses:

```
PIC -,---.99 VALUE 123.55.
PIC -,---.99 VALUE -123.55.
```

the numbers will display as `` `  123.55` `` and `` ` -123.55` ``, respectively.

Plus Sign Control

Numeric edited variables with plus sign control have a plus sign as the rightmost symbol in the picture clause. When the variable has a positive value, a **+** will be displayed as the rightmost symbol in the number; when the variable has a negative value, a **–** will be displayed as the rightmost symbol. For instance, if two variables have the following picture clauses:

```
PIC Z,999.99+ VALUE 123.55.
PIC Z,999.99+ VALUE -123.55.
```

the numbers will display as `` `  123.55+` `` and `` `  123.55-` ``, respectively.

Minus Sign Control

Numeric edited variables with minus sign control have a minus sign as the rightmost symbol in the picture clause. When the variable has a negative value, a **–** will be displayed as the rightmost symbol; when it has a positive value, the minus sign will be suppressed. For instance, if two variables have the following picture clauses:

```
PIC -,---.99 VALUE 123.55.
PIC -,---.99 VALUE -123.55.
```

the numbers will display as `  123.55 ` and `  123.55-`, respectively.

DB Control

Numeric edited variables with DB control have the letters **DB** in the rightmost position of the picture clause. When the variable has a positive value, DB will be displayed right of the number; when the variable has a negative value, **CR** will be displayed to the right of the number. For instance, if two variables have the following picture clauses:

```
PIC Z,999.99DB VALUE 123.55.
PIC Z,999.99DB VALUE -123.55.
```

the numbers will display as `  123.55DB` and `  123.55CR`, respectively.

CR Control

Numeric edited variables with CR control have the letters **CR** in the rightmost position of the picture clause. When the variable has a negative value, CR will be displayed to the right of the number; if the number is positive, display of the CR is suppressed. For instance, if two variables have the following picture clauses:

```
PIC Z,999.99CR VALUE 123.55.
PIC Z,999.99CR VALUE -123.55.
```

the numbers will display as `  123.55  ` and `  123.55CR`, respectively.

Replica and Occurs variables

Besides group item, alphanumeric elementary item, and numeric elementary item variables, two other variable forms are possible in CobolScript: REPLICA variables and OCCURS clause variables. For in-depth discussion of these two forms, refer to Chapter 3, *CobolScript Language Constructs*.

Picture Clause Examples

The following table contains examples that illustrate how literal values will display when moved to variables with certain CobolScript picture clauses. The Value of Sending Field and Displayed Result columns' text is highlighted to accentuate any spaces that may or may not be displayed.

Picture Type	Value of Sending Field	Picture of Receiving Field	Displayed Result
Alphanumeric	12345678901234567890	X(20)	12345678901234567890
Alphanumeric	`Cbscript`	XXXXXXXXXX	Cbscript
Numeric	12345	99999	12345
Signed Numeric	12345	S9(05)	+12345
Signed Numeric	-12345	S9(05)	-12345
Numeric with Implied Decimal	12345	9(05)V99	1234500
Numeric with Implied Decimal	12345	99999V99	1234500
Numeric with Literal Decimal	12345	99999.99	12345.00
Numeric Edited with Zero Suppression	12345	ZZZZZZZZ	12345
Numeric Edited with Zero Suppression And Comma	12345	ZZ,ZZZ,ZZZ.99	12,345.00
Numeric Edited with Floating Dollar Sign	12345	$$,$$$,$$$.99	$12,345.00
Numeric Edited with Asterisk Check Protection	12345	**,***,***.99	****12,345.00
Numeric Edited with Floating Plus Sign	12345	++,+++,+++.99	+12,345.00
Numeric Edited with Floating Plus Sign	-12345	++,+++,+++.99	-12,345.00
Numeric Edited with Floating Minus Sign	12345	--,---,---.99	12,345.00
Numeric Edited with Floating Minus Sign	-12345	--,---,---.99	-12,345.00
Numeric Edited with Plus Sign Control	12345	ZZZ,ZZZ.99+	12,345.00+
Numeric Edited with Plus Sign Control	-12345	ZZZ,ZZZ.99+	12,345.00-
Numeric Edited with Minus Sign Control	12345	ZZZZZZZ-	12345
Numeric Edited with Minus Sign Control	-12345	ZZZZZZZ-	12345-
Numeric Edited with DB Control	12345	ZZZ,ZZZ.99DB	12,345.00DB
Numeric Edited with DB Control	-12345	ZZZ,ZZZ.99DB	12,345.00CR
Numeric Edited with CR Control	12345	ZZZCR	345
Numeric Edited with CR Control	-12345	ZZZCR	345CR

Appendix

F

CobolScript® Basic Program Structure

CobolScript does not explicitly require a fixed framework of divisions, sections, and modules in every program. Instead, CobolScript variable definitions, file descriptions, and procedural statements can be placed in any location in a program and be considered valid. However, basic COBOL program framework is supported by CobolScript for former COBOL programmers to use, if desired. This appendix defines the components of that framework, for those programmers that wish to follow older coding conventions. Again, all header sentences are optional, and variable definitions and code statements may still appear anywhere within a program (except within the Identification and Environment divisions, if they are included in your program).

CobolScript Program Templates

The code below is a basic, complete CobolScript program template. CobolScript enforces conventions for code placement and commenting, so the column position, or offset, of your code is relevant; comments must begin with an asterisk in the seventh column, and all code statements must begin after the seventh column. (Column 7 is the seventh character position from the left edge of the text file that contains your code.) Note the lack of division and section headers, and the FD statement or variable definition placed between procedural statements:

```
* Comments begin with an * in the 7th column.
*****************************************
 FD `<filename>` RECORD IS <record-size-in-bytes> BYTES.

 1 <variable> PIC <picture-clause> VALUE `<value>`.
 1 <group-item-name>.
   5 <variable> PIC <picture-clause> VALUE `<value>`.
 1 <occurs-variable> OCCURS <dimension> TIMES PIC <picture-clause>.

 <procedural statement>.
 .
 .
 <FD statement or variable definition>.
 <procedural statement>.
 .
 .
```

```
 <STOP RUN>.
```

In contrast, the following code is a template of a CobolScript program that makes use of COBOL division and section headers:

```
 IDENTIFICATION DIVISION.
************************************************************
* Comment lines are any lines with an asterisk in column 7  *
************************************************************
 PROGRAM-ID.      <this-filename>.
 AUTHOR.          <authors-name>.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE COMPUTER. <source-computer-type>.
 OBJECT COMPUTER. <object-computer-type>.

 DATA DIVISION.
 FILE SECTION.
 FD <filename> RECORD IS <record-size-in-bytes> BYTES.

 WORKING-STORAGE SECTION.
 01 <data-item-name>     PIC <picture-clause> VALUE `<value>`.
 01 <group-item-name>.
     05 <data-item-name> PIC <picture-clause> VALUE `<value>`.
 01 <table-name> OCCURS <table-size> TIMES.
     05 <data-item-name> PIC <picture-clause> VALUE `<value>`.

 PROCEDURE DIVISION.
 <module-name>.
      <procedural statement>.
      .
      .
      <STOP RUN | GOBACK>.

 <module-name>.
      <procedural statement>.
      .
      .
 .
 .
```

Division headers, section headers, and module names, when used, must begin in column 8 or higher, just like code statements.

There are four program division headers, or sentences, that can optionally be specified in CobolScript programs. These headers indicate the beginning of a particular division to the CobolScript engine. The division headers are:

- The Identification Division sentence;
- The Environment Division sentence;
- The Data Division sentence;
- The Procedure Division sentence.

The entire content of the Identification and Environment divisions is treated as informational only by the CobolScript engine. Thus, no program logic or variable definitions can be specified in either of these divisions. Instead, they are used to describe the program and the machines it will be executed on.

The Data division normally contains file and variable information. File Description statements (FDs), COPY statements that load variable-only copybooks, and variable definitions may all be located within the Data Division.

The Procedure division contains the program logic. In COBOL, this division is restricted to only allow code statements, but in CobolScript, variable definitions and file descriptions may also be included.

Although the division style of program layout may strike some programmers today as unwieldy, the existence of the divisions has a historical basis, and arguably still has merit. Originally, COBOL was designed to follow the layout of technical specifications, and the four program divisions were meant to match the divisions of the specifications. This layout is familiar to COBOL programmers, and it does lend itself well to maintenance and documentation.

Each program division begins with its own sentence, called a division header, which is just the name of the division, followed by a period, as in:

```
IDENTIFICATION DIVISION.
```

Some of the divisions can be further divided into sections, which have their own section header sentences. Convention dictates that the division and section header sentences are in all capital letters, as in the example above and in the program template on the previous page, but this is not a requirement in CobolScript.

Either the GOBACK or STOP RUN command is required as the last statement in the first module of a program, if there is more than one module in your program. These statements both terminate the execution of the program and prevent control from 'falling through' to subsequent modules.

Note that it is acceptable to exclude certain division headers from a program but not others, if division headers are being used. For instance, the Identification division sentence can be excluded from a program that uses the Environment, Data, and Procedure division sentences, but the Procedure division sentence should not be excluded from a program that includes the Identification division sentence.

For this reason, we recommend that you either use all of the division headers, or else completely exclude them from your programs, in order to avoid confusion. If you are a programmer who is accustomed to languages like C, *you will probably be most comfortable in completely excluding division and section headers from your programs*. If this is the case, you may opt to skip the remainder of this appendix.

Each of the divisions is described below, in the order that their header sentences should appear within a program, if you choose to use them.

The Identification Division

The Identification Division contains descriptive information about the program. This information is essentially for documentation purposes; because of this, and because the Identification Division is the first division of every program, it is also an excellent place to put a comment block that gives an overview of the program.

The Identification Division contains only four sentences: The PROGRAM-ID sentence and its argument sentence, and the AUTHOR sentence and its argument sentence. The PROGRAM-ID sentence and its argument describe the name of the program, as in:

```
PROGRAM-ID.    TEST.CBL.
```

The AUTHOR sentence and its argument name the creator of the program, as in:

```
AUTHOR.        B. SHAKE-SPEARE.
```

The Identification Division is strictly an optional component of CobolScript programs.

Here's what a complete Identification Division might look like:

```
 IDENTIFICATION DIVISION.
************************************************************
* Comment lines are any lines with an asterisk in column 7  *
************************************************************
 PROGRAM-ID.      TEST.CBL.
 AUTHOR.          DESKWARE.
```

The Environment Division

The Environment Division contains descriptive information about the computer that the program was developed on, and the computer that the code will be executed on. Like the Identification Division, the Environment Division is solely for informational purposes in CobolScript and is entirely optional.

In CobolScript, the Environment Division contains a single section called the Configuration Section. The Configuration Section contains four sentences: The SOURCE COMPUTER and its argument sentence, and the OBJECT COMPUTER and its argument sentence. The SOURCE COMPUTER describes the environment the source was developed on, as in:

```
SOURCE COMPUTER. FreeBSD.
```

The OBJECT COMPUTER describes the execution environment for this program, as in:

```
OBJECT COMPUTER. LINUX.
```

The Input-Output Section (a COBOL section) is not required or supported in CobolScript.

Here's what a complete Environment Division could look like:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE COMPUTER. WinNT.
OBJECT COMPUTER. FreeBSD.
```

The Data Division

The Data Division is the division where files may be described and variables defined.

Describing Files

The File Section of the Data Division is an optional section header that indicates where file description (FD) statements will be located. If you prefer, you can also place your record variable definitions in the File Section. The syntax of the FD statement is described in the **Data and Copybook Files** section of Chapter 3, *CobolScript Language Constructs*.

A File Section with one file description sentence and one record definition looks like this:

```
FILE SECTION.
FD `info.txt` RECORD IS 500 BYTES.
1 info_record.
  5  ir_cust_name  PIC X(18).
  5  ir_free_text  PIC X(480).
```

Note that specifying the File Section header does not preclude you from placing FD statements or record definitions elsewhere in your program.

Defining Variables

The Working-Storage Section of the Data Division is an optional section header that indicates where variables definitions are to be placed. Here's an example Working-Storage Section:

```
WORKING-STORAGE SECTION.
1 text_input PIC X(40).
1 order_info.
  2  cust_info.
    3  name    PIC X(12).
    3  loc     PIC $,999.99.
  2  order_amt PIC 99.
1 input_val PIC X(25).
```

Each CobolScript variable definition, whether group item or elementary item, has a level number and variable name, and elementary items also have picture clauses that define the variable's length. Variable definitions are described in detail in the **Variables** section of Chapter 3, *CobolScript Language Constructs*, and picture clause formats in Appendix E, *CobolScript Picture Clauses*.

For a more detailed explanation of level numbers and how to use them to manipulate data see Chapter 8, *Other Advanced Programming Techniques Using CobolScript*.

As with the File Section header, specifying the Working-Storage Section header does not preclude you from placing variable and record definitions elsewhere in your program.

Here's what one version of a complete Data Division might look like:

```
DATA DIVISION.
FILE SECTION.
FD `test.dat` RECORD IS 500 BYTES.
1 info_record.
  5  ir_cust_name PIC X(18).
  5  ir_free_text PIC X(480).

WORKING-STORAGE SECTION.
1 text_input PIC X(40).
1 order_info.
  2  cust_info.
     3  name   PIC X(12).
     3  loc    PIC $,999.99.
  2  order_amt PIC 99.
1 component_1  OCCURS 10 TIMES PIC 99.
```

The Procedure Division

The Procedure Division is where the logic of your program is located. Since CobolScript code is executed sequentially, the first line of code in the Procedure Division of a CobolScript program is the line that will be performed first. Good programming practice warrants the use of modules (also known as *paragraphs*) however, so your first code sentence after the Procedure Division sentence would normally be the name of your first (parent) module. This is conventionally a name like MAIN, so that the first two lines of the Procedure Division might be:

```
PROCEDURE DIVISION.
MAIN.
```

This first paragraph is unique in that it should end with either the STOP RUN or GOBACK statement. These statements terminate the control flow of the program and prevent subsequent paragraphs from sequentially executing. A complete main module could look like this:

```
MAIN.
   DISPLAY `This is a test`.
   PERFORM MODULE-1.
   STOP RUN.
```

Code modularity is achieved by naming other paragraphs with *paragraph header sentences*, and then calling these other named paragraphs with the PERFORM statement, as in the PERFORM above. A paragraph header sentence is just a module name, with no spaces between characters, and a period following the name. Like MAIN, which is a paragraph header sentence, the code for a module begins with the line that follows the header sentence.

A particular paragraph ends where the subsequent one begins, i.e., immediately prior to the next paragraph header sentence. In the following example, the module MODULE-1 ends with the line prior to MODULE-2 (the MOVE statement) and MODULE-2 ends after the DISPLAY statement, since this is the last line of code in this particular program.

```
MODULE-1.
   PERFORM MODULE-2.
   MOVE var_1 TO var_2.
MODULE-2.
   DISPLAY `This is a test from MODULE-2`.
```

The entire Procedure Division in this example looks like this:

