Microsoft Windows®
Microsoft Windows 95™
Microsoft Windows NT™

Getting Started with Fujitsu COBOL

First Edition: January 1997

Preface

For many years, COBOL programmers were constrained to developing applications using relatively simple text-based interactive user interfaces. While the vast majority of these applications were developed for mainframe environments, the COBOL world has fairly recently experienced a metamorphosis. This change has been driven by the acceptance of microcomputers (PC's) as serious production-oriented platforms for delivering critical business applications.

Some PC COBOL vendors have attempted to deliver modernized versions of COBOL, along with graphical development tools. For the most part, however, these products have proven to be either highly unstable, devoid of key features, or simply not serious production-oriented development environments.

To further complicate the situation, these attempts have typically presented additional problems to the production-oriented developer. That is, some of these products force the usage of monolithic legacy runtime systems, some of which even contain large portions of code dedicated to mainframe compatibility which are not needed in the PC production environment. A simple 10 line COBOL program may actually require the packaging of megabytes of supporting run time code for distribution.

Once having gone through this development process and deployed multiple applications to end users, other potential pitfalls may develop. Because of the large and cumbersome nature of these legacy-supporting run time systems, a simple vendor-supplied update may in fact induce problems on previously developed applications.

Additionally, many of these vendors require the developer to pay some form of run time system royalty to deploy these applications to end users. It is well known in the corporate sector that simply tracking these royalty-based run time licenses causes more expense and headaches than the actual fees paid to the vendor.

In recognizing the above noted pitfalls, Fujitsu decided to develop a truly new generation COBOL development product suite from the ground up. Heavy emphasis was placed upon mission critical, production applications and performance, as well as adherence to existing standards.

The result of this effort is Fujitsu's three tiered family of COBOL Development products.

The first tier product is Fujitsu COBOL 3.0. This provides a fully functional COBOL development environment, designed to allow COBOL programmers to develop powerful COBOL applications for the Microsoft Windows family of platforms. It includes support which allows users to develop COBOL applications that interact with Microsoft's Visual Basic GUI development environment.

The second tier product, Fujitsu PowerCOBOL Professional Edition, builds on top of Fujitsu COBOL and adds a world class GUI Development Environment and a high end, full function SORT facility. The GUI development environment allows COBOL programmers to develop sophisticated GUI applications, using native COBOL code extensions. It provides an integrated GUI painter to aid this development, and client/server development technology.

The third tier product is Fujitsu's Flagship PowerCOBOL Enterprise Edition. This builds upon the first two tiers and adds in PowerGem Plus, a high end, full function SORT facility plus other powerful COBOL development tools. PowerGem Plus

provides sophisticated source code management capabilities for standalone and collaborative development teams.

Fujitsu PowerCOBOL represents the ultra modern COBOL development environment. It provides a rich array of tools, and has encompassed Graphical User Interface (GUI) development directly into the COBOL language.

Information on PowerCOBOL may be obtained directly from Fujitsu by calling **(800) 545-6774** or **(408) 428-0300**. You may also find information at Fujitsu COBOL's worldwide web Internet site (**www.adtools.com/powercobol**).

The product you are about to install provides a full-featured 32 bit development environment for COBOL programs. It allows developers to develop COBOL programs which integrate easily with Microsoft's Visual Basic Development Environment.

Audience

Prior to using Fujitsu COBOL 3.0, it is assumed that you have the following knowledge and have completed the following tasks:

- You have some basic understanding as to how to navigate through and utilize Microsoft Windows 95 and / or Microsoft Windows NT, depending on which environment you are using.
- You have the appropriate hardware and software configuration as described in Chapter 2.
- You understand the COBOL language from a development perspective.
- If you plan on using Microsoft's Visual Basic development environment, you have spent some time using Visual Basic to get a feel for it's interface and capabilities.

It is highly recommended that you go through the Visual Basic tutorial of building your first simple Visual Basic application. This tutorial may be found in the “Visual Basic Programmer’s Guide.” While it does not encompass COBOL, it will familiarize you with the Visual Basic development environment.

You will carry forward the experience of using Visual Basic and it’s controls and concepts when writing GUI COBOL applications.