```
PROCEDURE DIVISION.
MAIN.
   DISPLAY `This is a test`.
   PERFORM MODULE-1.
   STOP RUN.
MODULE-1.
   PERFORM MODULE-2.
   MOVE var_1 TO var_2.
MODULE-2.
   DISPLAY `This is a test from MODULE-2`.
```

For a detailed description of how to code in a modular fashion and how to use modules, see Chapter 8, *Other Advanced Programming Techniques Using CobolScript*.

Appendix

G

Setting Up ODBC and ODBC Data Sources for LinkMaker™

CobolScript LinkMaker™ is the database conduit technology that is integrated with CobolScript Professional Edition. LinkMaker™ uses the Open DataBase Connectivity (ODBC) specification to connect to a broad range of data sources such as DB2®, Oracle®, Informix®, MS SQL Server®, MS Access®, Postgres®, and MySQL™.

For LinkMaker™ to access data sources properly, so that you will be able to embed SQL in your CobolScript programs, some setup and configuration is required. In Microsoft® operating systems, this setup is relatively simple, while on Unix platforms it's slightly more involved; this is because the normal Unix environment will not have ODBC connectivity software already installed. In this chapter, we provide step-by-step instructions for configuring your data sources in both Windows® and Unix environments, and for setting up unixODBC, an open source ODBC connectivity package for Unix environments.

Prior to configuring a data source, you must obtain and install an ODBC driver for the data source you wish to access. Microsoft® operating systems come with many popular drivers, and many others are available from database and driver vendors. For Unix platforms, the authors of unixODBC provide several database drivers, and several other Unix database vendors provide ODBC drivers. Complete lists of drivers and vendor/author contact information are in the sections titled **Microsoft Windows® ODBC Drivers** and **UnixODBC ODBC Drivers** at the end of this appendix.

Note that the setup information in this appendix is for products not developed or supported by Deskware. For this reason, but it is not guaranteed to be current or complete, since Deskware has no control over these external products or their evolution.

Setup and Configuration in Windows® Environments

In most Windows® environments, ODBC will already be installed (an ODBC icon will be visible in the Windows® Control Panel if ODBC is installed). If ODBC is not already on your Windows® machine, you should install it from your Windows® CD prior to continuing.

Once you've confirmed that ODBC is installed, the only setup tasks required prior to using LinkMaker™ are:

1. Obtaining a driver for your data source, if one is not already present on your computer;

2. Configuring the data source.

The following subsection describes the steps necessary to configure your data source so that it can be accessed directly by LinkMaker™.

Configuring an ODBC Data Source in Windows®

1. From the Windows® *Start menu*, select *Settings* and then *Control Panel*. This will bring up your Microsoft Windows® Control Panel, as shown in Figure G.1. You should see an icon named *ODBC Data Sources (32bit)*. If this icon does not exist, you must install ODBC from your Windows® CD before proceeding.

Figure G.1 – The Microsoft Windows Control Panel.

2. Double-click on the *ODBC Data Sources (32bit)* icon. This will bring up the *ODBC Data Source Administrator* window as shown in Figure G.2.

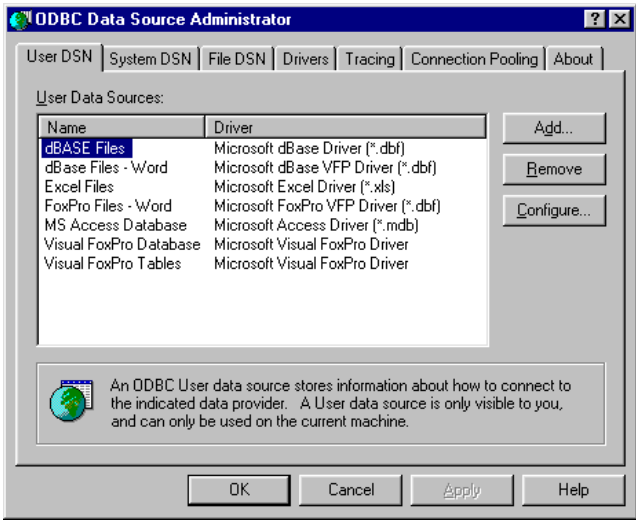


Figure G.2 – The ODBC Data Source Administrator.

3. From the *ODBC Data Source Administrator*, click on the *Add* button. This will bring up the Create New Data Source window, as shown in Figure G.3. From here you will be able to create a new data source using any ODBC drivers that are installed on your system. Windows® comes with ODBC drivers for many data sources; if your data source is not listed in this window, you will need to purchase a driver from a vendor. This appendix contains an exhaustive list of vendors that produce ODBC drivers. See the section later in this appendix titled **Microsoft Windows® ODBC Drivers** for more information.

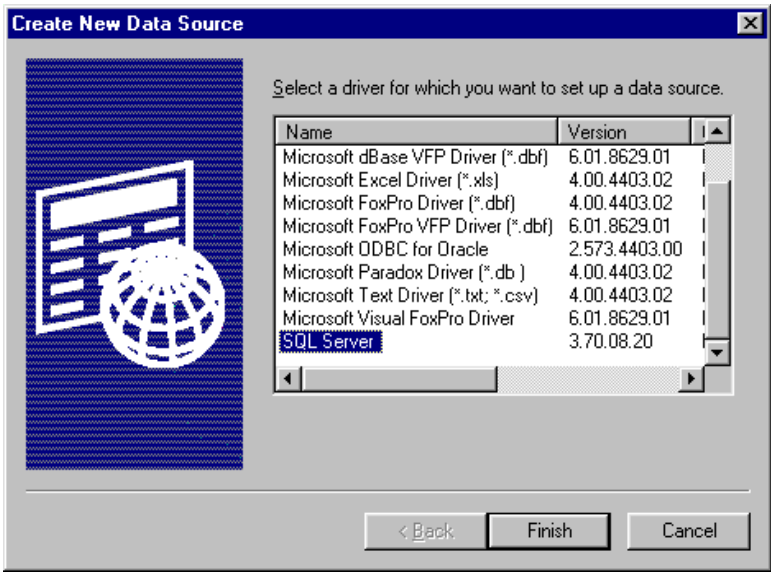


Figure G.3 – Creating a new data source in Windows®.

4. Select an ODBC driver for which you want to set up a data source. In Figure G.3, we have selected *SQL Server*. Click on the *Finish* button to bring up the next window.

5. In this next window (see Figure G.4), you will give your ODBC data source a name. This is the name that you will use as the first argument of the CobolScript OPENDB command to connect to the data source. In this example we will use *deskware*. You can also provide a description of the data source name in the *Description:* input box. The last input box is the *Server*. This refers to the name of the server that is hosting the data source. In this example we select *(local)* because our MS SQL Server database is on this machine. Because this window could differ from Figure G.4 depending on your ODBC driver, your configuration here may be slightly different. After you have entered all required information, click on the *Next* button.

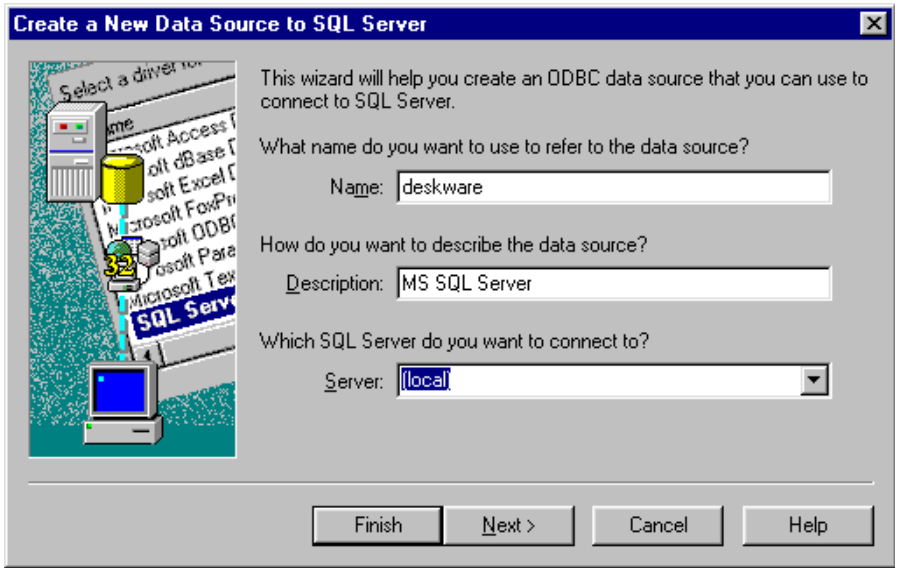


Figure G.4 – Selecting a data source.

6. Next, you will see a window similar to Figure G.5 that allows you to enter additional information about the data source that you are connecting to. You may want to enter a data source *Login ID* and *Password* here. If you do not know what Login ID and Password to enter here, you should contact your database administrator. The window that you see here may differ slightly from the one in Figure G.5 depending on the ODBC driver that you are installing. After you have finished, click on the *Next* button.

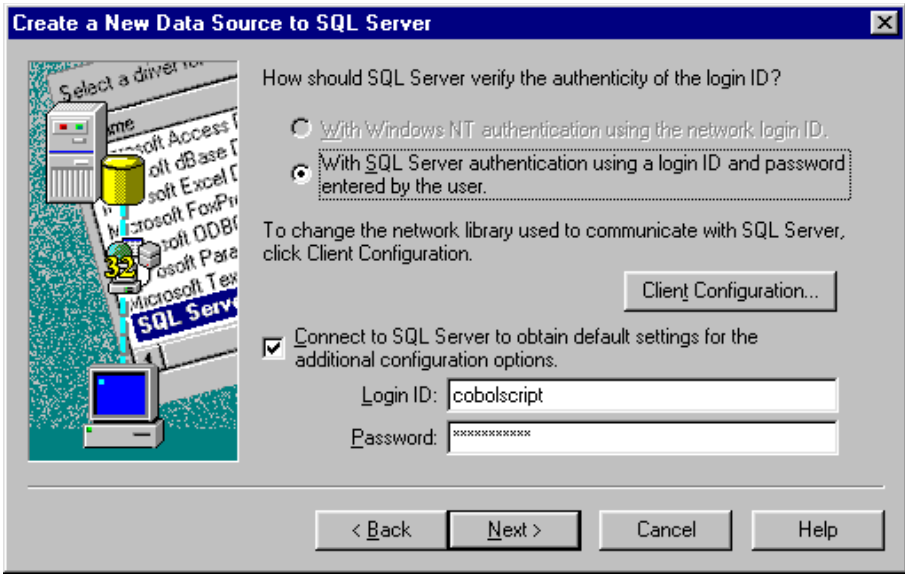


Figure G.5 – Entering a data source Login ID and Password.

7. Depending on your driver, the next window (see Figure G.6) may allow you to specify log files that will record and calculate statistics about the queries you send to the data source. These file are good audit trails, but they do slow down the performance of your application. After you have selected the options you want, click on the *Finish* button.

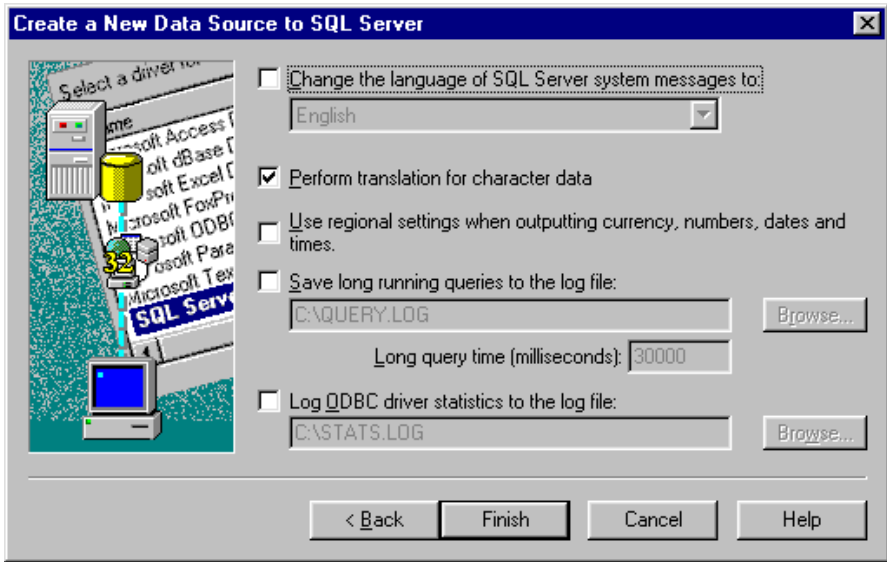


Figure G.6 – Specifying data source log files.

8. For most drivers, Figure G.7 is the last window you will see before your data source is set up. It gives you a chance to review the options you have selected. If you need to modify any of these, click on the *Cancel* button and you will be able to go back and change them. If not, click on the *OK* button.

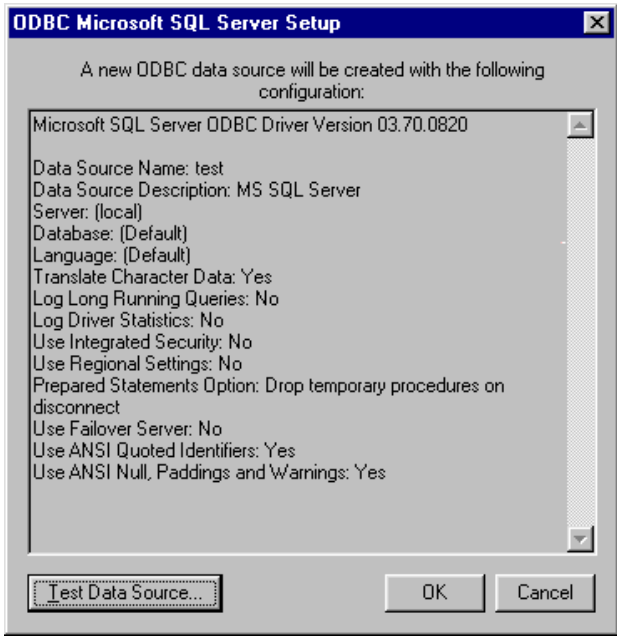


Figure G.7 – Verify data source settings.

9. After you have clicked on the *OK* button, you will be back at the *ODBC Data Source Administrator* window. If you are finished adding data sources, click on the *Cancel* button to exit. If you want to check the settings for the data source that you just added, select it and click on the *Configure* button.

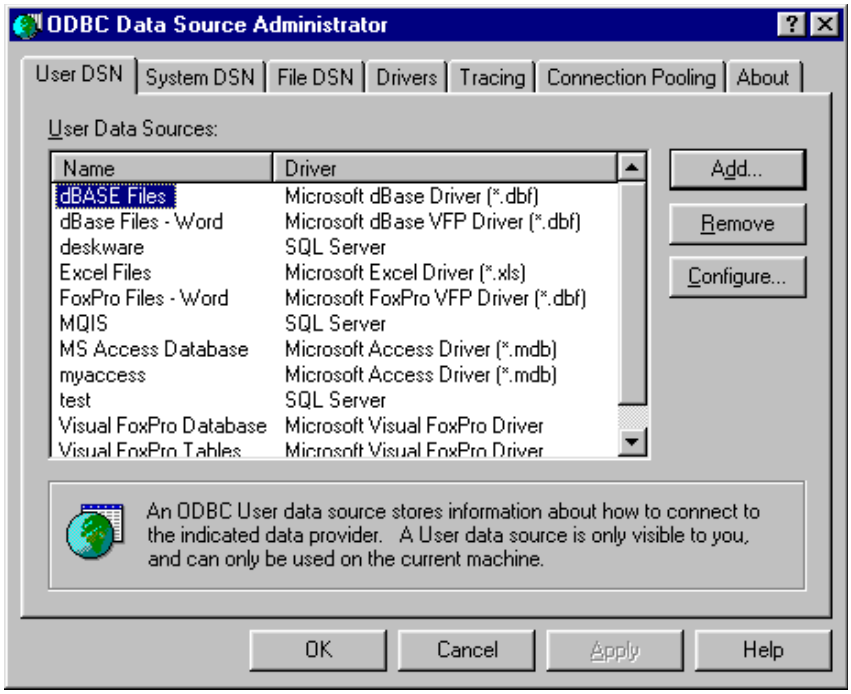


Figure G.8 – ODBC Data Source Administrator.

10. Now you will be able to connect to databases with CobolScript for Windows®. See Appendix H for more information on embedded SQL programming in CobolScript. Remember that you need to use the data source name that you set up and a valid data source Login ID and password.

Setup and Configuration in Unix Environments

UnixODBC Installation and Setup

You must install unixODBC in order to use the LinkMaker™ version of CobolScript Professional for Linux®, FreeBSD®, or SunOS®. UnixODBC also has a GUI (Graphical User Interface) component that requires installation of the QT graphics library, but installation of the GUI is not required in order to use LinkMaker™. If you are running Linux with a kernel older than 2.2.12, you **should not** install the GUI. Also, if you are interested in getting started as quickly as possible and with the least effort, and you are comfortable working in a non-graphical environment, you can skip the GUI installation.

The next two subsections provide step-by-step instructions on how to install unixODBC. The first subsection describes installation without the GUI; the second, the more involved installation with the GUI component.

Installing unixODBC without the GUI

The unixODBC manager is a freely available open source package developed by unixODBC.org You will need to go to the unixODBC web site to obtain this software.

1. Go to the unixODBC site and download the unixODBC driver manager package. The web site is at http://www.unixodbc.org. You can go directly to the download page by typing the following in URL in your web browser:

   ```
   http://www.unixodbc.org/download.htm
   ```

2. Once you've located the unixODBC web site, download the latest copy of unixODBC to your hard drive. The name of the file will be *unixODBC*.tar.gz*, where * is the version number of the latest release. You should save this file to the /usr/local directory on your machine. If you save it to another directory, bring up a command prompt and go to that directory. Then copy the file to the /usr/local directory by typing the following at the command prompt:

   ```
   cp unixODBC*.tar.gz /usr/local
   ```

3. The unixODBC package is *tar'd* and compressed with the *gzip* format. You will need to uncompress it and un-tar it. Uncompress it by using the gunzip program and typing the following at the command prompt, making sure you are in the /usr/local subdirectory on your machine when you do this:

   ```
   gunzip unixODBC*.tar.gz
   ```

4. You will now have what is called a *tar* file. This file contains many other files. Un-tar this file by typing the following command at the system prompt. It will create a new subdirectory in your /usr/local directory that will contain the unixODBC files. Again, make sure you are still in the /usr/local subdirectory on your machine when you type this command:

   ```
   tar -xvf unixODBC*.tar
   ```

5. Change your current directory to the unixODBC directory that was created by the tar command in step 4. Do this by typing:

   ```
   cd unixODBC*
   ```

6. You will be inside the unixODBC directory at this point. This directory contains the unixODBC *make* files. These *make* files will allow you to compile the unixODBC driver manager on your machine. Before you can run the *make* files you must configure them. Do this by entering the following at the command prompt:

   ```
   ./configure --enable-gui=no
   ```

   Notice that the option to the configure script is *—enable-gui=no*. This tells that configuration script that you wish to configure the *make* files for an installation of unixODBC without the graphical user interface.

7. After running the configuration script, you can compile the unixODBC package on your machine. You will do this by running the Unix *make* command. This command will look at a file named *makefile* in the current directory. This file was created by the configuration script. Enter the following at the command prompt to begin compiling the unixODBC package (it may take a few minutes to compile):

```
make
```

8. After you have run *make* and compiled the unixODBC package, install it on your machine by typing the following at the command prompt:

```
make install
```

9. Now unixODBC is compiled and installed on your system. This installation process creates various shared libraries on your machine and places them in the /usr/local/lib directory. In order for your system to recognize these libraries, you must directly edit the *ld.so.conf* file on your system and add a line to this file that contains */usr/local/lib* in it. To edit this file, type the following at the command prompt (consult your operating system's documentation for instructions on how to use the **vi** editor):

```
cd /etc
vi ld.so.conf
```

10. After you have edited and saved the *ld.so.conf* file, run the following from the command prompt. This will update your system that it can find the newly added shared libraries:

```
ldconfig
```

11. Next, you will set up the *odbcinst.ini* file for the data source you wish to access on this machine. You will need to go to the */usr/local/etc* directory and edit the file named *odbcinst.ini*:

```
cd /usr/local/etc
vi odbcinst.ini
```

12. You should consult the unixODBC documentation at http://www.unixodbc.org for additional information on how to edit *odbcinst.ini*. Here is an example of two entries in *odbcinst.ini*, one for MySQL and one for PostgreSQL:

```
[MySQL]
Description     = MySQL Driver
Driver          = /usr/local/lib/libmyodbc.so
Setup           = /usr/local/lib/libodbcmyS.so
FileUsage       = 1

[PostgreSQL]
Description     = PostgreSQL Driver
Driver          = /usr/local/lib/libodbcpsql.so
Setup           = /usr/local/lib/libodbcpsqlS.so
FileUsage       = 1
```

13. The next step is to create data source definitions in the *odbc.ini* file that is located in the */usr/local/etc* directory: Assuming you are still in the */usr/local/etc* directory, just type the following:

```
vi odbc.ini
```

14. Below are examples of *odbc.ini* file definitions for PostgreSQL and MySQL. Consult the unixODBC documentation at http://www.unixodbc.org for additional information on how to create these entries:

```
[PostgreSQL]
Description             = PostgreSQL
Driver                  = PostgreSQL
Trace                   = No
TraceFile               =
Database                = test
Servername              = localhost
UserName                = postgres
Password                = mypass
Port                    = 5432
Protocol                = 6.4
ReadOnly                = No
RowVersioning           = No
ShowSystemTables        = No
ShowOidColumn           = No
FakeOidIndex            = No
ConnSettings            =

[MySQL]
Description             = MySQL
Driver                  = MySQL
Trace                   = Yes
TraceFile               = /tmp/mysql.odbc.log
Server                  = localhost
Port                    = 3306
Database                = deskware
User                    = root
Password                = mypass
```

15. You are now ready to connect to a unixODBC data source using CobolScript LinkMaker™. Since you are working on a Unix platform, make certain that you have renamed the LinkMaker™-enabled version of CobolScript Professional Edition to *cobolscript.exe* and are using it instead of the default version of CobolScript Professional (see the readme.txt file included with Unix versions of CS Professional for more information). Also, be sure to remember to use the *database* name that you create in the *odbc.ini* file for your first argument of the CobolScript OPENDB command. See Appendix H for information on how to use SQL statements with CobolScript.

Installing unixODBC with the X Windows GUI

The GUI portion of the unixODBC package requires the QT graphics library. This library is freely available and can be obtained from a company called Troll Tech. Here are step-by-step instructions that explain where to get it and how to install this library.

1. The first step to installing unixODBC with the X Windows graphical user interface is to install the QT graphics library. Start a web browser and enter the following URL:

   http://www.trolltech.com/dl/qtfree-dl.html

2. After the above URL has successfully loaded, download the QT graphics library (the file named qt-2.0.2.tar.gz) to the */usr/local* directory on your machine. If you download it to another directory, bring up a command prompt and go to that directory. Now copy the file to the /usr/local directory by typing the following at the prompt:

   ```
   cp qt-2.0.2.tar.gz /usr/local
   ```

3. This file is a tar'd file that is compressed with the Unix *gzip* program. You will need to uncompress it. Do so by typing the following at the command prompt:

   ```
   gunzip qt-2.0.2.tar.gz
   ```

4. Now you will have a *tar* file in your */usr/local* directory. It contains many files. To extract these files, type the following at the command prompt. This will un-tar the files into a directory named */usr/local/qt-2.0.2*:

   ```
   tar -xvf qt-2.0.2.tar
   ```

5. You now need to change the name of the */usr/local/qt-2.0.2* directory to just plain */usr/local/qt*. You will accomplish this by typing the following at the command prompt:

   ```
   mv qt-2.0.2 qt
   ```

6. The next step involves setting environment variables in either your *.profile* or *.login* files, depending on which Unix shell you are using.

   In .profile (if your shell is bash, ksh, zsh or sh), add the following lines:

   ```
   QTDIR=/usr/local/qt
   PATH=$QTDIR/bin:$PATH
   if [ $MANPATH ]
   then
           MANPATH=$QTDIR/man:$MANPATH
   else
           MANPATH=$QTDIR/man
   fi
   if [ $LD_LIBRARY_PATH ]
   then
           LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
   else
           LD_LIBRARY_PATH=$QTDIR/lib
   fi
   LIBRARY_PATH=$LD_LIBRARY_PATH
   if [ $CPLUS_INCLUDE_PATH ]
   then
   ```

```
        CPLUS_INCLUDE_PATH=$QTDIR/include:$CPLUS_INCLUDE_PATH
else
        CPLUS_INCLUDE_PATH=$QTDIR/include
fi

export QTDIR PATH MANPATH LD_LIBRARY_PATH LIBRARY_PATH
export CPLUS_INCLUDE_PATH
```

In .login (if your shell is csh or tcsh), add the following lines:

```
if ( ! $?QTDIR ) then
    setenv QTDIR /usr/local/qt
endif
if ( $?PATH ) then
    setenv PATH $QTDIR/bin:$PATH
else
    setenv PATH $QTDIR/bin
endif
if ( $?MANPATH ) then
    setenv MANPATH $QTDIR/man:$MANPATH
else
    setenv MANPATH $QTDIR/man
endif
if ( $?LD_LIBRARY_PATH ) then
    setenv LD_LIBRARY_PATH $QTDIR/lib:$LD_LIBRARY_PATH
else
    setenv LD_LIBRARY_PATH $QTDIR/lib
endif
if ( ! $?LIBRARY_PATH ) then
    setenv LIBRARY_PATH $LD_LIBRARY_PATH
endif
if ( $?CPLUS_INCLUDE_PATH ) then
    setenv CPLUS_INCLUDE_PATH $QTDIR/include:$CPLUS_INCLUDE_PATH
else
    setenv CPLUS_INCLUDE_PATH $QTDIR/include
endif
```

7. After you have completed this step, you will need to either login again, or re-source the profile before continuing, so that the $QTDIR environment variable is set. This step is crucial; if you don't do this, the installation will fail. To re-source the profile, type one of the following (depending on whether you have a .profile or .login file) at the system prompt:

   ```
   source .profile
   source .login
   ```

8. Now that you have logged in again or re-sourced the profile, bring up a command prompt and enter the following command.

```
cd /usr/local/qt
```

9. You are now in the QT directory. This directory contains the configuration script that is required for the installation of the QT library. Run the configuration script by typing the following at the prompt:

```
./configure
```

10. The above step will create the *make* file that is required to compile QT on your system. You now need to run *make* to build QT on your system. Do this by entering the following at the command prompt (this step will take several minutes to complete):

```
make
```

11. You are now ready to install unixODBC with the graphical user interface. Go to the unixODBC web site and download the unixODBC driver manager package. Their web site is at http://www.unixodbc.org. You can go directly to the download page by typing the following in URL in your web browser:

```
http://www.unixodbc.org/download.htm
```

12. Now that you are at the unixODBC web site, download the latest copy of unixODBC to your hard drive. The name of the file will be *unixODBC*.tar.gz*, where * is the version number of the latest release. You should save this file to the /usr/local directory on your machine. If you save it to another directory, bring up a command prompt and go to that directory. Then copy the file to the /usr/local directory by typing the following at the command prompt:

```
cp unixODBC*.tar.gz /usr/local
```

16. Now that you are at the unixODBC web site, download the latest copy of unixODBC to your hard drive. The name of the file will be *unixODBC*.tar.gz*, where * is the version number of the latest release. You should save this file to the /usr/local directory on your machine. If you save it to another directory, bring up a command prompt and go to that directory. Then copy the file to the /usr/local directory by typing the following at the command prompt:

```
cp unixODBC*.tar.gz /usr/local
```

17. The unixODBC package is *tar'd* and compressed with the *gzip* format. You will need to uncompress it and un-tar it. Uncompress it by using the gunzip program and typing the following at the command prompt, making sure you are in the /usr/local subdirectory on your machine when you do this:

```
gunzip unixODBC*.tar.gz
```

18. Now that you have uncompressed the file, you will have what is called a *tar* file. This file contains many other files. Un-tar this file by typing the following command at the system prompt. It will create a new subdirectory in your /usr/local directory that will contain the unixODBC files. Again, make sure you are still in the /usr/local subdirectory on your machine when you type this command:

```
tar -xvf unixODBC*.tar
```

19. Change your current directory to the unixODBC directory that was created by the tar command in step 4. Do this by typing:

```
cd unixODBC*
```

20. You will be inside the unixODBC directory at this point. This directory contains the unixODBC *make* files. These *make* files will allow you to compile the unixODBC driver manager on your machine. Before you can run the *make* files you need to configure them. You will do this by entering the following at the command prompt:

```
./configure
```

21. Continue on by performing Steps 7-15 in the previous subsection, **Installing unixODBC without the GUI**.

unixODBC non-GUI Component (isql)

UnixODBC has a special interactive mode that can be entered with the *isql* command. You can connect to your data sources, send SQL commands to the data source, and receive results from the data source by using isql. You can start this tool by typing the following at the system prompt:

```
/usr/local/bin/isql
```

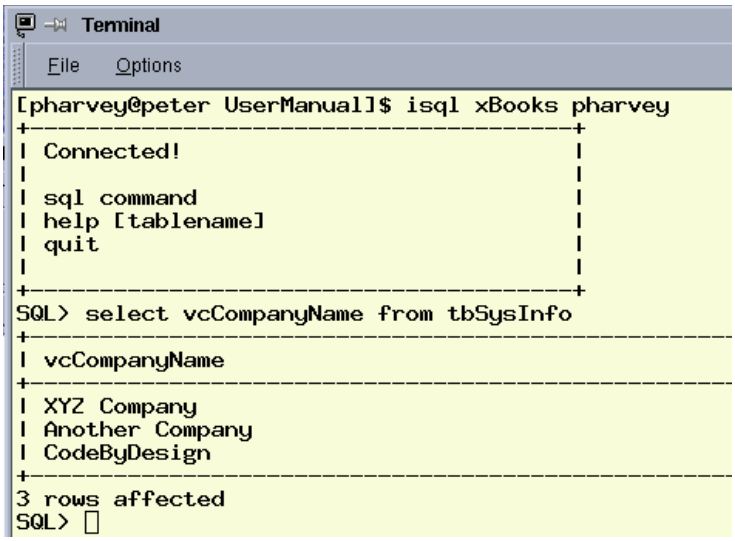


Figure G.9– unixODBC isql command line tool.

Specify an ODBC data source name as an argument in order to directly interact with your database, as in the following command line entry:

```
/usr/local/bin/isql MySQL
```

This will bring up an isql interactive session like the one shown in Figure G.9.

UnixODBC GUI Components

These components are only available if you have installed the GUI portion of unixODBC.

The *unixODBC Data Source Administrator*, or ODBCConfig, is a tool designed to allow you to easily set up data sources. Start the Data Source Administrator by typing the following at the system prompt:

```
/usr/local/bin/ODBCConfig
```

A window similar to Figure G.10 will appear. From here, you can add, remove, and configure data sources.

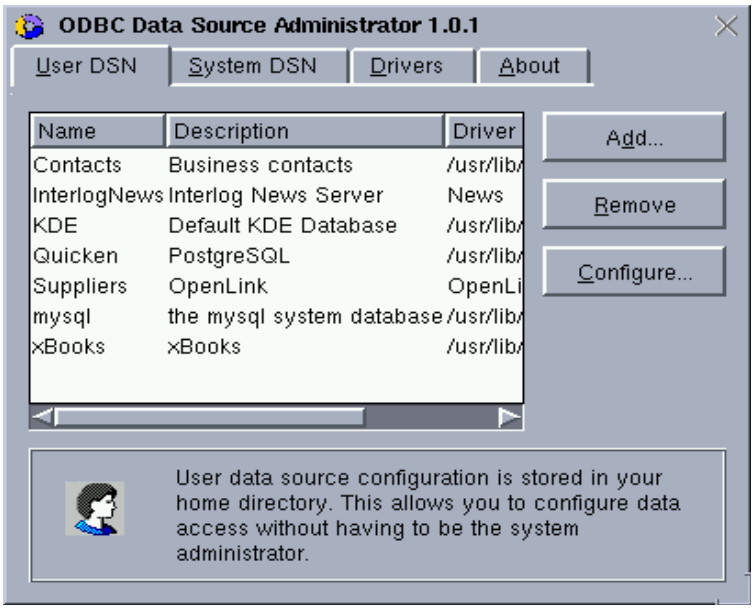


Figure G.10– The unixODBC Data Source Administrator.

Data Manager is a graphical tool for exploring data sources. Start the Data Manger by typing the following at the system prompt:

```
/usr/local/bin/DataManager
```

A window like the one in Figure G.11 will appear on your desktop. Using this tool, you can browse your data sources and execute SQL queries against them.

Microsoft Windows ODBC Drivers

Some ODBC drivers are included with the Microsoft Windows operating system. If you plan to work with a data source that Microsoft does not provide an ODBC driver for, you will need to purchase an ODBC driver from one of these companies.

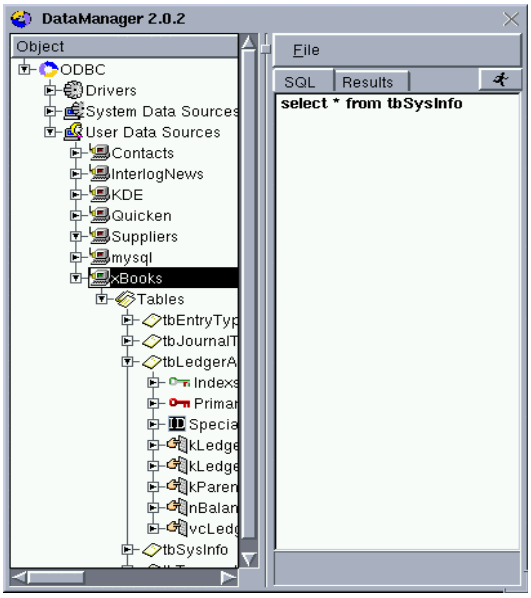


Figure G.11– The unixODBC Data Manager

Company	ODBC Driver	Data Source
Acucorp, Inc. Telephone: 619-689-4500 Fax: 619-689-4550 http://www.acucobol.com	AcuODBC Vision Driver	Vision indexed file system

Company	ODBC Driver	Data Source
Aonix Telephone: 415-543-0900 Fax: 415-543-0145 http://www.aonix.com	NOMAD RP/Server OD/Server	DB2 Teradata IMS IDMS ISAM QSAM VSAM
Applied Information Services Telephone: 301-489-1024 Fax: 301-489-1021 http://www.uniaccess.com	UniAccess ODBC Server	Unisys RDBMS MAPPER
Applix, Inc. Telephone: 508-870-0300 Fax: 508-366-2278 http://www.applix.com	TM1 ODBC	TM1 databases
Ardent Software Corporation Telephone: 508-366-3888 Fax: 508-366-3669 http://www.ardentsoftware.com	UniVerse ODBC UniDesktop ODBC	UniVerse UniData RDBMS
August Software Corp. Telephone: 714-454-9007 Fax 714-454-9032 http://www.augsoft.com	OverDriver ODBC Router (server)	Data sources with ODBC drivers
Autodesk Inc. Telephone: 415-507-5000 Fax: 415-507-5100 http://www.autodesk.com	AutoCAD ODBC Driver	AutoCAD SQL Extension (ASE)
Automation Technology Inc. Telephone: 408-473-0200 Fax: 408-473-0201 http://www.atinet.com	OpenAccess ODBC SDK OpenRDA ODBC drivers	Any non-SQL database and SQL Server, Microsoft Access from non-Window platforms
BORN Information Services Telephone: 612-404-4000 Fax: 612-404-4444 http://www.born.com	ODBC for the AS/400	IBM AS/400

Company	ODBC Driver	Data Source
Bull Worldwide Information Systems Telephone: 602-980-8575 Telephone: 602-862-6062 Fax: 602-862-3606 http://www.bull.com	DDA ODBC	Oracle DB2 Informix Teradata IMS VSAM OpenIngres Rdb RMS IDS II RFM II UFAS
Byte Designs, Ltd. Telephone: 604-534-0722 Fax: 604-534-2601 http://www.bytedesign.com	Byte Design ODBC	C-ISAM D-ISAM DISAM96 Informix
Centura Software Corp. Telephone: 415-321-9500 Fax: 415-321-5471 http://www.centurasoft.com	ODBC Drivers for SQLBase SQLHost/DB2	SQLBase DB2 Velocis
Cincom Systems Telephone: 513-662-2300 Fax: 513-459-7145 http://www.cincom.com	Supra ODBC Driver UniSQL ODBC Driver	Supra Server UniSQL (except Japan)
Computer Associates Int'l, Inc. Telephone: 800-225-5224 http://www.cai.com	CA-IDMS Server ODBC CA-Visual Express ODBC Driver for Ingres CA-Datacom Server	IDMS OpenIngres IBM DB2 IMS Rdb RMS VSAM AllBase/SQL Image/SQL CA-Datacom/DB
Computer Corporation of America http://www.cca-int.com	ODBC V2, Connect*	System 1032 Model 204
Computer Solutions Limited Telephone: +44 (0) 1905 794400 Fax: +44 (0) 1905 794 464 http://www.csllink.com/products.html	Linkway 32	AllBase/SQL Image/SQL
Cornerstone Telephone: 603-595-7480 Fax: 603-882-7313 http://www.corsof.com	Dyna Access	Tandem NonStop SQL Tandem Enscribe
Cross Access Corp. Telephone: 408-735-7545 Fax: 408-735-0328 http://www.crossaccess.com	Cross Access ODBC Driver	Adabas Datacom DB2 DL/1 IDMS IMS Oracle Sequential VSAM
Datafit Ltd. Telephone: 011-44-1-480-454-604	Datafit DP4	Datafit DP4

Company	ODBC Driver	Data Source
Decision Support, Inc. Telephone: 704-849-8904 Fax: 704-487-4875 http://www.dsinc.com	UniStar ODBC Driver	DMS II DARGAL server
Dharma Systems Telephone: 603-886-1400 Fax: 603-883-6904 http://www.products.dharma.com	Dharma ODBC SDK ODBC SDK Lite	BASISplus GT.M PROMIS custom drivers
Doric Computer Systems International Telephone: 800-223-2942 Telephone: 206-367-7974 http://www.doric.com	INFO~ODBC Direct Client INFO~ODBC Server	C-ISAM D-ISAM ESRI ARC/INFO Coverages INFO DBMS & 4G/L
Egan Systems, Inc. Telephone: 800-645-9898 Telephone: 516-588-8000 Fax 516-588-8001 http://www.egns.com/ODBC/	Interactive COBOL ODBC Driver	ICOBOL Server COBOL files
Easysoft Ltd. Telephone: +44 (0)1132220400 Fax: +44 (0)1132220500 http://www.easysoft.com	ODBC for RMS ODBC for ISAM ODBC for CODA ODBC for ROSS	RMS files C-ISAM D-ISAM T-ISAM
EasiRun Software Telephone: 619-587-0467 Fax: 619-587-0466 http://www.easirun.com	EasiODBC Relativity USQL Client-Server drivers	AcuCOBOL files RM-COBOL EXTFH C-ISAM Business BASIC ISAM U/FOS
Empress Software Inc. Telephone: 301-220-1919 Fax 301-220-1997 http://www.empress.com	Empress ODBC driver	Empress
Ensodex, Inc. Telephone: 612-766-8787 Fax 612-766-8792 http://www.ensodex.com	Hot Sockets ODBC Driver (server)	Data sources with 32-bit ODBC drivers
Esker, Inc. Telephone: 415-675-7777 Fax: 415-675-7775 http://www.esker.com	ODBC Driver Pack TunSQL	C-ISAM D-ISAM IBM DB2 Informix Oracle Progress Sybase
EveryWare Development Corp. Telephone: 905-819-1173 Fax 905-819-1172 http://www.everyware.com	Butler SQL ODBC Driver	Butler SQL
Farabi Technology Corp. Telephone: 800-565-3455 Telephone: 514-332-3455 Fax 514-332-3915 http://www.farabi.com	ODBC for Ultima/400	AS/400
FairCom Corp. Telephone: 800-234-8180 Telephone: 314-445-6833 Fax 314-445-9698 http://www.faircom.com	FairCom ODBC Driver	FairCom Server c-tree Plus

Company	ODBC Driver	Data Source
FFE Software Telephone: 510-232-6800 Fax 510-237-7433 http://www.firstsql.com	FirstSQL ODBC	FirstSQL dBASE
Filemaker, Inc. Telephone: 800-325-2747 Telephone: 408-987-7000 http://www.filemaker.com	Filemaker Pro ODBC Driver	Filemaker
FLEXquarters Telephone: 602-732-9217 Fax: 602-732-9590 http://www.flexquarters.com	Flex/ODBC	DataFlex files
M.B. Foster Associates Telephone: 613-448-2333 Fax: 613-448-2588 http://www.mbfoster.com/	DataExpress ODBCLink	HP 3000 Allbase TurboImage
Fulcrum Technologies, Inc. Telephone: 613-238-1761 Fax: 613-238-7695 http://www.fultech.com	Fulcrum SearchServer	Fulcrum Search Server
Generix Limited Telephone: +44 (0) 1924 500151 Fax: +44 (0) 1924 500515 http://www.generix.ltd.uk	CONNX	DB2 Oracle RDB RMS
gfs Gesellschaft fur Informationssysteme mbH Telephone: +49-40-450232-0 Fax: +49-40-450232-66	ODBC-Rocket	BS2000 DBMS SESAM/SQL UDS/SQL LEASY ISAM
Harbinger Corporation Telephone: 800-555-2989 Telephone: 404-467-3000 Fax: 404-841-4364 http://www.harbinger.com	STX for Windows	STX
Hill Croft Information Technologies Telephone: +44 1908 666244 Fax: +44 1908 666244 http://www.LinkEase.co.uk	LinkEase	DataEase
HiT Software, Inc. Telephone: 408-369-7290 Fax: 408-369-7299 http://www.hit.com	HS*ODBC	AS/400 DB2
Hewlett Packard Telephone: 800-637-7740 http://www.hp.com	AllBase Image	AllBase/SQL Image/SQL
HOB GmbH & Co. KG Germany Brandstaetterstrasse 2-10 D-90513 Zirndorf Telephone: +49-911-9666-393 Fax: +49-911-9666-271 http://www.hob.de	HOBLink DRDA	DB2 OS/390 DB2 UDB DB2 MVS DB2 VSE&VM DB2/400 DB2/6000 DB2/2 VSAM* IMS/DB* DL/1* * requires HOBDB online

Company	ODBC Driver	Data Source
IBM Corp. Telephone: 800-IBM-4YOU http://www.software.ibm.com/data/db2/db2connect	DB2 Connect	DB2 for OS/390 DB2 for MVS/ESA DB2/400 DB2 for VSE and VM DB2 UDB (UNIX, Windows NT and OS/2 servers)
IBM Corp. http://www.networking.ibm.com/gso/gsohome.html	Connection ODBC Client	Connection Server
Information Builders, Inc. Telephone: 800-969-4636 Telephone: 212-736-4433 Fax 212-629-8819 http://www.ibi.com	EDA/Extender for ODBC EDA/SQL (server)	DB2 IMS VSAM IDMS Datacom TOTAL Teradata ADABAS Oracle SQL Server OpenIngres Informix Supra Server SQL/DS RMS Rdb M Sharebase C-ISAM Image/SQL Allbase/SQL Others
Informix Software, Inc. Telephone: 800-388-0366 Telephone: 415-926-6300 Fax: 913-599-8753 http://www.informix.com	Informix ODBC Driver	Informix OnLine Dynamic Server, SE
Intersoft, Inc. http://www.inter-soft.com	Essentia ODBC	Essentia SQL-Server
Intersolv, Inc. (see MERANT)	DataDirect ODBC Drivers SequeLink	See MERANT
IQ Software Telephone: 770-446-8880 Fax: 770-448-4088 http://www.iqsc.com	IQ Smart Server	Several DBMSs

Company	ODBC Driver	Data Source
ISG International Software Group Telephone: 781.221.1450 Fax: 781.272.2531 http://www.isgsoft.com	ISG Navigator/ODBC ISG Navigator/Bridge	ADABAS Btrieve C-ISAM D-ISAM DB2 IMS Informix Mumps OpenIngres Oracle SQL Server Sybase Rdb Red Brick RMS TANDEM Enscribe TANDEM SQL-MP TANDEM SQL-MX Text VSAM ODBC data sources, OLEDB data sources any 3GL Application (Application connector), any database (SDK) Bridge to OLE DB providers (Windows, NT and Unix)
Javera Software, Inc. Telephone: 412-397-4061 Fax 412-397-4062 http://www.javera.com	JetConnect	Any ODBC data source
KB Systems, Inc. Telephone: 703-318-0405 Fax: 703-318-0569 http://www.kbsystems.com	KB_SQL ODBC Driver	KB_SQL M
Kerridge Computer Company, Ltd. Telephone: +44(1635) 523456 Fax: +44(1635) 30300 http://www.kerridge.com	K-ISAM ODBC Driver	K-ISAM
KE Software, Inc Telephone: 604-877-1960 Fax: 604-877-1961 http://www.kesoftware.com	TexODBC	KE Texpress ODBMS
Liant Software Corporation Telephone: 800-349-9222 Telephone: 508-872-8700 Fax 508-626-2221 http://www.liant.com	Relational Data Bridge (formerly Relativity)	VSAM ISAM Btrieve RMS Micro Focus COBOL files RM/COBOL files

Company	ODBC Driver	Data Source
Liberty Integration Software Telephone: 604-682-8293 Fax: 604-682-8499 http://www.libertyodbc.com	Liberty ODBC Driver	PICK UniData UniVerse PI-Open Sequoia/Pro GA-Power95 GA-R91 mvBase jBase Reality/X UltPlus Alpha Microsystems Mentor/Pro Advanced PICK D3
Lotus Development Corp. Telephone: 617-577-8500 Fax: 617-693-6080 http://www.lotus.com	NotesSQL Driver	Lotus Notes
Marxmeier Software GmbH Telephone: +49 202 24314-40 Fax: +49 202 24314-20 http://www.msede.com	SQL/R ODBC (server)	HP Eloquence databases
MEGAsoft, Inc. Telephone: 407-423-0460 Email: 72702.1303@compuserve.com	ODBC driver for MEGAdata	PASSdata MEGAdata
MERANT Telephone: 800-876-3101 Telephone: 919-461-4200 Fax 919-461-4526 http://www.merant.com/datadirec	DataDirect ODBC Drivers DataDirect SequeLink (server)	Btrieve Clipper DB2 dBASE FoxPro Excel Informix OpenIngres Oracle Paradox Progress AS/400 SQL/DS SQLBase SQL Server Teradata Text XDB DB2-DDCS/2 MDI Gateway Sybase Net-Gateway IBM DB2 Oracle Microsoft SQL Server Sybase Informix OnLine and SE OpenIngres any ODBC-compliant database

Company	ODBC Driver	Data Source
Micro Data Base Systems Inc. Telephone: 800-445-6327 Telephone: 765-463-7200 Fax 765-463-1234 http://www.mdbs.com	Titanium ODBC Driver GURU ODBC Driver	Titanium GURU KnowledgeMan Object/1
Micro Focus (see MERANT) Telephone: 415-856-4161 Fax: 415-856-6134	Correlate	Micro Focus files
Microrim Inc. (see R:BASE Technologies)	Ottero ODBC	Ottero
Microsoft Corp. Telephone: 800-426-9400 Telephone: 206-882-8080 Fax: 206-936-7329 http://www.microsoft.com	ODBC Desktop Database Drivers Microsoft SQL Server Driver DB2/Integrator	SQL Server Excel Text dBASE Paradox Access FoxPro Btrieve DB2
MiniSoft, Inc. Telephone: 800-682-0200 Telephone: 360-568-6602 Fax: 360-568-2923 http://www.minisoft.com	MiniSoft ODBC/32	HP Image/SQL
Monette Information Systems Telephone: 1-800-MONETTE Fax: 757-357-5163 http://www.monette.org	Synergex ODBC Driver	Monette application data
NCR Corporation Telephone: 513-445-5000 http://www.ncr.com	Teradata ODBC Driver	Teradata
Neon Systems Telephone: 218-491-4200 Telephone: 800-505-6366 Fax: 281-242-3880 http://www.neonsys.com	ShadowDirect Enterprise Direct (server)	DB2 IMS VSAM Oracle Sybase ADABAS
NobleNet, Inc. (Rogue Wave) Telephone: 508-460-8222 Fax: 508-460-3456 http://www.noblenet.com	NobleNet One Driver SDK (server)	Data sources with ODBC drivers
Nogginware Corporation http://www.nogginware.com	RemoteDB Gateway (server)	Data sources with ODBC drivers
NTT Data Corporation Telephone: 03-3647-8611 Fax: 03-3647-7511 http://unisql.www.nttdata.co.jp	Inforover ODBC Driver	Inforover UniSQL
Oberon microsystems, Inc. Telephone: ++41-1-445-1751 Fax: ++41-1-445-1752 http://www.oberon.ch	ODBC Driver for Sql Subsystem	Oberon/F Sql Subsystem (Black Box Component Builder)
Object Design, Inc. Telephone: 617-674-5000 Fax: 617-674-5010 http://www.odi.com	ObjectStore ODBC Driver Open Access (server)	ObjectStore

Company	ODBC Driver	Data Source
Objectivity, Inc. Telephone: 650-254-7100 Fax: 650-254-7171 http://www.objectivity.com	Objectivity ODBC Driver	Objectivity/DB
Ocelot Computer Services Telephone: 780-472-6838 Email: 71022.733@compuserve.com	Ocelot ODBC Driver	Ocelot SQL-92
Open Horizon Inc. (See IBM)	Connection ODBC Client	Connection Server
OpenLink Software Inc. Telephone: 781-273-0900 Fax: 781-229-8030 http://www.openlinksw.com	OpenLink Universal Data Access Driver Suite (client or server-side)	Informix Ingres Kubl Oracle Microsoft SQL Server Postgres Progress SOLID Sybase Velocis
Operating System Support Telephone: 561- 241-9900 Telephone: 800-333-5899 Fax 561-241-0003 http://www.ossfl.com	ViaODBC-32	Advanced Plus UltPlus
Oracle Corporation 415-506-7000 http://www.oracle.com	Oracle ODBC Driver Rdb ODBC Driver Oracle Open Gateways	Oracle ADABAS Access Btrieve IDMS Datacom DB2 DMS II FOCUS Image/SQL IMS Infoman Informix Ingres ISAM M Model 204 QSAM Rdb RDMS RMS SAP SESAM SQL Server Supra Sybase System 2000 Teradata TOTAL UDS Jukebox VSAM

Company	ODBC Driver	Data Source
PARKWAY Software GmbH Telephone: +49 089 6518-034 Fax: +49 089 6518-161 http://www.parkway-software.com	ConnectWare	Btrieve C-ISAM CA-Realia D-ISAM Micro Focus files mbp VSAM
Persistent Systems Private Limited Fax Telephone: +91 20 37 6701 http://www.pspl.co.in/PSEnList/	PS EnList	LDAP servers
Pervasive Software Telephone: 800-287-4383 Telephone: 512-794-1719 http://www.pervasive.com	Btrieve Pervasive SQL ODBC driver	Pervasive SQL
Phoenix Systems, Inc. Telephone: 404-633-2466 Fax: 404-634-9975 http://www.phoenix-systems-inc.com	FUNDS ODBC Interface	FUNDS System Databases
Pioneer Systems, Inc. Telephone: 513-247-1500 Fax: 513-247-1400 http://www.pioneersys.com	INFOAccess ODBC driver	Unisys A series and NX systems: DMS II and keyed files. Unisys 2200 and IX: DMS 1100 and DMS2200
Platinum Technology Inc. (see Computer Associates) Telephone: 800-442-6861 Telephone: 708-620-5000 Fax: 708-691-0417 http://www.platinum.com	InfoHub	IDMS IMS ADABAS DB2 VSAM Sequential Files
Poet Software Corp. Telephone: 800-950-8845 Telephone: 415-286-4640 Fax: 415-286-4630 http://www.poet.com	Poet ODBC Driver	Poet ODBMS
Professional Data Associates, Inc. Telephone: 718-263-1334 Fax: 718-263-1350 http://www.pdaco.com/tbred.htm	TS ODBC Data Server	Thoroughbred files
Progress Software Telephone: 781-280-4000 Fax: 781-280-4895 http://www.progress.com	Apptivity Server	Progress Oracle Microsoft Access SQL Server Informix Sybase other ODBC or JDBC data sources
R:BASE Technologies, Inc. Telephone: 724-733-0053 Fax: 724-733-0196 http://www.rbasetechnologies.com	Ottero ODBC	Ottero Engine
QBS Software Ltd. Telephone: +440 181 956 8001 Fax: +440 181 956 8010 http://www.qbss.com	Advantage ODBC Driver	Advantage Database Server
Quadbase Systems Inc. Telephone: 408-982-0835 Fax: 408-982-0838 http://www.quadbase.com	Quadbase-SQL ODBC Driver	Quadbase-SQL
Raima (see Centura Software)	Velocis ODBC Driver	Raima Data Manager++

Company	ODBC Driver	Data Source
Telephone: 800-327-2462 Telephone: 206-515-9477 Fax: 206-748-5200 http://www.raima.com	RDM ODBC Gateway	Velocis Database Server
Real Time eXecutive, Inc. Telephone: 508-384-7717 Fax: 504-384-9074 http://www.rtx.com	ODBC for RTXHDB	RTXHDB
Recital Corporation http://www.recital.com	Recital ODBC Developer	Recital
Red Brick Systems (see Informix)	Red Brick ODBC Driver	Red Brick Warehouse VPT
Red Point Software Telephone: 214-355-5200 Fax: 214-355-5201 http://www.redpt.com	SnmpQL ODBC Driver	Data from SNMP devices
Santa Cruz Operation Telephone: +44(0) 113 251 2222 Fax: +44(0) 113 251 2223 http://www.sco.com	SCO SQL Retriever	Informix Oracle OpenIngres InterBase Sybase Progress
SAS Institute Telephone: 919-677-8000 Fax: 919-677-8123 http://www.sas.com	SAS/Access Interface to ODBC	SAS
ShowCase Corp. Telephone: 800-829-3555 Telephone: 507-288-5922 Fax: 507-287-2803 http://www.showcasecorp.com	ShowCase ODBC Driver	IBM AS/400
Siemens Nixdorf Informationssysteme http://www.sni.de	DBA.D DBA.X DBA.2000	Informix Oracle OpenIngres SESAM/SQL UDS/SQL
Simba Technologies Inc. Telephone: 800-388-4933 Telephone: 604-601-5300 Fax: 604-601-5320 http://www.simbatech.com	Simba Express (server) Simba ODBC Driver SDKs ODBC drivers	Oracle Sybase DB2 Informix SQL Server
Software AG Telephone: 800-423-2227 Telephone: 703-860-5050 http://www.adabas.com	ADABAS ODBC Driver	ADABAS D ADABAS
Software Clearing House Telephone: 513-579-0455 Fax: 513-579-1064 http://www.sch.com	Open/A ODBC Driver	Open/A A-Series
Software Migration and Conversion Telephone: 612-452-9270 Fax: 612-688-2191	Open CQL Driver	DMS-1100 PCIOS RDMS-1100
SoftOption Telephone: +44(0)1322 278603 Fax: +44(0)1322 289630	ODBC for CTOS	CTOS ISAM

Company	ODBC Driver	Data Source
Solid Information Technology, Ltd. Fax: +358-9-4774 7390 http://www.solidtech.com	SOLID ODBC Driver	SOLID Server
SolutionsIQ Telephone: 888-882-6669 Telephone: 425-519-6613 Fax: 425-453-8871 http://www.solutionsiq.com	CONNX	Oracle DataFlex Rdb RMS
SQLData Systems, Inc. http://www.sqldata.com	SQLData Enterprise Server (server)	ODBC data sources
StarQuest Software Telephone: 510-704-2000 Fax: 510-704-2001 http://starweb.starware.com	StarSQL ODBC Driver	DRDA DB2 SQL/DS
Superbase Developers plc 310-374-4125 +44 1223 365550 Fax +44 1223 363302 http://www.superbase.com	Superbase ODBC Driver	Superbase
Sybase, Inc. Telephone: 800-879-2273 Telephone: 800-221-3634 Telephone: 510-922-3500 Fax: 510-922-4850 http://www.sybase.com	Adaptive Server Adaptive Server Anywhere ODS ODBC-ODS EnterpriseCONNECT Gateway	Adaptive Server Anywhere Adaptive Server DB2 SQL/DS Teradata SQL/400 VSAM IMS IDMS ADABAS Oracle SQL Server DB2/2
Sysdeco Mimer AB Telephone: +46 18-185000 Fax: +46 18-185100 http://www.mimer.se	MIMER ODBC Driver	MIMER SQL RDBMS
SYWARE, Inc. Telephone: 617-497-1376 Fax: 617-697-8729 http://www.syware.com	Dr. DeeBee ODBC Driver Kit	Xbase ISAM
Synergex International Corporation Telephone: 800-366-3472 Telephone: 916-635-7300 Fax: 916-635-6549 http://www.synergex.com	Synergy ODBC Driver	Synergy databases
Tandem Computers Telephone: 800-482-6336 Telephone: 408-285-5446 Fax: 408-285-6010 http://www.tandem.com	NonStop ODBC Server	Tandem NonStop SQL
TimesTen Software Telephone: 800-970-1248 Telephone: 650-526-5100 Fax: 650-526-5199 http://www.timesten.com	TimesTen ODBC	TimesTen Server
TopSpeed Corporation Telephone: 800-354-5444 Telephone: 954-785-4555 http://www.topspeed.com/tsodbc.htm	TopSpeed ODBC Interface	Clarion TopSpeed databases

Company	ODBC Driver	Data Source
Transoft, Ltd. Telephone: 770-933-1965 Fax: 770-933-3464 http://www.transoft.com	Transoft U/SQL Client-Server (server)	C-ISAM Micro Focus COBOL EXTFH AcuCOBOL Oracle Informix Sybase
Trifox, Inc. http://www.trifox.com	VORTEXodbc	ADABAS D DB2 Ingres Informix GENESIS Oracle Rdb Microsoft SQL Server Sybase
Trilogy Technology International Telephone: 818-854-6288 Fax: 818-854-6289 http://www.openpath.com	OpenPath RDA/ODBC	Informix Oracle Sybase IBM DB2 RDA
UniSQL, Inc. (for Japan, see NTT Data Corp. Otherwise, see Cincom.)	UniSQL/X UniSQL/M	UniSQL
Unisys Telephone: 800-874-8647 Telephone: 800-448-1424 Telephone: 714-380-6460 http://www.unisys.com	TransIT ODBC HDBC Component	HMP NX A_Series: DMS II LINC SQLDB KEYED1011 sequential files
Usoft Telephone: +31 (0) 35 6990699 Fax: +31 (0) 35 6950124 http://www.usoft.com	USoft Open Rules API	DB2 Informix Oracle SQL Server Solid Sybase (via USoft Open Rules Engine)
Vertisoft Telephone: 905-474-1862 Telephone: 800-361-0099 Fax: 905-474-0006	C-ISAM ODBC Driver	C-ISAM
Versant Object Technology Telephone: 510-789-1500 Fax: 510-789-1515 http://www.versant.com	Versant/ODBC	Versant
Viaserv, Inc. Telephone: 800-348-3964 http://www.viaserv.com	ViaSQL for VSE-VSAM ViaSQL for VSE-SQL/DS	Viaserv Gateway SQL/DS
Wall Data Inc. Telephone: 206-814-9255 Fax: 206-861-3175 http://www.walldata.com	Rumba Arpeggio	DB2 IBM DB/VM IBM AS/400 IMS
White Cross Systems, Inc. Telephone: 310-577-8188 Fax: 310-577-8192 Telephone: 44 1344 300 770 Fax: 44 1344 301 424 http://www.whitecross.com	White Cross 9000	White Cross RDBMS

Company	ODBC Driver	Data Source
XDB Systems, Inc. (see MERANT) Telephone: 410-312-9300 Fax: 410-312-9500	ExpressLane (server)	XDB IBM DB2 IBM AS/400
YARD Software Gmbh Telephone: +(49) 221/98664-0 Fax: +(49) 221/98664-99 http://www.yard.de	ODBC Driver for YARD-SQL	YARD-SQL

UnixODBC ODBC Drivers

The following table is a list of unixODBC drivers and the companies that produce them. Most of these are freely downloadable unixODBC drivers. There is also a project underway to create a unixODBC driver for Oracle.

Because some of the drivers below are freeware products, they may not be production-worthy or free of bugs. You should contact the driver author if you have problems installing or using a specific driver. For data sources marked with a plus sign (+), we successfully installed and tested the driver with unixODBC. For those marked with a minus (-), we were unable to succeed in making the driver work with unixODBC. The drivers for data sources not marked by a plus or minus were not tested.

Company	unixODBC Driver	Data Source
unixODBC http://www.unixodbc.org	PostgreSQL Driver included with unixODBC	PostgreSQL (+)
T.C.X DataKonsult AB Fax: +46-8-7296905 http://www.mysql.com/download_myodbc.html	MyODBC Driver for unixODBC	MySQL, version 3.23 and higher (+)
IBM Corp. Telephone: 800-IBM-4YOU http://www.software.ibm.com/data/db2	The libdb2.so library, part of DB2 for Linux, can serve as an ODBC driver.	DB2 for Linux
YARD Software GmbH Telephone: +49 221 98664-0 Fax: +49 221 98664-99 http://www.yard.de	YARD unixODBC Driver	YARD-SQL
Ke Jin's Net News ODBC Driver	Internet News Server Driver included with unixODBC	Internet News Server
Easysoft Telephone: +44 (0) 113 222 0400 Fax: +44 (0) 113 222 0500 http://www.easysoft.com/	Easysoft's ODBC-ODBC Bridge	ODBC-ODBC
unixODBC http://www.unixodbc.org	MiniSQL Driver included with unixODBC	MiniSQL (-)
unixODBC http://www.unixodbc.org	SQL unixODBC Driver for Text Files	Text Files (-)

Appendix H

Using LinkMaker™ Embedded SQL in CobolScript® Professional

CobolScript Professional Edition with LinkMaker™ allows for the use of embedded SQL in your Linux®, Microsoft Windows®, SunOS®, and FreeBSD® programs. Embedded SQL is a term used to describe the use of Structure Query Language from within a programming language. In CobolScript Professional, LinkMaker™ embedded SQL statements are preceded by the keywords EXEC SQL and end with END-EXEC. Any valid SQL statements may be placed between these tokens, as shown below.

The general format is:

```
EXEC SQL
   <SQL statement>
END-EXEC.
```

Before you can execute SQL statements in your CobolScript programs, you must successfully establish an ODBC connection to your database by setting up and configuring an ODBC data source name. If you are working in a Unix environment, you must also install unixODBC and switch to the LinkMaker™-enabled version of CobolScript. Refer to Appendix G for platform-specific instructions on how to install unixODBC and configure ODBC data sources.

The first step in using embedded SQL is to programmatically connect to an ODBC data source. This is done with the OPENDB command:

```
OPENDB USING <data-source-name> <user-id> <password> <return-code>.
```

After you have finished executing SQL statements, you should close the connection to the data source with the CLOSEDB command:

```
CLOSEDB USING <return-code>.
```

Refer to the OPENDB and CLOSEDB entries in Appendix A, *Command Reference*, for more information on these two commands.

An SQL communications area is required when working with an ODBC data source. In CobolScript, this area of memory is allocated by defining the variable *sql-return-codes*. You should include this definition in your programs, or include the sample copybook SQL.CPY, which contains this variable definition:

```
1 sql-return-codes.
  5 sqlstate        PIC X(5).
  5 sqlnativeerror  PIC S9(6).
  5 sqlerrormessage PIC X(500).
  5 sqlstatement    PIC X(500).
```

Below is an example of how to connect to a data source, execute an SQL statement, and close the connection. *MySQLTest* is an ODBC data source name that is configured to use a MySQL database. The parameters `testuser` and `testpass` are used to identify the UserName and Password of a MySQL user with SELECT permission on the table named *customer*:

```
MOVE `MySQLTest`  TO data_source_name.
MOVE `testuser`   TO user_id.
MOVE `testpass`   TO password.
OPENDB USING  data_source_name
              user_id
              password
              return_code.
EXEC SQL
   SELECT firstname, lastname, description, balance
   INTO :customer_first_name
        ,:customer_last_name
        ,:customer_description
        ,:formatted_balance
   FROM  customer
   WHERE   lastname = 'Doe'
       AND firstname = 'Charles'
END-EXEC.

DISPLAY `firstname: ` & customer_first_name
DISPLAY `lastname: ` & customer_last_name
DISPLAY `description: ` & customer_description
DISPLAY `balance: ` & formatted_balance

CLOSEDB USING return_code.
```

In addition to standard SQL and DDL statements, CobolScript allows for the use of *cursors*. A cursor is a result set that is declared, opened, traversed with the fetch command, and then closed. Here's an example:

```
EXEC SQL
   DECLARE cust_cursor cursor FOR
     SELECT firstname, lastname, balance
     FROM customer
     ORDER BY balance
END-EXEC.

EXEC SQL
```

```
   OPEN cust_cursor
END-EXEC.

DISPLAY LINEFEED.
DISPLAY `firstname            lastname                balance`.
DISPLAY `-----------------------------------------------------`.

PERFORM UNTIL sqlnativeerror NOT = 0 OR sqlstate NOT = `00000`
   EXEC SQL
      FETCH NEXT cust_cursor
      INTO :customer_first_name,:customer_last_name, :formatted_balance
   END-EXEC

   IF sqlnativeerror = 0 AND sqlstate = `00000`
       DISPLAY customer_first_name & SPACE
             & customer_last_name  & SPACE
             & formatted_balance
   END-IF
END-PERFORM.

DISPLAY `-----------------------------------------------------`.
DISPLAY LINEFEED.

EXEC SQL
   CLOSE cust_cursor
END-EXEC.
```

Note the steps in the life of the cursor. The cursor is first declared, then opened (the open actually executes the SELECT statement). Next, fetches are performed inside a loop until an error is encountered or the end of the cursor is reached (the end-of-cursor state is normally signified by a sqlstate value of `S1010`). Finally, the cursor is closed.

Embedded SQL Quick Reference

The following reference provides basic syntax descriptions and examples of embedded SQL statements. Refer to your database or database driver's documentation for a more detailed explanation of these statements, and to determine the full range of SQL statements that are available to you for your specific data source.

ALTER TABLE

Command:	**ALTER TABLE**
Syntax:	ALTER TABLE <table-name> ADD <column1> <sql-data-definition>
Description:	Changes some component of a database table.

Example Usage:	`EXEC SQL`
	`  alter table customer add`
	`  balance decimal(6,2)`
	`END-EXEC.`

CLOSE

Command:	**CLOSE**
Syntax:	CLOSE <cursor-name>
Description:	Closes a cursor defined by a DECLARE command.

Example Usage:	`EXEC SQL`
	`  close cust_cursor`
	`END-EXEC.`

COMMIT

Command:	**COMMIT**
Syntax:	COMMIT
Description:	Commits the most recent changes to a database.

Example Usage:	`EXEC SQL`
	`  commit`
	`END-EXEC.`

CREATE INDEX

Command:	**CREATE INDEX**
Syntax:	CREATE INDEX <index-name> ON <table-name> (<column1>,…<columnX>)
Description:	Creates an index for a database table.

Example Usage:	`EXEC SQL`
	`  create index cust_index on customer (firstname)`
	`END-EXEC.`

CREATE TABLE

Command:	**CREATE TABLE**

Syntax:	CREATE TABLE <table-name> (<column1> <sql-data-definition>, <column2> <sql-data-definition>, : <columnX> <sql-data-definition>),
Description:	Creates a table inside a database.

Example Usage:	`EXEC SQL`
	`  create table customer`
	`  ( firstname  varchar(20),`
	`  lastname  varchar(20),`
	`  description varchar(50))`
	`END-EXEC.`

DECLARE

Command:	**DECLARE**
Syntax:	DECLARE <cursor-name> CURSOR FOR SELECT <column1>, <column2>, : <columnX> FROM <table-name> WHERE <condition>
Description:	Defines a result set to be traversed with the FETCH command.

Example Usage:	`EXEC SQL`
	`  declare cust_cursor cursor for`
	`  select firstname, dollar_amount`
	`  from customer`
	`  order by firstname`
	`END-EXEC.`

DELETE

Command:	**DELETE**
Syntax:	DELETE FROM <table-name> <condition>
Description:	Deletes a row from a database table.

Example Usage:	`EXEC SQL`
	`  delete from customer`
	`  where firstname = 'dean6'`
	`END-EXEC.`

DROP INDEX

Command:	**DROP INDEX**
Syntax:	DROP INDEX <index-name> ON <table-name>
Description:	Removes an index for a table from the database.

Example Usage:	`EXEC SQL`
	`  drop index cust_index on customer`
	`END-EXEC.`

DROP TABLE

Command:	**DROP TABLE**
Syntax:	DROP TABLE <table-name>

Description:	Removes a table from a database.

Example Usage:	`EXEC SQL`
	`  drop table customer`
	`END-EXEC.`

FETCH

Command:	**FETCH**
Syntax:	FETCH [{NEXT | PRIOR | FIRST | LAST | ABSOLUTE {<int-constant> | <cobolscript-host-variable> } | RELATIVE {<int-constant> | <cobolscript-host-variable> }}] <cursor-name> INTO host-variable [,...]
Description:	Retrieves data from a single row in a result set defined by a DECLARE command. In most databases, the end-of-cursor state is signified by a sqlstate value of `S1010`. Check for this sqlstate value (or, for sqlstate NOT = `00000`) to gracefully terminate a FETCH loop.

Example Usage:	`EXEC SQL`
	`  fetch relative :row-position  cust_cursor`
	`  into :customer-first-name, :customer-dollar-amount`
	`END-EXEC.`

INSERT

Command:	**INSERT**
Syntax:	INSERT INTO <table-name> VALUES (<literal | cobolscript-host-variable>, : <literal | cobolscript-host-variable>)
Description:	Insert data from host cobolscript variables or literals into a database table.

Example Usage:	`EXEC SQL`
	`  insert into customer`
	`  values (:customer-first-name,`
	`  :customer-last-name,`
	`  :customer-description)`
	`END-EXEC.`

OPEN

Command:	**OPEN**
Syntax:	OPEN <cursor-name>
Description:	Opens a cursor defined by a DECLARE command.

Example Usage:	`EXEC SQL`
	`  open cust_cursor`
	`END-EXEC.`

ROLLBACK

Command:	**ROLLBACK**
Syntax:	ROLLBACK
Description:	Rollback or undo the most recent changes to a database.
Example Usage:	`EXEC SQL`

	rollback END-EXEC.

SELECT

Command:	**SELECT**
Syntax:	SELECT <column1>, <column2>, : <columnX> INTO <:cobolscript-host-variable1>, <:cobolscript-host-variable2>, : <:cobolscript-host-variableX> FROM <table-name> WHERE <condition>
Description:	Retrieves data from a table and places it in host cobolscript variables.
Example Usage:	EXEC SQL select firstname, lastname, description into :customer-first-name, :customer-last-name , :customer-description from customer where firstname = 'dean8 ' END-EXEC.

UPDATE

Command:	**UPDATE**
Syntax:	UPDATE <table-name> SET <column1> = <literal | :cobolscript-host-variable> : <condition>
Description:	Updates a column or columns in a table.
Example Usage:	EXEC SQL update customer set description = 'update test again' where firstname = :customer-first-name and lastname = :customer-last-name END-EXEC.

Appendix

CobolScript® Error Messages

The messages described in this appendix are produced when the CobolScript engine encounters an error while executing a CobolScript program. An error can be indicative of incorrect syntax, program inconsistencies, missing files, or improper system setup. The error messages listed below are in numerical order; where sequential error numbers have the same error message and description, the error number is provided as a range, and the errors are described in a single entry.

Error Number:	**0000**
Error Message:	"Usage: cobolscript.exe –b <filename> "
Description:	In order to build an executable from the command prompt using CobolScript AppMaker, you must specify a target filename to build.

Error Number:	**0001**
Error Message:	"Error opening file. File name may not exist: "
Description:	The file was not found. Attempts were made to open the file as named, and with extensions of .cbl, .cob, .CBL, .COB, but all of them failed. Check the spelling of the filename and verify that the file exists on your machine, in the specified path. If you are working on a Unix system, check the case of the filename as well.

Error Number:	**0002**
Error Message:	"Program file is empty: "
Description:	The specified program file exists but does not contain any CobolScript code.

Error Number:	**0003**
Error Message:	"Unrecognized CobolScript syntax."
Description:	An error has occurred that has stopped the execution of your program. This error is either a runtime error or a syntax error. Check the syntax of the line that caused the error, as well as the line immediately prior to it.

Error Number:	0004
Error Message:	"Stack overflow."
Description:	The maximum permitted CobolScript stack size has been exceeded. With this version of CobolScript, use fewer modules and avoid recursive calls where possible.

Error Number:	0005
Error Message:	"Line limit bypassed. This version of CobolScript has a line limit of: "
Description:	Your program has exceeded the maximum number of lines of code permitted per program for this version of CobolScript. The maximum is 32,767 lines.

Error Number:	0006
Error Message:	"Line limit bypassed. This version of CobolScript has a line limit of: "
Description:	*Interactive mode error* – Your program has exceeded the maximum number of lines of code permitted per program for this version of CobolScript. The maximum is 32,767 lines.

Error Number:	0007
Error Message:	"Nothing to run Use "load <filename>" to load a program. "
Description:	*Interactive mode error* – You are attempting to run a program in interactive mode, but no code has been loaded into the program buffer. Use the load command to load a CobolScript program before you try to run it.

Error Number:	0008
Error Message:	"Cannot display variable: "
Description:	*Interactive mode error* – You are trying to display a variable that has not been defined. Either it has not been created yet, or you have misspelled the variable name. In interactive mode, you must run a program before you can display its variables.

Error Number:	0009
Error Message:	"No program in buffer. Use load <filename> to load a program."
Description:	*Interactive mode error* – You are trying to list program code in interactive mode, but you have not loaded it into the program buffer yet. You must load a text file containing valid CobolScript code into the program buffer before you can list it.

Error Number:	0010, 0011
Error Message:	"No variables defined in program in buffer."
Description:	*Interactive mode error* – No variables have been defined yet. You must run a program before you can display its variables or its variable positions in interactive mode. The variables are defined as the CobolScript program is interpreted, so you must at least step through your variable definitions before you can display the variables.

Error Number:	0012
Error Message:	"No files defined in program in buffer."
Description:	*Interactive mode error* – No files are in memory yet. Because CobolScript is an interpreter, files will not be displayed until the appropriate FD statement is interpreted. You must run or step through a program before you can display its files.

Error Number:	0013
Error Message:	"No modules defined in program in buffer."
Description:	*Interactive mode error* – No modules have been defined yet. You must load a CobolScript program before you can display its modules. The code must have at least one module in order for it to be a valid program (modules are any paragraphs defined in your program).

Error Number:	0014-0017
Error Message:	"Error dumping variables to { dump.var | dump.mod | dump.lst | dump.pos }."
Description:	*Interactive mode error* – Make sure there is sufficient disk space to create the named dump file. Also, verify that the appropriate permissions are set to allow writing to the working directory.

Error Number:	0018
Error Message:	"Following file not found: "
Description:	*Interactive mode error* – The file was not found. Attempts were made to open the file as named, and with extensions of .cbl, .cob, .CBL, .COB, but all of them failed. Check the spelling of the filename and verify that the file exists on your machine, in the specified path.

Error Number:	0019
Error Message:	"Error saving to file: "
Description:	*Interactive mode error* – Cannot save file because the filename you have specified cannot be opened for writing. Check the permissions on the target directory and the available disk space on this machine, and make certain the file is not being accessed by another application.

Error Number:	0020
Error Message:	"Error writing to file: "
Description:	*Interactive mode error* – Cannot save to this file. The filename you have specified has been opened correctly, but an error has occurred while writing to that file. Check available disk space on this machine and permissions on the target directory, and make certain the file is not being accessed by another application.

Error Number:	0021-0025
Error Message:	"This line of code is too long: "
Description:	A single line of code in your CobolScript program has exceeded the maximum permissible length. Check for an unbalanced string (a missing ` symbol on either side of a string literal), a missing period in this or a prior sentence, or simply a line of code that is too large.

Error Number:	0026-0027
Error Message:	"Error loading copybook: "
Description:	A copybook could not be loaded. Make sure that the filename exists on the current machine, in the specified path. If you are working on a Unix system, check the case of the filename as well. Also, refer to any more specific errors issued in conjunction with this error message.

Error Number:	0028
Error Message:	"Program sentence is missing a terminating period. "

Description:	A program sentence is not correctly terminated with a period. A program sentence is a complete statement that is not embedded inside conditional or loop logic. All program sentences must end with a period.

Error Number:	**0029**
Error Message:	"Program sentence is missing a terminating period, END-IF, or END-PERFORM."
Description:	Either an IF statement is missing an END-IF, a PERFORM statement is missing an END-PERFORM, or a program sentence is missing a terminating period. Check the IF and PERFORM statements in your code for their ending keywords, and check for missing periods in the line causing the error and previous line(s).

Error Number:	**0030**
Error Message:	"Program line should not have a terminating period."
Description:	A code statement has a terminating period at the end of the line when it should not. Remove the period from the end of the line. Terminating periods are not permitted on lines that are embedded inside conditional or loop logic.

Error Number:	**0031-0036**
Error Message:	"Check for missing period. { *Specific problem explanation* }."
Description:	The variable definition has incorrect syntax. Check for a missing period on this or the previous line, or see Chapter 3 for information on how to define variables.

Error Number:	**0037, 0038**
Error Message:	"String in line is not properly terminated."
Description:	A string in a line is not properly terminated with an ending string delimiter. Literal strings cannot extend across multiple lines.

Error Number:	**0039**
Error Message:	"This version of CobolScript does not permit more than two OCCURS clauses in a single group-level data item."
Description:	This version of CobolScript permits no more than two OCCURS clauses in one group-level data item. You must restructure your variable definitions so that this limit is not bypassed.

Error Number:	**0040**
Error Message:	"Nonnumeric variable not permitted in TIMES clause: "
Description:	The TIMES clause of OCCURS variables is only permitted to have a literal number or numeric variable index, but the existing index is non-numeric. Correct this by using a numeric literal or numeric variable qualifier instead.

Error Number:	**0041, 0042**
Error Message:	"Multilevel OCCURS clauses not permitted in CobolScript Standard Edition. Upgrade to Professional Edition for multilevel OCCURS support."
Description:	You have CobolScript Standard Edition, which does not permit multilevel OCCURS clauses (multidimensional arrays). Upgrade to CobolScript Professional Edition at https://www.cobolscript.com/cgi-bin/cobolscript.exe?catalog.cbl for multilevel OCCURS clause support.
Error Number:	**0043**
Error Message:	"Variable index argument to TIMES clause not found. Make certain variable is defined prior to this OCCURS definition."

Description:	The TIMES clause index variable is not defined. Define this variable, as a numeric, prior to defining the OCCURS variable that is causing this error.

Error Number:	**0044**
Error Message:	"Stack error in variable stack. Check your variable definition syntax."
Description:	CobolScript encountered an error while manipulating its internal variable stack. Correct any errors in your variable definition syntax to fix the problem; make certain that all variable definitions begin on a new line, with a positive integer value to begin the definition.

Error Number:	**0045**
Error Message:	"Stack overflow in variable stack. Reduce amount of nesting in gldi variables."
Description:	CobolScript ran out of variable stack space while pushing gldi nesting levels to its internal variable stack. Reduce the amount of nesting in your group-level data item variables.

Error Number:	**0046**
Error Message:	"The first variable defined in a program must be a top-level variable, with an outline level of 1."
Description:	The first variable defined in a program must have an outline level of 1. Modify the outline level of your first variable definition to remedy this.

Error Number:	**0047**
Error Message:	"Picture clause exceeds maximum token length permitted."
Description:	A picture clause must be no more than 80 characters in length. Change your picture clause to use shortened picture clause notation, such as PIC X(200), instead of PIC XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.

Error Number:	**0048**
Error Message:	"Invalid variable definition syntax."
Description:	Check the syntax of your variable definition for any extra tokens or incorrect keyword usage.

Error Number:	**0049-0051**
Error Message:	"There was an error while setting the VALUE of this variable: "
Description:	There was a problem setting the VALUE of a variable in its definition. This may be because the picture clause of the variable and the value clause are incompatible. Alternatively, there may be an issue with the size of the VALUE you are setting.

Error Number:	**0052**
Error Message:	"To use PIC X(n), you must either specify a VALUE clause or use the implied PIC X(n) format."
Description:	The use of PIC X(n) requires that you specify a VALUE clause, or alternatively use the implied PIC X(n) format. Specify a value clause, or change the picture clause to the implied format by omitting everything from the variable definition except the variable outline level number and the literal value enclosed in string delimiters.

Error Number:	**0053-0055**

Error Message:	"Missing string delimiters or misspelled keyword in VALUE clause of variable definition."
Description:	This error is a result of specifying a VALUE clause in a variable definition that is not a literal keyword such as SPACE, SPACES, LINEFEED, TAB, etc., not a numeric literal, and not an alphanumeric literal enclosed in string delimiters (`, the accent symbol, by default). You may not use variables in VALUE clauses; the value must either be a keyword or literal.

Error Number:	**0056-0059**
Error Message:	"Nonnumeric value in numeric variable's VALUE clause: "
Description:	Only numeric literals (i.e., numbers that are not enclosed by delimiters) are permitted in the VALUE clause of a numeric variable. Change the value clause to a literal number, not enclosed by delimiters.

Error Number:	**0060, 0061**
Error Message:	"VALUE keyword specified in variable definition but no VALUE clause exists."
Description:	Insert a value clause (i.e., an alphanumeric or numeric literal) as appropriate after the VALUE keyword.

Error Number:	**0062, 0063**
Error Message:	"Invalid variable definition syntax."
Description:	Some portion of this variable's picture clause definition is incorrect. Check your syntax, paying attention to the picture clause definition.

Error Number:	**0064**
Error Message:	"Invalid alphanumeric picture clause: "
Description:	Some portion of this picture clause definition is incorrect. Correct any syntax errors.

Error Number:	**0065**
Error Message:	"Variable names must begin with a letter."
Description:	You cannot use numeric or special characters as the first character of a variable name. Modify your variable names to begin with letters only.

Error Number:	**0066**
Error Message:	"Maximum variable name length exceeded: "
Description:	The maximum length of a variable name has been exceeded. The maximum length is 80 characters. Reduce the number of characters in your variable name.

Error Number:	**0067-0104**
Error Message:	"Variable not defined: "
Description:	The named variable is not defined. Check the variable name against the variables you have defined in your program. If you are using an array element like variable-name(counter), make sure that counter is not zero, and is in the allowable range for that particular OCCURS clause.

Error Number:	**0105**
Error Message:	"A CGI variable in the submitting form does not have a matching CobolScript variable."

Description:	A CGI variable in the submitting form does not have a matching CobolScript variable. Create a matching variable in your program with the same name as the field name in the submitting form.

Error Number:	**0106**
Error Message:	"Invalid syntax in MOVE statement."
Description:	The syntax in the MOVE statement is incorrect. A MOVE statement must have at least 4 tokens, as in "MOVE <source> TO <target>". Verify the statement syntax, and verify that there are no unbalanced string terminators in the preceding line of code.

Error Number:	**0107**
Error Message:	"Cannot execute MOVE statement."
Description:	There was a problem executing this MOVE statement. Make certain that the source variable is a valid literal or variable, and that the target variable has been defined. It is possible that the MOVE failed because the picture clauses of the source and target variables were incompatible. If either the source or the target contains an array definition or a positional variable definition, make sure that it is properly defined prior to this statement.

Error Number:	**0108**
Error Message:	"Functions are not allowed as stand-alone arguments in MOVE statements."
Description:	Functions not allowed as stand-alone arguments in MOVE statements. If you wish to use functions or expressions as arguments in a variable assignment use the COMPUTE statement instead of MOVE.

Error Number:	**0109**
Error Message:	"Unrecognized or illegal source variable type."
Description:	The source variable of an illegal or unrecognized type for this MOVE statement. See the Language Reference for information on what source – target data type combinations are permissible in the MOVE statement.

Error Number:	**0110**
Error Message:	"Source or target argument in MOVE statement is too long."
Description:	The source or target argument in the MOVE statement is too long. It must be less than 180 characters.

Error Number:	**0111**
Error Message:	"Invalid syntax in INITIALIZE statement."
Description:	The INITIALIZE statement requires a minimum of 2 tokens to work properly; the first should be the keyword INITIALIZE, and the second should be a valid variable name.

Error Number:	**0112**
Error Message:	"INITIALIZE target variable not defined: "
Description:	The target variable name in the INITIALIZE statement was not properly defined. Verify that this variable is properly defined prior to this statement.

Error Number:	0113
Error Message:	"Invalid syntax in SET statement."
Description:	The SET command requires at least 4 tokens to work properly. See Language Reference for proper syntax of SET.

Error Number:	0114
Error Message:	"SET target variable not defined: "
Description:	The target variable name in the SET command could not be found in the variables in memory. Verify the name of this variable, and make certain it is defined prior to this statement.

Error Number:	0115
Error Message:	"Cannot execute SET statement."
Description:	There was a problem executing this SET statement. Make certain that the source variable is a valid literal or variable, and that the target variable has been defined. It is possible that the SET failed because the picture clauses of the source and target variables were incompatible. If either the source or the target contains an array definition or a positional variable definition, make sure that it is properly defined prior to this statement.

Error Number:	0116
Error Message:	"You have exceed the maximum allowable record length of 10k."
Description:	File records must be 10,000 bytes or less. You must split your file into multiple files if the current record size exceeds this limit.

Error Number:	0117
Error Message:	"Cannot close the following file: "
Description:	The file you are attempting to CLOSE has not been described in this program. Check the name of your file against the corresponding FD statement in your program.

Error Number:	0118
Error Message:	"Invalid syntax in CLOSE statement."
Description:	The CLOSE statement requires at least 2 tokens. See the Language Reference for proper syntax of the CLOSE statement.

Error Number:	0119
Error Message:	"Invalid syntax in OPEN statement."
Description:	At least 4 tokens are required for a properly defined OPEN statement. See the Language Reference for proper syntax of the OPEN statement.

Error Number:	0120
Error Message:	"The file you are attempting to open has not been described in an FD statement: "
Description:	You must describe all files used by a program in that program's FD (File Description); check for an appropriate FD entry for the named file, and create it if it does not exist. The FD describes the filename and its record size to CobolScript which must be done before any I/O operations can take place on the file.

Error Number:	0121-0124
Error Message:	"Possible directory/file permission problem. This file could not be properly opened for { READING | UPDATING | WRITING | APPENDING } : "
Description:	The specified file could not be opened properly. The filename may be in use by another application, or may have too restrictive file permissions set. On Unix platforms, check the permissions on the directory and the named file; they must both allow writing by the user running the CobolScript program, and reading as well in the READING and APPENDING cases.

Error Number:	0125-0129
Error Message:	"You must open a file before you can { read from | write to | close | use POSITION on } it."
Description:	You are trying to perform a file operation on a file that has not yet been opened. Use the OPEN statement to open the file before performing any other file operations.

Error Number:	0130
Error Message:	"Cannot set buffering for file: "
Description:	There is a problem with the internal file buffering on your machine. If you get this error, raise a Problem Tracking Report (PTR) on the Deskware Registered User web site for further assistance.

Error Number:	0131
Error Message:	"Variables that are not alphanumerics or GLDIs cannot be used as file arguments: "
Description:	You used a non-alphanumeric, non-group item variable as a file argument in a file processing statement. Only alphanumeric variables and group-level data item variables can be used as filename arguments.

Error Number:	0132-0135
Error Message:	"Cannot find the FD for this file: "
Description:	The FD statement for the named file could not be found. Check that the FD statement exists. If it does, check for misspelling, or a wrong alphabetic case.

Error Number:	0136
Error Message:	"Invalid syntax in FD sentence."
Description:	A File Description sentence requires at least 6 tokens to be valid. See Chapter 3 for more on the syntax of FD.

Error Number:	0137
Error Message:	"Invalid syntax in READ statement."
Description:	At least 4 tokens are required for a properly defined READ statement. See the Language Reference for proper syntax of the READ statement.

Error Number:	0138
Error Message:	"Invalid target variable in READ statement: "

Description:	You have attempted to read a record from a file and move it into a variable that is either invalid or does not exist. Check to make sure that this variable has been properly defined prior to this statement.

Error Number:	**0139**
Error Message:	"End of file reached. Use AT END clause in READ statement to trap this error. File is: "
Description:	You have READ from a file and have encountered the end of the file. Avoid this error by placing an AT END MOVE <literal|var> TO <var> clause at the end of the READ statement. This will trap this error and set a variable to indicate the end of file has been reached. See the Language Reference for a full description of the READ statement.

Error Number:	**0140**
Error Message:	"Incorrect CobolScript syntax following AT END clause in READ statement."
Description:	A complete and valid CobolScript statement must follow the AT END clause in a READ statement.

Error Number:	**0141**
Error Message:	"Missing imperative following AT END clause in READ statement."
Description:	The AT END clause in this READ statement is missing an imperative statement after it. A complete and valid CobolScript statement must follow the AT END clause.

Error Number:	**0142**
Error Message:	"You are attempting to read data that contains a decimal into a variable with an implied decimal format. "
Description:	This error occurs when you are trying to read a field of data that contains a explicit decimal into a variable that has an implied decimal ("V") format. If your numeric data for this field in the record contains a decimal, you should use a decimal format for the variable that is to receive this data. This error can also occur when you are moving a group level data item to another group level data item and there is a mismatch of implied decimal and explicit decimal variables.

Error Number:	**0143**
Error Message:	"The file you are attempting to write to has not been described in an FD statement: "
Description:	You must describe all files used by a program in that program's FD (File Description); check for an appropriate FD entry for the named file, and create it if it does not exist. The FD describes the filename and its record size to CobolScript which must be done before any I/O operations can take place on the file.

Error Number:	**0144-0150**
Error Message:	"{ Write | Rewrite | POSITION statement } error. Possible directory/file permission problem, disk space problem, or file has not been opened: "
Description:	An error occurred while attempting to write to or position a file. The file may not have been opened with the OPEN statement, the permissions in the directory where the file exists may not be properly set to permit writing, or the disk may be full.

Error Number:	**0151, 0152**
Error Message:	"Error writing to file. File may not have been opened properly."

Description:	An error has occurred while trying to write data to a file. Verify that the permissions in the directory where the file exists allow writing by the user running the program. Also make certain that the file you are attempting to write to has been opened.

Error Number:	**0153**
Error Message:	"Rewrite error. Check data file record length against byte size declared in FD statement for: "
Description:	When using the REWRITE statement, the record byte size that you specify in the FD statement for the file must exactly match the actual physical record length of the file records. Determine the actual record length of your file record and use this value in the BYTE clause of your FD statement.

Error Number:	**0154-0157**
Error Message:	"POSITION offset argument is invalid. It should be a numeric integer variable or value: "
Description:	The offset argument to a position statement must be either an integer numeric literal, or a numeric variable containing an integer value. If you are specifying a variable as an argument, make certain that the variable is defined with a numeric picture clause, and that it does not have any post-decimal component.

Error Number:	**0158, 0159**
Error Message:	"POSITION offset argument is causing an out of range value for the record position: "
Description:	The argument that you have specified to the POSITION statement has a value that is pointing to a position that is either before the beginning of the file or after the end of the file.

Error Number:	**0160**
Error Message:	"POSITION statement error. Check data file record length against byte size declared in FD statement for: "
Description:	When using the POSITION statement, the record byte size that you specify in the FD statement for the file must exactly match the actual physical record length of the file records. Determine the actual record length of your file record and use this value in the BYTE clause of your FD statement.

Error Number:	**0161**
Error Message:	"Cannot open file: "
Description:	The specified file argument to DISPLAYFILE could not be opened. The filename may not exist as named on your machine (a file must already exist on your machine in order for it to be successfully opened for reading). On Unix platforms, check the permissions on the directory and the named file; they must both allow reading by the user running the CobolScript program.

Error Number:	**0162**
Error Message:	"Error in internal DISPLAY of: "
Description:	This error applies to an internal Deskware debugging command. It should not arise in your normal programming.

Error Number:	**0163, 0164**
Error Message:	"Error in { DISPLAY | DISPLAYLF } of: "

Description:	You have attempted to DISPLAY or DISPLAYLF an argument to standard output, but it failed. If the argument is a variable name or contains a variable name, check to see if this variable has been properly defined prior to the display statement.

Error Number:	**0165**
Error Message:	"No arguments specified."
Description:	No arguments have been specified in this DISPLAY statement. There must be at least one argument to DISPLAY.

Error Number:	**0166**
Error Message:	"Line too long in DISPLAY statement."
Description:	You have exceeded the maximum line length of 500 characters in this DISPLAY statement. Shorten this DISPLAY by placing a portion of the argument on another line in a new DISPLAY statement.

Error Number:	**0167, 0168**
Error Message:	"Missing ampersand or string delimiter in DISPLAY of literal."
Description:	A error in the DISPLAY of a literal has occurred. Check for missing string delimiters, or missing ampersands (&) between multiple DISPLAY arguments.

Error Number:	**0169-0191**
Error Message:	" TCP/IP is not currently available."
Description:	You are attempting to use a command that requires TCP/IP, but the TCP/IP protocol is not currently available on your machine. Make certain that you have an open internet or network connection utilizing TCP/IP before using this command.

Error Number:	**0192-0194**
Error Message:	"{ FTPCLOSE | FTPBINARY | FTPASCII } failed."
Description:	An FTP statement has failed. Your connection may have timed out prior to the execution of the statement.

Error Number:	**0195**
Error Message:	"The FTPCONNECT statement has failed to connect to the following server name: "
Description:	An attempt to connect to an FTP server has failed. Make certain that the named remote server is available, and is running as an FTP server. Also, if you are using variable names to contain your parameters, make sure they have been properly defined prior to this statement.

Error Number:	**0196**
Error Message:	"The FTPCD statement failed to change directories to the following path: "
Description:	The FTPCD command has failed. Verify that the named path exists on the remote server. If you are using a variable to hold the directory name, make certain it has been properly defined prior to this statement. It is also possible that the FTP connection timed out before the FTPCD statement could be successfully executed.

Error Number:	**0197**

Error Message:	"The FTPGET statement failed for the file: "
Description:	The FTPGET command has failed for the named file. Verify that the file exists on the remote server. Also, if you are using a variable to hold the file name, make certain it has been properly defined prior to this statement. It is also possible that the FTP connection timed out before the FTPGET command could be successfully executed.

Error Number:	**0198**
Error Message:	"The FTPPUT statement failed for the file: "
Description:	The FTPPUT command has failed. Verify that the local file exists in the specified path. If you are using a variable to hold the file name, make certain it has been properly defined prior to this statement. It is also possible that the FTP connection timed out before the FTPPUT command could be successfully executed.

Error Number:	**0199-201**
Error Message:	"The { SENDMAIL | GETMAILCOUNT | GETMAIL } statement has failed."
Description:	A mail command that you are trying to execute has failed. If you are using variables to hold the contents of your parameters, verify that they are properly defined prior to this statement. Verify that the userid and password are valid for the SMTP server you are trying to connect to, and verify that you can establish connectivity to the SMTP server from your local machine, using *ping* or a similar operating system command. Note: Some firewalls will prevent you from connecting and receiving data from a SMTP server. Also, some SMTP servers will not allow you to forward mail unless you have a valid userid and password.

Error Number:	**0202**
Error Message:	"The GETWEBPAGE statement failed."
Description:	The GETWEBPAGE command you are trying to execute has failed. If you are using variables to hold the contents of your parameters, verify that they are properly defined prior to this statement. Verify that the URL you are trying to GET is valid. Also, verify that you can establish connectivity to the remote server.

Error Number:	**0203, 204**
Error Message:	"{ CREATESOCKET | CLOSESOCKET } failed for socket: "
Description:	A socket command has failed. Check the contents of the TCP/IP return code variables for further information.

Error Number:	**0205-0210**
Error Message:	"Could not { use SHUTDOWNSOCKET on | BINDSOCKET to | LISTENTOSOCKET on | ACCEPTFROMSOCKET on | SENDSOCKET to | RECEIVESOCKET from } socket: "
Description:	A socket command has failed. Check the contents of the TCP/IP return code variables for further information.

Error Number:	**0211**
Error Message:	"The GETENV statement failed to get: "
Description:	If you are using a variable name as a parameter, verify that is has been properly defined prior to this statement.

Error Number:	**0212**

Error Message:	"Syntax error in GETENV statement."
Description:	Your GETENV statement syntax is incorrect. See the Language Reference for the correct syntax.

Error Number:	**0213, 0214**
Error Message:	"There was an error getting the { hostname | host by name } : "
Description:	There was a problem getting the hostname or host by name. If you are using a variable to store the hostname, make sure that it is properly defined prior to this statement.

Error Number:	**0215**
Error Message:	"There was an error with GETTIMEFROMSERVER."
Description:	There was an error getting the time from a server. If you are using a variable to store the hostname, make sure that it is properly defined prior to this statement. Also, make certain that the machine has a time daemon running on it – if it doesn't, you will not be able to get the time from it.

Error Number:	**0216-0222**
Error Message:	"{ TCPIP-HOSTENT-ADDRESS-TYPE | TCPIP-HOSTENT-ADDRESS-LENGTH | TCPIP-HOSTENT-ADDRESS | TCPIP-HOSTENT-ALIAS | TCPIP-HOSTENT-HOSTNAME | TCPIP-RETURN-CODE | TCPIP-RETURN-MESSAGE } not defined."
Description:	You attempted to use an internetworking command, but the named variable was not properly defined in your program. This variable must be defined when this command is used. See the Language Reference for the internetworking command you are attempting to implement.

Error Number:	**0223-0232**
Error Message:	"Maximum number of sockets exceeded."
Description:	Your socket command has exceeded the maximum number of sockets allowed in this version of CobolScript. The maximum number of sockets is 20.

Error Number:	**0233**
Error Message:	"Shutdown method must be 0,1, or 2."
Description:	SHUTDOWNSOCKET requires that a shutdown method equal to 0, 1, or 2 be specified.

Error Number:	**0234**
Error Message:	"SLEEP statement failed. Argument was : "
Description:	This error message should not be encountered with this version of CobolScript

Error Number:	**0235-0238**
Error Message:	"Module not defined: "
Description:	The named module could not be found, probably because of a misspelling. Verify that the module named references a valid module name.

Error Number:	**0239,0240**

Error Message:	"This PERFORM statement is missing an UNTIL keyword."
Description:	A PEFORM statement was found to be missing the UNTIL keyword in a case where UNTIL was required. See the Language Reference for proper syntax of PERFORM .. UNTIL.

Error Number:	**0241,0242**
Error Message:	"Missing UNTIL keyword in this PERFORM VARYING."
Description:	A PEFORM .. VARYING statement was found to be missing the UNTIL keyword. See the Language Reference for proper syntax of PERFORM .. VARYING.

Error Number:	**0243-0246**
Error Message:	"Invalid syntax in { inline PERFORM | PERFORM VARYING | PERFORM UNTIL } statement."
Description:	A PERFORM statement was found to have too few tokens to be valid. See the Language Reference for proper syntax of the type of PERFORM you are attempting to implement.

Error Number:	**0247-0250**
Error Message:	"The PERFORM VARYING statement must contain a { FROM | BY } keyword and value."
Description:	The FROM and BY keywords and values are required in PERFORM .. VARYING; the FROM value specifies the initial value for the varying variable, while the BY value specifies the quantity by which the varying variable will be incremented.

Error Number:	**0251,0252**
Error Message:	"The VARYING variable is not defined as numeric: "
Description:	The variable name after the VARYING keyword in the PERFORM .. VARYING statement was not defined with a numeric picture clause. Make sure that it is defined as a numeric prior to this statement.

Error Number:	**0253, 0254**
Error Message:	"Cannot initialize VARYING variable to the starting value: "
Description:	The variable name after the VARYING keyword in the PERFORM .. VARYING statement could not be initialized to the starting value (value after FROM keyword). Make sure that the VARYING variable is defined as a numeric, and that the FROM value is also a valid numeric.

Error Number:	**0255, 0256**
Error Message:	"Inline { PERFORM | PERFORM VARYING } must have code statements between the PERFORM and END-PERFORM."
Description:	In your version of CobolScript, inline PERFORMs must have statements between the PERFORM and END-PERFORM. Modify your code accordingly.

Error Number:	**0257**
Error Message:	"IF or PERFORM statement missing END-IF or END-PERFORM keyword."
Description:	Either an IF statement is missing an END-IF, or a PERFORM statement is missing an END-PERFORM. Check the IF and PERFORM statements in your code for their ending keywords.

Error Number:	0258-0261
Error Message:	"Missing END-IF for IF clause."
Description:	An END-IF is missing for an IF statement. Insert the END-IF.

Error Number:	0262-0264
Error Message:	"{ ELSIF | ELSE | END-IF } without initial IF clause."
Description:	The named keyword is not preceded by an IF clause. This keyword must succeed an IF clause for the syntax to be correct.

Error Number:	0265
Error Message:	"An ACCEPT statement has failed."
Description:	The ACCEPT statement has failed, possibly because one of the receiving variables is not defined. Alternatively, the syntax is incorrect. See the Language Reference for proper syntax of the ACCEPT statement.

Error Number:	0266
Error Message:	"The FROM keyword is missing from this ACCEPT statement."
Description:	The FROM keyword was missing in the ACCEPT statement. See the Language Reference for the proper syntax of the ACCEPT statement.

Error Number:	0267
Error Message:	"Error ACCEPTing standard input into: "
Description:	There was an error accepting standard input into the named target variable. Check the picture type and length of the variable, and make certain it is properly defined.

Error Number:	0268
Error Message:	"An error occurred while ACCEPTing CGI data from the web server."
Description:	An ACCEPT of CGI data from a web server failed. Make certain that there is data to ACCEPT; this can be verified by accepting and examining the environmental variable CONTENT_LENGTH; if CONTENT_LENGTH is populated with zero, then there is no data to accept from the web server.

Error Number:	0269
Error Message:	"Syntax error in ACCEPT statement."
Description:	A syntax error was found in the ACCEPT statement. See the Language Reference for the proper syntax of the ACCEPT statement.

Error Number:	0270
Error Message:	"Third parameter to GETBANNER must be a group-level data item: "
Description:	The third parameter of the GETBANNER function must be a 01 level group-level data item.

Error Number:	0271
Error Message:	"GETBANNER requires a group-level data item with 8 elementary data items as the target variable."
Description:	The GETBANNER command requires, as the target variable, a group-level data item with eight elementary data items defined as subvariables.

Error Number:	0272, 0273
Error Message:	"Unable to allocate memory required to { print | create } calendar."
Description:	There is not sufficient memory available when this program runs to successfully print or create the calendar. Either reduce the size of your program or the number of variables declared in your program, or use a computer with more memory.

Error Number:	0274, 0275
Error Message:	"{ CALENDAR | GETCALENDAR } year data type mismatch. Year input must be numeric: "
Description:	There is a data type mismatch - the year parameter to this statement must be a numeric literal, or a variable defined with a numeric picture clause. See the Language Reference for the proper syntax of the CALENDAR or GETCALENDAR statement.

Error Number:	0276, 0277
Error Message:	"{ CALENDAR | GETCALENDAR } month data type mismatch. Month input must be numeric: "
Description:	There is a data type mismatch - the month parameter to this statement must be a numeric literal, or a variable defined with a numeric picture clause. See the Language Reference for the proper syntax of the CALENDAR or GETCALENDAR statement.

Error Number:	0278, 0279
Error Message:	"Range error in month input to { CALENDAR | GETCALENDAR } statement. Following input month does not fall within range of 1-12: "
Description:	The month input to this statement must be a numeric value in the range 1 to 12 (January to December).

Error Number:	0280, 0281
Error Message:	"{ CALENDAR | GETCALENDAR } year must be greater than zero, but following input year is not greater than zero: "
Description:	Range error: The input year to this statement must be greater than zero.

Error Number:	0282, 0283
Error Message:	"{ CALENDAR | GETCALENDAR } statement does not support pre-Julian calendar dates (dates prior to August 1752)."
Description:	This statement does not handle pre-Julian dates, but your input is pre-Julian. Only dates after August 1752 will be correctly processed.

Error Number:	0284
Error Message:	"Third parameter to GETCALENDAR must be a group-level data item: "

Description:	The third parameter of the GETCALENDAR function must be a 01 level group-level data item with eight elementary data item subvariables.

Error Number:	**0285**
Error Message:	"GETCALENDAR requires a group-level data item with 8 elementary data items as the target variable."
Description:	The GETCALENDAR statement requires, as the target variable, a group-level data item with eight elementary data items defined as subvariables.

Error Number:	**0286**
Error Message:	"An argument to the EXECUTE statement contains an undefined variable or a literal that is missing delimiters."
Description:	Check the variable that is the statement argument for the EXECUTE statement. There is a problem with one of the underlying variable or literal constructs of your dynamic statement variable.

Error Number:	**0287**
Error Message:	"The syntax of the statement being executed is incorrect."
Description:	Check the variable that is the statement argument for the EXECUTE statement. There is a problem with the statement syntax.

Error Number:	**0288**
Error Message:	"Missing argument in EXECUTE statement."
Description:	Check the syntax of your EXECUTE statement and specify an appropriate argument.

Error Number:	**0289**
Error Message:	"Invalid CobolScript statement. This syntax is unsupported."
Description:	You have entered an invalid CobolScript statement. Check your syntax and try again.

Error Number:	**0290-0294**
Error Message:	"A { COMPUTE | ADD | SUBTRACT | MULTIPLY | DIVIDE } statement has failed."
Description:	Verify that the variables in your math statement are properly defined, and make certain that any expressions and function calls are valid.

Error Number:	**0295**
Error Message:	"Error encountered while evaluating COMPUTE expression."
Description:	Make sure that the variables you are using in the COMPUTE statement are properly defined, and that the expression has valid syntax.

Error Number:	**0296**
Error Message:	"Alphanumeric literals are not permitted inside computational statements."
Description:	String literals are not permitted inside computational statements; computational statements are used for mathematical operations only. To manipulate strings, use the MOVE statement.

Error Number:	0297
Error Message:	"Incorrect use of ROUNDED keyword in COMPUTE statement."
Description:	The ROUNDED keyword is used incorrectly in a COMPUTE statement. See the Language Reference entry for the COMPUTE statement for information on proper use of the ROUNDED keyword.

Error Number:	0298
Error Message:	"Invalid or missing variable name after REMAINDER keyword."
Description:	You have used the REMAINDER keyword in a DIVIDE statement, but have not specified a valid variable name to accept the remainder information. See the Language Reference entry for the DIVIDE statement for information on proper use of the REMAINDER clause.

Error Number:	0299, 0300
Error Message:	"Syntax error in REMAINDER clause in DIVIDE statement."
Description:	You have used the REMAINDER keyword in a DIVIDE statement, but the syntax is incorrect. See the Language Reference entry for the DIVIDE statement for information on proper use of the REMAINDER clause.

Error Number:	0301
Error Message:	"A GIVING clause must accompany DIVIDE statements that use the BY keyword."
Description:	You have used the BY keyword in a DIVIDE statement, but have failed to specify a GIVING clause. See the Language Reference entry for the DIVIDE statement for information on proper syntax.

Error Number:	0302
Error Message:	"Input length issue. Your expression must be smaller."
Description:	In this version of CobolScript expressions must be less than 180 characters total; the expression down into multiple statements, or make it less than 180 characters.

Error Number:	0303
Error Message:	"Expression expected in statement."
Description:	An expression was expected in the statement, but none was found. Check for the proper syntax of your statement in the Language Reference.

Error Number:	0304-0309
Error Message:	"Syntax error in expression."
Description:	Check the syntax of any expressions in this statement. If you are displaying an expression, make certain you separate multiple arguments with an ampersand. Refer to Chapter 3, *CobolScript Language Constructs*, for more information on expression syntax

Error Number:	0310, 0311
Error Message:	"Expressions are not allowed in this statement's syntax."
Description:	Expressions are not allowed in positional string referencing for this particular statement. You must first set a variable equal to the expression of interest using a COMPUTE statement, then substitute this variable for the expression in the positional string reference.

Error Number:	0312, 0313
Error Message:	"Maximum individual argument length exceeded in expression."
Description:	The maximum individual argument length in an expression has been exceeded. Insert spaces between expression elements to reduce argument length. The maximum individual argument length is 80 characters.

Error Number:	0314-0316
Error Message:	"Missing parenthesis in expression."
Description:	A parenthesis was missing from an expression. Insert the missing parenthesis.

Error Number:	0317
Error Message:	"Operator encountered without appropriate operand in expression."
Description:	An operator (+, -, /, *, %, etc.) was encountered in an expression but there was no target operand. In other words, the operator was a dangling operator. Insert the operand in the expression.

Error Number:	0318
Error Message:	"Non-numeric variables are not permitted here. Contents of this variable are non-numeric:"
Description:	You are attempting to use a non-numeric variable in a computational statement. Revise your statement to only use numeric variables.

Error Number:	0319
Error Message:	"Misplaced comma in expression."
Description:	You have placed a comma inside an expression when it should not be there. Commas are only valid in expressions as function parameter separators or occurs clause argument separators (Professional Edition only). Remove the misplaced comma.

Error Number:	0320
Error Message:	"Invalid terminating character in expression."
Description:	The terminating character in the expression is an operator or other inappropriate character. Check your expression syntax.

Error Number:	0321
Error Message:	"OCCURS clause group items are not allowed in expressions."
Description:	OCCURS clause group items are not allowed in expressions. Rewrite this expression using an elementary data item. If necessary, substitute an elementary data item directly for the OCCURS clause group item.

Error Number:	0322
Error Message:	"Group-level data items are not allowed in expressions."
Description:	Group-level data items are not permitted in expressions, since their value is not strictly numeric, but is comprised of other variables Modify the expression or condition to use only numeric literals and variables.

Error Number:	0323
Error Message:	"Missing parenthesis in function call."
Description:	A parenthesis is missing from a function call. Add the missing parenthesis.

Error Number:	0324
Error Message:	"Function called with wrong number of parameters. Function is: "
Description:	The function was called with the wrong number of parameters. Check the function syntax in the Function Reference.

Error Number:	0325
Error Message:	"Too many parameters in call to function."
Description:	The function was called with the wrong number of parameters. Check the function syntax in the Function Reference.

Error Number:	0326
Error Message:	"Undefined function."
Description:	A function is undefined. Check the spelling of your function name.

Error Number:	0327
Error Message:	"In function: "
Description:	Parameter error in function. Specific error messages always follow this message.

Error Number:	0328, 0329
Error Message:	"Division by zero is not permitted."
Description:	Your are attempting to perform division with a zero-valued divisor. Correct the expression so that division by zero does not occur.

Error Number:	0330
Error Message:	"Raising negative numbers to fractional powers is not permitted."
Description:	You are attempting to raise a negative number to the power of a fractional number. Even roots of negative numbers are imaginary numbers, which are not supported; however, because of the difficulty in determining whether a root is even or odd, raising negative numbers to any fractional power is prohibited. Modify the expression so that no negative numbers are raised to fractional powers.

Error Number:	0331
Error Message:	"Empty argument in reference modification or array variable."
Description:	No argument was specified in an array or reference modification. Insert a valid index in your positional string reference or array.

Error Number:	0332
Error Message:	"Empty second argument in reference modification."
Description:	The positional string reference was missing the second argument. Insert a valid numeric argument in your positional string reference or array.

Error Number:	0333
Error Message:	"Invalid syntax in reference modification."
Description:	You have specified more than two arguments or more than one separating colon in a reference modification argument. Proper reference modification syntax is of the form: variable-name(start-pos:end-pos)

Error Number:	0334, 0335
Error Message:	"Incorrect syntax inside reference modification or array variable. Value of argument must be a positive integer: "
Description:	All arguments to arrays and positional string references must be positive (strictly greater than zero) integers. Correct the argument that is not a positive integer.

Error Number:	0336
Error Message:	"Reference modification is not permitted with numeric variables or group-level data items."
Description:	Positional string referencing is not permitted with numeric variables or group-level data items. If you are working with a numeric or group-level data item variable and wish to isolate certain component characters or digits, move the variable to a group-level data item with the appropriate elementary items.

Error Number:	0337
Error Message:	"Internal error in variable processing."
Description:	Internal error in variable processing. This error should not be encountered in the course of normal programming.

Error Number:	0338-0341
Error Message:	"Internal error in processing of { ADD | SUBTRACT | MULTIPLY | DIVIDE } statement."
Description:	Internal error with math statement processing. This error should not be encountered in the course of normal programming.

Error Number:	0342-0344
Error Message:	"Internal error - stack space violation."
Description:	The expression stack limit was violated. Break your expression into multiple assignment statements with smaller component expressions.

Error Number:	0345-0348
Error Message:	"Internal error – *explanatory text*."
Description:	Internal error with expression evaluation. These errors should not be encountered in the course of normal programming.

Error Number:	0349-0352
Error Message:	"LinkMaker SQL { FETCH | CLOSE | OPEN } error: Cursor is not declared: "
Description:	Before you can use an SQL FETCH, CLOSE, or OPEN statement, your cursor must be declared using an SQL DECLARE.

Error Number:	0353
Error Message:	"LinkMaker SQL FETCH error: ABSOLUTE row number must be a positive integer. "
Description:	When using FETCH ABSOLUTE, the row number value following the ABSOLUTE keyword must be a positive integer, but the row number value that you specified was not.

Error Number:	0354
Error Message:	"LinkMaker SQL FETCH error: Invalid extended fetch type."
Description:	The extended fetch type (the keyword that follows FETCH in an extended fetch) is invalid. Valid extended fetch types are FETCH NEXT, FETCH PRIOR, FETCH FIRST, FETCH LAST, FETCH ABSOLUTE, and FETCH RELATIVE.

Error Number:	0355
Error Message:	"LinkMaker error: You have exceeded the maximum number of allowable cursors."
Description:	The maximum number of allowed cursors open at one time is 100. Reduce the number of open cursors in your program.

Error Number:	0356
Error Message:	"LinkMaker error: Data source returned an invalid column datatype."
Description:	The data source that you have established a connection with has returned a column datatype that is not a valid ODBC column datatype. Valid ODBC column datatypes are CHAR, VARCHAR, SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL, FLOAT, AND DOUBLE. Your database's ODBC driver must convert any native database datatypes to these ODBC datatypes, so this error indicates that a bug exists in your ODBC driver. Contact your ODBC driver vendor to report the problem.

Error Number:	0357
Error Message:	"LinkMaker error: SQL host variable not found: "
Description:	A host (receiving) variable was not defined. Host variables must be defined in your CobolScript program using normal variable definition syntax before being used in a LinkMaker SQL statement.

Error Number:	0358
Error Message:	"LinkMaker DBOPEN error: Return code variable not found: "
Description:	The return code variable for your DBOPEN statement was not defined.

Error Number:	0359
Error Message:	"LinkMaker DBCLOSE error: Return code variable not found: "
Description:	The return code variable for your DBCLOSE statement was not defined.

Error Number:	0360

Error Message:	"LinkMaker error: You must be connected to a data source before executing SQL statements."
Description:	You attempted to execute a LinkMaker SQL statement, but there is not data source connection established. Establish a connection with your data source first by using DBOPEN.

Error Number:	**0361**
Error Message:	"LinkMaker DBOPEN error: Data source name variable is not defined: "
Description:	The data source name specified in the OPENDB statement is not a validly defined variable. If you wish to specify a literal value instead of a variable, enclose the literal in string delimiters.

Error Number:	**0362**
Error Message:	"LinkMaker DBOPEN error: Data source user id variable is not defined: "
Description:	The data source user id specified in the OPENDB statement is not a validly defined variable. If you wish to specify a literal value instead of a variable, enclose the literal in string delimiters.

Error Number:	**0363**
Error Message:	"LinkMaker DBOPEN error: Data source password variable is not defined: "
Description:	The data source password specified in the OPENDB statement is not a validly defined variable. If you wish to specify a literal value instead of a variable, enclose the literal in string delimiters.

Error Number:	**0364**
Error Message:	"LinkMaker DBOPEN error: Return code must be a variable instead of a literal: "
Description:	You have specified a literal to accept the return code value, but the return code must be a previously defined variable.

Error Number:	**0365**
Error Message:	"LinkMaker DBCLOSE error: Return code must be a variable instead of a literal: "
Description:	You have specified a literal to accept the return code value, but the return code must be a previously defined variable.

Error Number:	**0366**
Error Message:	"LinkMaker error: Unable to establish connection with data source. SQLERRORMESSAGE is: "
Description:	There was a problem while attempting to connect with the data source. See the SQLERRORMESSAGE text following this error message for further details.

Error Number:	**0367**
Error Message:	"REPLICA variable's value parent is not already defined: "
Description:	A variable must be defined before a REPLICA of that variable can be defined. Check to make sure that the variable you are making a REPLICA of has been defined. Also, check to make sure that the spelling of the REPLICA variable and its parent are the same.

Error Number:	0368
Error Message:	"REPLICA variable does not have the same level number as its value parent."
Description:	The contents of the value parent cannot be properly duplicated if it has a different level number than the REPLICA variable.

Error Number:	0369
Error Message:	"REPLICA variable's value parent cannot be a group level data item."
Description:	The value parent of a REPLICA variable cannot be a group level data item. Only elementary data items can be used as value parents.

Error Number:	0370
Error Message:	"REPLICA variable cannot be a group level data item: "
Description:	A REPLICA variable must be defined as an elementary data item.

Error Number:	0371
Error Message:	"An argument to the CALL statement contains an undefined variable or a literal that is missing delimiters."
Description:	One of the arguments of the CALL statement is either an undefined variable, or a literal that is not enclosed in string delimiters. If it is a variable, verify that it has been defined and that you have used the correct spelling in the statement. If it is a literal, make sure that you put a string delimiter before and after the literal.

Error Number:	0372
Error Message:	"The syntax of the statement being called is incorrect."
Description:	See Appendix A for a description of the CALL command. You may use multiple arguments for the CALL statement. These arguments may be literals, keywords or variables. Verify that your literals are properly delimited and that your variables have been defined.

Error Number:	0373
Error Message:	"Missing argument in CALL statement."
Description:	The CALL statement requires at least one literal or keyword argument. See Appendix A for a description of the CALL command.

Error Number:	0374
Error Message:	"Variable not defined: "
Description:	The named variable is not defined. Check the variable name against the variables you have defined in your program.

Error Number:	0375
Error Message:	"This variable argument to the BYTES clause must be defined as a numeric: "
Description:	You are attempting to define the byte size of a record with a variable that is not defined as a numeric data type. Change the definition so that it is a numeric data type.

Error Number:	0376
Error Message:	"BYTES clause argument cannot have a value of zero."

Description:	You cannot specify a BYTES clause argument that has a value of zero. If you need to create data files with 0 bytes in them, use an arbitrary BYTES clause size (For example: 80) and then OPEN and CLOSE the file for WRITING. A file of zero length will be created.

Error Number:	**0377, 0378**
Error Message:	"The maximum number of permitted files has been exceeded. {*Explanatory text*}."
Description:	You have exceeded the maximum number of permitted files for your edition of CobolScript. If you have CobolScript Standard Edition, you should upgrade to the Professional Edition at https://www.cobolscript.com/cgi-bin/cobolscript.exe?catalog.cbl for support for greater numbers of variables. If you have the Professional Edition and you require additional variable support, call the number provided in the error message to order a custom edition of CobolScript with enhanced file support.

Error Number:	**0379-0386**
Error Message:	"The maximum number of allowed variable declarations has been exceeded. {*Custom text*}."
Description:	You have exceeded the maximum number of permitted variables for your edition of CobolScript. If you have CobolScript Standard Edition, you should upgrade to the Professional Edition at https://www.cobolscript.com/cgi-bin/cobolscript.exe?catalog.cbl for support for greater numbers of variables. If you have the Professional Edition and you require additional variable support, call the number provided in the error message to order a custom edition of CobolScript with enhanced variable support.

Error Number:	**0387**
Error Message:	"The EXECUTE recursive stack limit has been exceeded. Reduce the number of recursive EXECUTE calls in your program."
Description:	You have exceeded the maximum number of recursive calls using the EXECUTE statement. You must reduce the number of recursive calls in this particular EXECUTE recursion.

Error Number:	**0388-420**
Error Message:	"Variable not defined: "
Description:	The named variable is not defined. Check the variable name against the variables you have defined in your program. If you are using an array element like variable-name(counter), make sure that counter is not zero, and is in the allowable range for that particular OCCURS clause.

Error Number:	**0421**
Error Message:	"Cannot open file: "
Description:	The specified file argument to DISPLAYASCIIFILE could not be opened. The filename may not exist as named on your machine (a file must already exist on your machine in order for it to be successfully opened for reading). On Unix platforms, check the permissions on the directory and the named file; they must both allow reading by the user running the CobolScript program.

Error Number:	**0422**
Error Message:	"The argument of the GETHOSTNAME command must be a variable."
Description:	You must specify an alphanumeric variable (not a literal or numeric variable) to receive the value that is returned from the GETHOSTNAME command. The value returned will be the host name of your machine.

Error Number:	0423
Error Message:	“The email number must be a variable. "
Description:	You must specify a numeric variable (not a literal) that will receive the email number when you use the GETMAILCOUNT command.

Error Number:	0424
Error Message:	“The argument to the INITIALIZE command cannot be a literal."
Description:	You cannot use a literal as an argument to the INITIALIZE command. Specify a variable that is to be initialized as the INITIALIZE argument.

Error Number:	0425
Error Message:	"The last argument to GETBANNER cannot be a literal."
Description:	The last argument to the GETBANNER statement must be a group item variable with eight elementary item subvariables, rather than a literal or an elementary item. Once populated, it will contain the contents of a Unix-style banner. See the GETBANNER entry in Appendix A for more information about this command.

Error Number:	0426
Error Message:	"Non-numeric data and/or group items are not permitted in this statement."
Description:	Non-numeric literals and group item variables are not permitted in expressions, since their value is not strictly numeric, but is comprised of other variables Modify the expression or condition to use only numeric variables and literals.

Error Number:	0427
Error Message:	"This group item variable is too large to evaluate inside this statement: "
Description:	Group item variables must be less than 2000 bytes when they are tested or evaluated inside a condition. The group item variable that you are testing exceeds 2000 bytes in length.

Error Number:	0428
Error Message:	"Group item variables are not permitted in this type of expression."
Description:	Group item variables may only appear in conditions, not in COMPUTE or other mathematical expressions.

Error Number:	0429
Error Message:	"File mode could not be set to binary."
Description:	An error occurred while attempting to redirect standard output using DISPLAYFILE on a Windows®-platform computer. Reboot your computer and try again.

Glossary

A

alias An alternative name for something, such as a variable, file, or IP address.

algorithm Well-defined rule or process for arriving at a solution to a problem. Also, a step-by-step approach, in which improvement is made in every step until the solution is reached.

alphabetic character A character that belongs to the following set of letters: {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, the space character, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z} .

alphanumeric character Any character in the computer's character set.

ANSI American National Standards Institute. This organization is responsible for standards like ASCII and ANSI85 COBOL.

anonymous FTP Method for logging into certain FTP servers. Entering 'anonymous' as your login at an FTP site allows you to use resources that the system administrator has made available to the public.

API Application Programming Interface. A set of routines, protocols, and tools for building software applications.

AppMaker™ A feature of CobolScript Professional Edition that creates executables from CobolScript programs.

arithmetic expression A sequence of numeric variables, numeric literals, or other expressions, that are separated by arithmetic operators and can be enclosed in parentheses to show evaluation order.

arithmetic operator See *operator*.

argument An identifier, literal, or an expression that specifies a value to be used as input or a variable to hold output for a command.

array A series of variables, all of which are the same size and type. Each variable in an array is called an array element.

ASCII American National Standard Code for Information Interchange. This a standard based on a coded character set of 7-bit characters used for information exchange between computer systems.

B

bind To assign a value to a symbolic placeholder. During compilation, for example, the compiler assigns symbolic addresses to some variables and instructions, thereby binding them.

breakpoint The place in a program where execution may be interrupted by a user-specified intervention.

browser GUI-based hypertext client application, such as Netscape Navigator or Internet Explorer, which is used to access hypertext documents and services located on the Internet.

BSD Berkeley Software Distribution. Term used to describe any of a variety of Unix-type operating systems based on the UC-Berkeley BSD operating system.

buffer A storage area used for handling data in transit.

byte A string of bits, usually 8, that represents a character.

C

called program An external program that is executed through the use of the CALL command.

CGI Common Gateway Interface. This is the parameter-passing technique used to enable web browsers to pass data to web servers.

character A single unit of a language.

character set A list of all valid characters available on a computer.

client A computer program that requests services from another computer (called the server).

COBOL COmmon Business-Oriented Language. A business programming language, invented by U.S. Navy Rear Admiral Grace Hopper.

code A programming term loosely used to describe the statements in a computer program.

CodeBrowser™ The code browsing and colorizing component of CobolScript Professional Edition.

column The position in a text file, counting from left to right, that describes the placement of a character in a set of characters.

comment line A line of code in your program that has an asterisk in the eighth column and is not executed when the program is run. Usually comment lines are used for documentation purposes.

compiler A program than translates programs from a high-level language into machine language.

condition A special type of expression that is evaluated in order to determine the flow of logic. Conditions that evaluate to 1 are considered to be TRUE; all other condition values are FALSE.

Control Panel An administrative feature of CobolScript Professional Edition that provides GUI access to other Professional Edition features.

copybook An external file that contains a sequence of code that is to be included in a program at run time.

CRLF Carriage Return/Line Feed. One common combination of ASCII characters used to end one line of text and start a new one. Used on Windows® platforms.

cron A Unix system utility that executes tasks at regularly scheduled intervals.

cross-platform A term used to describe the ability to run a program on different operating systems without having to modify code.

cursor In SQL, the current row of a table being manipulated by one query statement. Similar to a pointer.

D

Data Division An optional division header of a CobolScript program that indicates where file descriptions and variable definitions will be located.

database A shared collection of logically related data, designed to meet the information needs of multiple users.

Database Administrator A person who is responsible for controlling the use, maintenance, and upkeep of computer databases. Commonly referred to as DBA.

data type A classification of a particular type of information. In CobolScript, there are numeric and alphanumeric data types.

DBMS DataBase Management System. A software product for keeping computerized records and managing the storage and retrieval of the data. Normally, the term DBMS is used to refer to *relational* databases, which store data in *tables* that have *rows* (records) and *columns* (fields).

DDL Data Definition Language. The language component of a DBMS that is used to describe the logical (and sometimes physical) structure of a database. DDL enables the definition of tables, indexes, and other database components.

debugging The process of finding and correcting errors that exist in a program's logic.

delimiter A character that identifies the end of a field in a record and the beginning of the next field; a field separator.

device Any machine or component that attaches to a computer.

device driver A program that controls an external device.

directory A special kind of file used to organize other files into a hierarchical structure. Directories contain bookkeeping information about files.

DNS Domain Name Service. An Internet service that maps symbolic names to IP addresses by distributing queries among the available pool of computers supporting the service. Also, Domain Name Server, used to describe a server that stores and provides these symbolic mappings.

DML Data Manipulation Language. A language component of a DBMS that is used by a programmer to access and modify the contents of a database.

domain A part of the DNS name. The domain to the right of the rightmost period is called the top-level domain. For example, in the domain name deskware.com, .com is the domain. In the name my.yahoo.au, .au is the domain.

download The process of copying files from another computer to your own computer via a network.

DSN Data Source Name. The named used to define an ODBC data source.

E

editor A software tool to aid in writing and modifying text documents; in programming, a tool that aids in editing program code.

elementary data item A variable that has a defined format and size; a CobolScript variable that does not have subvariable components.

encryption The process of coding (or scrambling) data so that it cannot be read by unauthorized parties.

Environment Division An optional division header of a CobolScript program used to describe the computer on which the program was developed and executed.

environment variable An operating system variable, external to the program and provided to the program by the operating system.

executable A term sometimes used to describe a binary file that can be directly launched and run by the operating system. Executable files contain machine code native to the computer platform on which they are running.

export To transfer data from one location/application to another in a format that is comprehensible to the receiving location/application.

extranet An organizational network that uses the public Internet as part of its infrastructure. Access to an extranet can be controlled by a number of security measures such as user logins and passwords. Often extranets are used so that information contained in them can be accessed externally by remote offices or business partners without the expense of creating a private WAN (Wide Area Network).

F

FAQ Frequently Asked Question. Document

which contains a list of commonly asked questions on a specific topic.

FreeBSD A free Unix operating system based on the Berkeley Software Distribution (BSD) of Unix.

field In web systems, an HTML form component that contains data submitted for processing by a user. In data files, an individual data component of a record.

file A term used to describe the place where data is stored on a disk system.

file system The operating system's management program that handles the request for input and output from disk storage devices.

fixed width record A file organization in which all records contain the exact same number of character positions, and individual fields begin in specific positions within the record.

FTP File Transfer Protocol. A protocol used to transfer files across remote machines. FTP requires that FTP server software be configured and running on the serving machine.

function A command that performs a calculation; the calculation should be related to the descriptive name of the function.

G

GIF Graphics Interchange Format file; the most common type of image file used on the Internet.

gldi See *group-level data item*.

group-level data item A variable that is made up of one or more subordinate variables or subvariables.

group item See *group-level data item*.

GUI Graphical User Interface. A graphical interface to a software program, that makes use of components such as buttons, menus, and toolbars, and relies heavily on mouse input for user interaction.

gzip A Unix compression program that reduces the size of a file and saves it with a .gz extension.

H

HDML Handheld Device Markup Language. A protocol similar to HTML, that defines a communication standard between web servers and handheld devices.

host A computer connected to a network.

hostent A name used to describe any set of variables that contains the results of a host name or address query against a DNS.

hostname The name of an individual computer on the Internet.

HTTP HyperText Transfer Protocol. The basic web protocol that defines the interaction between web browsers and web server.

hyperlink A highlighted, underlined phrase or word on a web page that can be clicked to go to another part of the page or to another web page.

I

Identification Division An optional division header of a CobolScript program that contains comments that describe the nature of the program and its intended use.

imperative A statement of code that is dependent on a condition being met by a previous statement of code. For example:

```
READ file INTO record
AT END
     MOVE 1 TO eof.
```

In the above statement, MOVE 1 TO eof is an imperative.

import To load and use data produced by another application/location.

INI A text file that contains configuration information. With respect to LinkMaker™, a configuration file with ODBC information.

inline code Statements inside a module that continue to execute until a certain condition is met, but do not call any other modules or external programs.

input A term used to describe the process of accepting data from a user or the act of reading data from a file.

interface Any type of information that is exchanged between computer programs or systems. This can be either exchanging files or a real-time data exchange.

Internet The term used to describe all the worldwide TCP/IP-based computer networks that are connected together.

intranet An internet-style network internal to an organization; an intranet can be used by anyone who is directly connected to the organization's network.

interpreter A special type of computer program that directly executes the instructions specified in a high-level programming language program file, without compiling the program code first.

I/O Input/Output; usually used to refer to file input and output.

IP Internet Protocol.

IP address An identifier that describes the unique location of a computer on a network or the Internet, similar to a telephone number. An IP address is expressed in numbers separated by periods, e.g., 25.92.80.170.

J

JavaScript A client-side interpreter that runs inside a web browser. Used to control actions in a user's browser between web server interactions.

JPEG Joint Photographic Experts Group file; a type of image file used on the Internet.

K

KB Kilobyte. 1000 bytes.

kernel Software that is the heart of an operating system.

keyword A special word that is reserved for use as a command, statement component, function, or identifier in a programming language.

L

level A number that refers to the hierarchy of a variable. If a variable has a level number greater than 1, then it is a subvariable of a higher-level group item variable.

literal A character string that is interpreted as the figurative constant is represents.

LinkMaker™ The database conduit technology that is integrated with CobolScript Professional Edition.

Linux® An open source Unix-like operating system initially developed by Linus Torvalds.

logical operator A reserved word that is used in evaluating conditions. AND, OR, XOR, and NOT are logical operators.

M

MIME Multipurpose Internet Mail Extensions.

module The term used to describe a separate paragraph of code. Modules normally are used to partition code into logically distinct units of work.

middleware Software that connects two otherwise separate applications.

N

network A number of computers physically connected to each other to enable inter-communication.

numeric A variable that is defined in a way that permits it to only have a value that contains numbers or numeric formatting characters. Numerics can be used in calculations and expressions.

O

ODBC Open DataBase Connectivity. A standard developed by Microsoft® that enables applications to access data from a variety of database management systems using SQL.

operator A single character that belongs to the following set:

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (mod)
^	raised to the power of

See also *unary operator*, *logical operator* and *relational operator*.

operand The variable that is modified or given a new value upon execution of a calculation.

OS Operating System. The software on a computer that controls the interaction between the user and the computer, and provides a platform on which other software can run.

output A term used to describe the displaying of data to a user or the writing of data to a file.

P

paragraph A term used to describe a unit of code that performs a specific piece of logic. See also *module*.

paragraph header A data name that is used as a descriptor of a paragraph in a program.

parsing The process of decoding data based on a set of predefined rules.

path A list of directories where the operating system looks for executable files if it is unable to find them in the working directory; an operating system environment variable that contains this information, that is normally set at system startup or user login.

picture clause Term used to describe the length and format of a variable.

POP3 Post Office Protocol 3. Protocol for retrieving email.

portable Term used to describe a program that can be executed on dissimilar operating systems without modification.

Procedure Division An optional division header used to indicate where the processing logic in a program will be located.

protocol Any standard that defines a structured method for systems to interact with each other.

pseudo-conversational A programming technique in which interaction only takes place when a request is made by a user.

R

random access The ability to access any record in a file by specifying the record number and using the POSITION statement.

readme file A text file copied onto software distribution disks that contains last-minute updates or errata that have not been printed in normal documentation.

record A term used to describe the logical units of data in a file.

redirection Directing input and output to files and devices other than the default I/O devices.

relational operator A reserved word or special character(s) used to describe a specific type of value comparison in a condition. In CobolScript, a member of the following set:

Operator	Meaning
=	equal to
EQUAL [TO]	equal to
NOT =	not equal to
NOT EQUAL [TO]	not equal to
>	greater than
GREATER [THAN]	greater than
<	less than
LESS [THAN]	less than
>=	greater than or equal to
NOT <	greater than or equal to
NOT LESS [THAN]	greater than or equal to
<=	less than or equal to
NOT >	less than or equal to
NOT GREATER [THAN]	less than or equal to

relational database A database whose records are organized into tables that can be processed by either relational algebra or relational calculus.

Replica A special type of CobolScript variable that may be defined in several places, but which points to the contents of a single variable.

reserved word A character string that has special meaning in a program.

root directory The top directory in a file system.

S

script A synonym for computer program, often used when the program is interpreted rather than compiled. Programmers often refer to their programs as scripts.

sentence A sequence of one or more statements that is terminated by a period.

sequential file A data file organization in which records are physically stored one after the other.

SMTP Simple Mail Transfer Protocol. A protocol used to send and deliver email.

socket A software structure that describes and identifies a logical end point for communications when using the TCP/IP protocol.

SQL Structured Query Language. A standard data definition and manipulation language for relational databases.

Sql State A value describing a database statement or cursor state, as defined by the X/Open and SQL Access Group SQL CAE specification (1992). Sql state values are strings that contain five characters.

SSL Secure Sockets Layer. A protocol developed by Netscape for transmitting private documents via the Internet.

standard input Term used to describe the normal data that is accepted by a program for processing.

standard output Term used to describe the normal output of data from a program.

string Group of alphanumeric characters enclosed within string delimiters.

string delimiter Character that indicates where a string begins and stops. By default, the string delimiter in CobolScript is the Gravé accent (`) symbol.

syntax The rules for placing and ordering terms, punctuation, and values when writing computer programs.

subdirectory A directory contained within another directory.

subscript A symbol or number used to identify an element in an array. Usually, the subscript is placed in brackets or parentheses following the array name, depending on language syntax.

SunOS® A Unix operating system developed by Sun Microsystems®.

T

tag A term used to describe the basic components that make up HTML documents.

TCP/IP Transmission Control Protocol/Internet Protocol. A relatively low-level protocol for communicating and transmitting data across networks and the Internet.

tar Tape ARchival program. A utility used to compress and uncompress files. These files usually have a .tar extension.

telnet A program that allows remote login to another computer.

text/html The MIME Content-type for HTML. This is used when displaying HTML data to web browsers.

text/plain The MIME Content-type for plain text. This is used when displaying plain text data to web browsers.

transaction A sequence of steps that constitute some well-defined business activity.

U

unary operator A plus or minus sign that precedes a variable or expression, or the logical operator NOT when used to negate a whole expression, e.g.:

Unary Operator	Expression
-	-(variable_val)
-	-(x + y + z – 2)
+	+(variable_val)
+	+(x + y + z – 2)
NOT	NOT (x > y)

Unix - An operating system, originally developed

by Ken Thompson of AT&T BellLabs.

unixODBC - A Unix-based implementation of the Microsoft® Open Database Connectivity standard.

upload The process of copying files from your own computer to another computer via a network.

URL Uniform Resource Locator. The naming scheme used to identify Web sites. Also, the text input box in a web browser where a URL name can be entered in order to directly navigate to a web site.

URL encoding A method used by web browsers to pass data to web servers. URL encoding replaces spaces with plus signs, and substitutes hex codes for a range of other characters.

V

variable A unit of data that has a specific format and size.

W

web browser A graphical application that allows a user to retrieve HTML (Hypertext Markup Language) documents from web servers across the Internet.

web server A program that runs on a computer and processes requests for HTML (Hypertext Markup Language) or other types of markup documents.

wild card character A special symbol that stands for one or more characters.

WML Wireless Markup Language. A protocol similar to HTML, that defines a standard way of communications between web servers and browsers on hand-held devices.

WWW World Wide Web. A large network of Internet servers providing hypertext services to client applications such as web browsers.

Z

zero suppression Used to describe the format of a variable that, when printed, will display spaces in place of zero characters.

ZIP A compression format that is used to combine many files into one file of smaller size with a .zip extension.

Index

\ (the backslash) 23
` (the Gravè accent) 23

A

A_Series DMS II 221
ABS 160
ACCEPT 66, 118, 246
ACCEPTFROMSOCKET 119
Access 216-218
ACOS 160
ACOSH 160
ACTION, FORM tag attribute 66
AcuCOBOL files 211, 221
ADABAS 210-220
ADABAS D 219, 221
Adaptive Server 220
ADD 120
Advanced PICK 215
Advanced Plus 217
Advantage Database Server 218
AI 41
algorithm 259
alias 259
aliases 75
AllBase/SQL 210-213
Alpha Microsystems 215
ALPHABETIC 38, 39, 161
alphabetic character 259
alphanumeric character 259
animate, interactive mode command 15
ANNUITY 161
ANNUITYFV 161
anonymous FTP 259
ANSI 259
API 259
AppMaker 112, 115, 259
argument 259
Arithmetic Commands
  ADD 120
  DIVIDE 129, 249
  MULTIPLY 146
  SUBTRACT 155
arithmetic operator 259
array 28, 259
AS/400 211-222
ASCII text 55, 59, 259, 260
A-Series 219
ASIN 161
ASINH 162
ATAN 162
ATAN2 162
ATANH 162
AUTHOR sentence 190
AutoCAD 209

B

BANNER 120, 121
BASISplus 211
bind 259
BINDSOCKET 78, 121
break, interactive mode command 15
breakpoint 259
browser 64, 264
BS2000 DBMS 212
Btrieve 214-218
Business BASIC ISAM 211
Butler SQL 211
byte 259

C

CA-Datacom/DB 210
CALENDAR 122, 247
CALL 56, 58, 122, 259
called program 259
CALTOJ 162
CA-Realia 218
CARRIGERETURN 24
CCTOIN 163
CEILING 163
CGI 63, 259
CGI Data, capturing 66
CGI directory 5, 7
CGI form components 86
  Checkboxes 89
  Hidden Fields 90
  List Boxes 88
  Radio Buttons 89
  Text Area 87
  Text Boxes 87
CGI Programming
  Environment Variables 84, 85
  Sending email using CGI input 92
character 259
character set 259
chmod 7, 15, 17, 18
CHOOSE 163
CINTOCC 163
C-ISAM 210-221
Clarion Top Speed 220
clear, interactive mode command 16
client 259
Clipper 215
CLOSE 47, 48, 122, 238
CLOSEDB 123, 223

CLOSESOCKET 123
CMTOIN 163
COBOL 259
CobolScript Professional
AppMaker 112, 115
CodeBrowser 109, 114
Control Panel 113
Using LinkMaker to access databases 223
code 259
CodeBrowser 109, 114
Controlling access to 110
column 259
Command line, running CobolScript from 10
commands
database 40
dynamic processing 41
email 40
file processing 39
FTP 40
general program control 39
TCP/IP 41
Unix shell-style 41
web processing 40
Comments 45, 259
compiler 259
COMPUTE 37, 124, 129, 171, 248
condition 37, 38, 142, 147, 260-262
Conditions
evaluation of 37
general rules of 37
syntax of 38
Type I 38
Type II 38
CONNECTTOSOCKET 80, 124, 125
CONTINUE 125
Control Panel, CobolScript 113
Conversion Functions
CALTOJ 162
CCTOCIN 163
CINTOCC 163
CMTOIN 163
FTOM 165
GMTOOZ 166
HPTOKW 166
INTOCM 166
JTOCAL 167
KGTOPD 167
KMTOML 167
KWTOHP 167
LTOGAL 168
MLTOKM 168
MTOF 168
OZTOGM 169
PDTOKG 169
COPY 33, 125, 144
Copy Book Commands
COPY 33, 125, 144
INCLUDE 33, 125, 143, 144
copybook 260
Copybook files 33
COS 164
COSH 164
count, interactive mode command 16
CREATESOCKET 78, 126, 243
CRLF 24, 260
csaccess file 110, 114
CTOFAHR 164
CTOS ISAM 219
c-tree Plus 211

D

D3 215
DARGAL server 211
Data Division 191, 260
Data files 32
Data Sources
Configuring in Unix 200
Configuring in Windows 195
Databases, relational, interacting with CS Standard 55
Datacom 210, 213, 217
DataEase 212
Datafit DP4 210
DataFlex 212, 220
Date Commands
ACCEPT 118, 246
CALENDAR 122, 247
GETCALENDAR 137, 248
DB2 209-222
DB2/2 220
dBASE 212, 215, 216
debugging 260
delimited data 21, 23, 32, 48, 50-53
delimiter 260
Delimiter, string 23
Depreciation Functions
DDBAMT 124, 164
STRLINEAMT 172
SYDAMT 172
design 100
design, program 99
deskware, interactive mode command 16
D-ISAM 210, 211, 214, 218
DISAM96 210
display 126
DISPLAY 68, 126
display, interactive mode command 16
DISPLAYASCIIFILE 94, 95, 127, 128
DISPLAYFILE 94, 95, 128
DISPLAYLF 68, 128
DIVIDE 129, 249
DL/1 ..... 210, 212
DMS 218
DMS II 211, 217
DMS-1100 219
DNS 75, 260
DNS Commands 75
GETHOSTBYNAME 75, 77, 85, 138

GETHOSTNAME 75, 139, 242
domain 260
DOUBLEQUOTE 12, 23, 24
DRDA 220
dump 233
dump, interactive mode command 16
Dynamic File Naming 104
Dynamic Statement Execution 105

E

Editing programs 9
Elementary Data Item 25
ELSE 142, 143
ELSIF 142, 143
email 139, 140
  sending, using CGI form input 92
Email Commands
  GETMAIL 74, 139, 140
  GETMAILCOUNT 74, 140
  GETMAILCOUNT 140
  SENDMAIL 73, 92, 153, 243
Empress 211
Environment Division 190, 260
Environment Variables 6, 84, 85, 260
Error Messages
  complete listing of 231
  HTML-based 19
  in command line mode 13
ESRI ARC/INFO Coverages 211
Essentia SQL Server 213
Excel 215, 216
EXEC SQL 130
EXECUTE 105, 131
EXP 164
expression 34, 259
Expressions
  Inside DISPLAY statements 103
  rules of construction 36
  Segment and OCCURS clause variable arguments 103
EXTFH 211, 221

F

FACT 165
FAHRTOC 165
FairCom 211
FAQ 261
FD 32, 33, 44, 48, 51, 52, 131, 238, 239
field 261
file 261
file descriptor 48
file system 261
Filemaker 212
Files
  Appending new records to 50
  Closing 48
  Copybook 33
  Data 32
  Describing 32
  Opening 48
  Reading records from 49
  Transferring 71
  Transmitting through the web 93
  Updating 52, 53
  Writing to 50
files, interactive mode command 16
FILLER variables 24, 27, 96
Financial Functions
  ANNIUTYFV 161
  ANNUITY 161
  FV 165
  PV 170
  PVANNUITY 170
FirstSQL 212
fixed width 32, 48, 51, 52, 88, 261
FLOOR 165
FOCUS 217
forms 124, 146
FoxPro 215, 216
FTP Commands
  FTPASCII 72, 132
  FTPBINARY 72, 133
  FTPCD 72, 133, 242
  FTPCLOSE 134, 242
  FTPCONNECT 72, 134, 242
  FTPGET 72, 73, 135, 243
  FTPPUT 72, 73, 135, 243
FTP, anonymous 71
Fulcrum Search Server 212
function 261
FUNDS System Databases 218
FV 165
FVANNUITY 166

G

GALTOL 166
GA-Power95 215
GA-R91 215
GENESIS 221
Geometric Functions
  ACOS 160
  ACOSH 160, 161
  ASIN 161
  ASINH 162
  ATAN 162
  ATAN2 162
  ATANH 162
  COS 164
  COSH 164
  PI 36, 169, 170
GETBANNER 136, 247
GETCALENDAR 137, 248
GETENV 137
GETHOSTBYNAME 75, 77, 85, 138
GETHOSTNAME 75, 139, 242
GETMAIL 74, 139, 140
GETMAILCOUNT 74, 140
GETTIMEFROMSERVER 141, 244

GETWEBPAGE....69, 141, 142, 243
GMTOOZ.... 166
GOBACK.... 142
Group-level data items 26
GT M.... 211
GUI.... 261
GURU 216
gzip.... 261

H

HDML.... 261
help screen.... 15
help, interactive mode command.... 15, 16
Hidden fields, using.... 90
Higher Math Functions
ABS.... 160
CEILING.... 163
EXP.... 164
FLOOR 165
LN 167
LOG 168
ROOT.... 171
SIGN 171
SQRT 36, 124, 172
HMP NX 221
host.... 261
hostent.... 261
hostname 75, 134, 138, 139, 141, 244, 261
HP 3000 Allbase.... 212
HP Eloquence databases.... 215
HP Image/SQL 216
HPTOKW.... 166
HTML
Forms, creating.... 66
Virtual HTML.... 64, 65
HTTP protocol.... 261

I

IBM AS/400.... 209
ICOBOL.... 211
Identification Division.... 190, 261
IDMS.... 209-213, 217, 218, 220
IDS II.... 210
IF 37, 142, 143, 168
Image/SQL 210, 212, 213, 217
imperative.... 261
implied decimal.... 240
Implied operators.... 37
implied PIC X(n).... 27, 68
Implied subjects.... 37
IMS....209-221
INCLUDE 33, 125, 143, 144
INFO DBMS 211
Infoman 217
Informix....210-221
Inforover.... 216
Ingres.... 217, 221
INITIALIZE 144, 237, 238
inline code 261
input 261
Inserts, table, in CS Standard 59
Installation
FreeBSD 1, 4, 5, 6, 18
Linux....1, 4, 5, 6
Microsoft Windows 1
SunOS....1, 4, 5
Interactive Mode Commands
animate 15
break 15
clear 16
count 16
deskware 16
display....16
dump listing 16
dump modules....16
dump positions....16
dump variables....16
files 16
help 15, 16
ip14, 16
list 14, 16
load 16
modules....16
positions....16
q 17
run....14, 17
save 17
stack....17
stepoff....17
stepon....17
variables....14, 17
ver....17
Interactive Mode, running CobolScript in 14
InterBase....219
Internal Line Number....14
Internet....261
Internet News Server....222
interpreter....4, 6, 261
INTOCM....166
IP....262
IP Address....262
ip, interactive mode command....14, 16
ISAM 209, 212, 214, 217, 220

J

JavaScript....95, 96, 262
jBase 215
JTOCAL 167

K

KB_SQL 214
KE Texpress ODBMS....214
KEYED1011....221
Keywords....21, 24, 262
KGTOPD 167
K-ISAM 214
KMTOML....167
Knowledge Man....216

Kubl 217
KWTOHP 167

L

LDAP 218
ldconfig 202
LEASY 212
level 262
LINC 221
LINEFEED 24, 65
LinkMaker 195, 223
list, interactive mode command 14, 16
LISTENTOSOCKET 78, 144, 145
literal 262
Literals
  Alphanumeric 22
  Numeric 22
LN 167
load, interactive mode command 16
LOG 168
logical operator 262
loopback address 65, 80
Lotus Notes 215
LTOGAL 168

M

M 213, 214, 217
maintenance 189
MAPPER 209
mbp 218
MEGAdata 215
Mentor Pro 215
Micro Focus COBOL files 214, 218, 221
MIME 262
MIME header 65, 94, 95
MIMER SQL RDBMS 220
MiniSQL 222
MLTOKM 168
Model 204 ..... 210, 217
modular programming 99
module 262
modules, interactive mode command 16
Monette 216
MOVE 145
MOVE Techniques
  Basic Moves 101
  ELDI to GLDI Moves 102
  GLDI to ELDI Moves 102
  Segmented Moves 101
MTOF 168
multidimensional arrays 30
MULTIPLY 146
Mumps 214
mvBase 215
MySQL 222

N

NAME, INPUT tag attribute 66
network 262
numeric 262
NUMERIC 38, 39, 168
NUMPMTS 169

O

Oberon 216
OBJECT COMPUTER sentence 190
Object/1 216
Objectivity/DB 217
ObjectStone 216
OCCURS clause variables 28
Ocelot SQL 217
ODBC 195
ODBC Drivers
  Alpha Microsystems 215
ODBC Drivers 209
  A_Series DMS II 221
  Access 216-218
  AcuCOBOL files 211, 221
  ADABAS 210-220
  ADABAS D 219, 221
  Adaptive Server 220
  Advanced PICK 215
  Advanced Plus 217
  Advantage Database Server 218
  AllBase/SQL 210-213
  AS/400 211-222
  A-Series 219
  AutoCAD 209
  BASISplus 211
  BS2000 DBMS 212
  Btrieve 214-218
  Business BASIC ISAM 211
  Butler SQL 211
  CA-Datacom/DB 210
  CA-Realia 218
  C-ISAM 210-214, 218, 221
  Clarion Top Speed 220
  Clipper 215
  CTOS ISAM 219
  c-tree Plus 211
  DARGAL server 211
  Datacom 210, 213, 217
  DataEase 212
  Datafit DP4 210
  DataFlex 212, 220
  DB2 209-222
  DB2/2 220
  dBASE 212, 215, 216
  D-ISAM 210, 211, 214, 218
  DISAM96 210
  DL/1 ..... 210, 212
  DMS 218
  DMS II 211, 217
  DMS-1100 219
  DRDA 220
  Empress 211
  ESRI ARC/INFO Coverages 211
  Essentia SQL Server 213

Excel 215, 216
EXTFH..... 211, 221
FairCom 211
Filemaker 212
FirstSQL..... 212
FOCUS..... 217
FoxPro..... 215, 216
Fulcrum Search Server..... 212
FUNDS System Databases..... 218
GA-Power95 215
GA-R91..... 215
GENESIS 221
GT M..... 211
GURU 216
HMP NX 221
HP 3000 Allbase 212
HP Eloquence databases 215
HP Image/SQL..... 216
IBM AS/400..... 209
ICOBOL..... 211
IDMS..... 209, 210, 213, 217-220
IDS II 210
Image/SQL..... 210-213, 217
IMS 209
IMS 210-221
INFO DBMS..... 211
Infoman..... 217
Informix 210-221
Inforover 216
Ingres..... 217, 221
InterBase 219
ISAM.....209, 212, 214, 217, 220
jBase..... 215
KB_SQL 214
KE Texpress ODBMS..... 214
KEYED1011 221
K-ISAM 214
Knowledge Man..... 216
Kubl..... 217
LDAP 218
LEASY..... 212
LINC 221
Lotus Notes 215
M.....213, 214, 217
MAPPER..... 209
mbp 218
MEGAdata 215
Mentor Pro 215
Micro Focus COBOL files 214, 218, 221
MIMER SQL RDBMS..... 220
Model 204 ..... 210, 217
Monette 216
Mumps 214
mvBase..... 215
Oberon..... 216
Object/1..... 216
Objectivity/DB 217
ObjectStone..... 216
Ocelot SQL 217
Open/A 219
OpenIngres.....210, 213-215, 219
Oracle 210-221
Ottero.....216, 218
Paradox.....215, 216
PASSdata.....215
PCIOS.....219
Pervasive SQL 218
PI Open.....215
PICK.....215
Poet ODBMS.....218
Postgres.....217
Progress 211, 215, 217-219
PROMIS 211
QSAM.....209, 217
Quadbase-SQL.....218
Raima.....219
RDA.....221
Rdb 210-214, 217, 220, 221
RDB.....212
RDMS.....217
RDMS-1100.....219
Reality/X.....215
Recital.....219
Red Brick.....214, 219
RFM II 210
RM COBOL 211
RM/COBOL 214
RMS.....210-214, 217, 220
RTXHDB.....219
SAP.....217
SAS.....219
Sequoia/Pro.....215
SESAM.....217
SESAM/SQL 212, 219
Sharebase 213
SNMP 219
SOLID 217, 220, 221
SQL Server 213
SQL Server 214-221
SQL/400 220
SQL/DS 213, 215, 220, 221
SQLBase.....210, 215
SQLDB 221
STX.....212
Superbase.....220
Supra.....217
Supra Server 213
Sybase.....211-221
Synergy databases.....220
System 1032 210
Tandem Enscribe 210
TANDEM Enscribe 214
Tandem NonStop SQL 210, 220
TANDEM SQL MP.....214
TANDEM SQL MX 214
Teradata 209-220
Thoroughbred files.....218
Times Ten Server.....220

T-ISAM 211
Titanium 216
TM1 209
TOTAL 213, 217
TurboImage 212
U/FOS 211
UDS 217
UDS/SQL 212, 219
UFAS 210
UltPlus 215, 217
UniData 215
UniData RDBMS 209
UniSQL 210, 216, 221
Unisys RDBMS 209
UniVerse 209, 215
Velocis 210, 217, 219
Versant 221
Viaserv Gateway 221
Vision indexed file system 209
VSAM 209-220
White Cross RDBMS 221
Xbase 220
XDB 215, 222
YARD-SQL 222
OPEN 47, 48, 146, 147, 238, 239, 240
Open/A 219
OPENDB 147, 223
OpenIngres 210, 213-215, 219
operand 262
operations, order of 35
operators 34, 259, 262
Oracle 210-221
Ottero 216, 218
output 262
OZTOGM 169

P

Paradox 215, 216
paragraph 262
paragraph headers 192, 262
parsing 12, 47, 68, 102, 104, 262
Intelligent variable 104
of CGI data 67
PASSdata 215
PATH variable 4
PCIOS 219
PDTOKG 169
PERFORM 88, 100, 147, 168, 245
PERFORM VARYING 149
Perl, interacting with 92
permissions, setting 7, 17
PERMUTAT 169
Pervasive SQL 218
PI 36, 169, 170
PI Open 215
PICK 215
Picture 262
picture clauses 181
alphanumeric 181
implied PIC X(n) 27
numeric 182
implied decimals 182, 185
signed 182
Numeric
literal decimals 182
numeric edited 182
asterisk check protection 183, 185
commas 182
cr control 184, 185
db control 184, 185
floating dollar sign 183, 185
floating minus sign 183, 185
floating plus sign 183, 185
minus sign control 184, 185
plus sign control 183, 185
zero suppression 183, 185
PIC X(n) 2, 22, 27, 181, 235
Poet ODBMS 218
POP3 Protocol 262
portable 262
POSITION 47, 53, 150
positional string referencing 88, 103, 126, 250, 252
positions, interactive mode command 16
POST method 63
Postgres 217
PostgreSQL 222
Probability Functions
CHOOSE 163
FACT 165
PERMUTAT 169
RANDOM 170
Procedure Division 44, 192, 263
program flow 39, 40, 41, 118, 142
PROGRAM-ID sentence 190
Progress 211, 215-219
PROMIS 211
protocol 263
Protocols
HTTP 261
POP3 262
SMTP 263
TCP/IP 263
pseudo-conversational 263
PV 170
PVANNUITY 170

Q

q, interactive mode command 17
QSAM 209, 217
QT Graphics Library 203
Quadbase-SQL 218

R

Raima 219
RAM 4
RANDOM 170
RDA 221
Rdb 210, 213, 214, 217, 220, 221

RDB 212
RDMS 217
RDMS-1100 219
READ 47, 49, 151, 239, 240
Reality/X 215
RECEIVESOCKET 80, 152
Recital 219
record definition 33, 48
record, defined 263
records 26, 32, 47
Red Brick 214, 219
relational operator 263
relative file processing 53
REMAINDER 129, 249
REPLICA 27, 184
reserved words 42, 263
REWRITE 47, 52, 152
RFM II 210
RM COBOL 211
RM/COBOL 214
RMS 210-214, 217, 220
ROOT 171
ROUNDED 36, 117, 124, 129, 146, 171
RTXHDB 219
run, interactive mode command 14, 17

S

sample programs 19, 177
SAP 217
SAS 219
save, interactive mode command 17
script 263
Segmented Moves 101
Selects, table, in CS Standard
  Dynamic 57
  Static 56
SENDMAIL 73, 92, 153, 243
SENDSOCKET 80, 154
Sentence 263
Sentences 44
sequential file 263
Sequoia/Pro 215
SESAM 217
SESAM/SQL 212, 219
SET 154, 238
Sharebase 213
SHUTDOWNSOCKET 123, 155, 243, 244
SIN 36, 171
SINGLEQUOTE keyword 23
SINH 172
SMTP 73, 74, 92, 243
SNMP 219
sockets 77, 244, 263
SOLID 217, 220, 221
SOURCE COMPUTER sentence 190
Source Line Number 14
SPACE 24
SPACES 24
Spaces, CGI input fields and 87
SQL Commands
  ALTER TABLE 226
  CLOSE 226
  COMMIT 226
  CREATE INDEX 226
  CREATE TABLE 227
  DECLARE 227
  DELETE 227
  DROP INDEX 227
  DROP TABLE 228
  FETCH 228
  INSERT 228
  OPEN 228
  ROLLBACK 229
  SELECT 229
  UPDATE 229
SQL Server 213-221
SQL, embedded 223
SQL/400 220
SQL/DS 213, 215, 220, 221
SQLBase 210, 215
SQLDB 221
SQRT 36, 124, 172
stack, interactive mode command 17
standard input 263
standard output 263
Statements 43
stepoff, interactive mode command 17
stepon, interactive mode command 17
STOP RUN 142, 156
string delimiter 12, 23
STRLINEAMT 172
STX 212
subscript 29, 263
SUBTRACT 155
Superbase 220
Supra 217
Supra Server 213
Sybase 211-221
SYDAMT 172
Synergy databases 220
syntax 263
syntax, command 117
System 1032 210

T

tag 263
TAN 173
Tandem Enscribe 210
TANDEM Enscribe 214
Tandem NonStop SQL 210, 220
TANDEM SQL MP 214
TANDEM SQL MX 214
TANH 173
tar 263
TCP/IP Socket Commands
  BINDSOCKET 78, 121
  CLOSESOCKET 123
  CONNECTTOSOCKET 80, 124, 125

CREATESOCKET 78, 126, 243
LISTENTOSOCKET 78, 144, 145
RECEIVESOCKET 80, 152
SENDSOCKET 80, 154
SHUTDOWNSOCKET 123, 155, 243, 244
telnet 263
Template, CobolScript program 187
Teradata 209-217, 220
text/html 65, 263
text/plain 264
Thoroughbred files 218
Times Ten Server 220
T-ISAM 211
Titanium 216
TM1 209
TOTAL 213, 217
TurboImage 212

U

U/FOS 211
UDS 217
UDS/SQL 212, 219
UFAS 210
UltPlus 215, 217
unary operator 264
UniData 215
UniData RDBMS 209
UniSQL 210, 216, 221
Unisys RDBMS 209
UniVerse 209, 215
Unix 9, 17
unixODBC 200
unixODBC Drivers 222
DB2 222
Internet News Server 222
MiniSQL 222
MySQL 222
PostgreSQL 222
YARD-SQL 222
Updates, file record 50
Updates, table, in CS Standard 60
URL 264
URL encoding 264

V

VALUE, Input tag attribute 89
variables 24, 264
Variables
Basic Moves 101
Manipulating CobolScript Variables 101
Segmented Move 101
variables, interactive mode command 14, 17
Velocis 210, 217, 219
ver, interactive mode command 17
Versant 221
Viaserv Gateway 221
Vision indexed file system 209
VSAM 209-218, 220

W

web 141
Web page input, capturing 66
Web pages, retrieving 69
web server 264
White Cross RDBMS 221
WML 264
WRITE 50, 156

X

Xbase 220
XDB 215, 222

Y

YARD-SQL 222

Z

ZERO 24
zero suppression 264
ZEROS 24