How this Manual is Organized

This manual contains the following information:

Chapter 1	Introduction
Chapter 2	Installing Fujitsu COBOL 3.0
Chapter 3	A Quick Tour
Chapter 4	Creating Your First Visual Basic COBOL Application.
Chapter 5	Integrating COBOL Programs to Visual Basic
Chapter 6	COBOL 32 Bit Run-time Files
Appendix A	Sample Programs

Conventions Used in This Manual

Example of convention	Description
setup, `setup`	Characters you enter appear in bold.
Program-name	Underlined text indicates a placeholder for information you supply.
ENTER	Small capital letters are used for the name of keys and key sequences such as ENTER and CTRL+R. A plus sign (+) indicates a combination of keys.
…	Ellipses indicate the item immediately preceding can be specified repeatedly.
Edit, Literal	Names of menus and options appear with the initial letter capitalized.
[def]	Indicates that the enclosed item may be omitted.
{ABC | DEF}	Indicates that one of the enclosed items delimited by | is to be selected.
CHECK WITH PASCAL LINKAGE ALL PARAGRAPH-ID COBOL ALL	Commands, statements, clauses, and options you enter or select appear in uppercase. Program section names, and some proper names also appear in uppercase. Underlined text indicates the default.

`PROCEDURE DIVISION`
`:`
`ADD 1 TO POW-FONTSIZE OF LABEL1.`
`IF POW-FONTSIZE OF LABEL1 > 70 THEN`
`MOVE 1 TOW POW-FONTSIZE OF LABEL1.`
`END-IF.`
	This font is used for examples of program code.

The *sheet* acts as an application creation form.	Italics are occasionally used for emphasis.
"COBOL85 User's Guide" See "Compile Options" in Chapter 5.	References to other publications or sections within publications are in quotation marks.

Related Manuals

Related documentation is listed below:

When creating a user program in COBOL:

Fujitsu COBOL Debugging Guide
COBOL85 User's Guide
COBOL85 Reference Manual

Trademarks

MS-DOS is a registered trademark of Microsoft Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

Windows is a registered trademark of Microsoft Corporation.

Windows 95 is a registered trademark of Microsoft Corporation.

Windows NT is a registered trademark of Microsoft Corporation.

COBOL/2 is a registered trademark of Micro Focus Inc.

Table of Contents

Chapter 1. Introduction

This chapter introduces Fujitsu COBOL and outlines its major functions. The following topics are discussed:

- A brief history
- Fujitsu COBOL 3.0
- Event-driven programming
- Using the Fujitsu COBOL 3.0

A Brief History

It is generally agreed upon that COBOL programs make up anywhere from 50-75% of the world's current business applications. Today COBOL programmers need a modern, powerful, and easy to use development environment that fully exploits the Microsoft Windows family of operating systems for both development and production.

The Microsoft Windows family of operating systems has become the preferred end user platform for the vast majority of PC users. A number of pitfalls arose, however, as traditional COBOL programmers from the mainframe world attempted to embrace this exciting new platform.

Primarily, the intense time and complexity of low level Windows API (application programming interface) programming prevents most developers from creating sophisticated graphical user interfaces using traditional 3GL (3rd generation language) compilers and tools.

Two approaches have been developed to deal with this problem. The first is to extend the COBOL language itself to encompass GUI capability directly in the language. This approach offers the advantage of utilizing a single COBOL development environment, and keeping the entire application in native COBOL. Fujitsu's flagship PowerCOBOL product is the best example of such an approach.

The second approach is to separate the user interface from the actual COBOL application and have it managed by a wholly separate facility. Some COBOL vendors have in the past attempted to deliver such a combination, but these have fallen short for a variety of reasons. Primarily, these environments have proven to be highly unstable, lacking in many features, and very

proprietary in nature. Additionally, these environments have been in a constant state of flux as vendors attempted to deliver better stability and a wider array of functionality. The results have typically been less than successful for developers.

In recognizing these challenges, Fujitsu has developed the world's premier COBOL GUI and Client Server application development environment -- Fujitsu's flagship PowerCOBOL product line.

Fujitsu has also recognized the need for a world class COBOL development product that will integrate, coexist and exploit Microsoft's Visual Basic development environment. The product you are about to install is a fully standalone COBOL development environment. It does not, however, provide a GUI development environment.

If you want to develop full-fledged GUI applications, you should inquire about Fujitsu's PowerCOBOL product (**www.adtools.com/powercobol**), which is a superset of the product you are about to install. You may additionally utilize Microsoft's Visual Basic (version 4.0 or later) with this product to develop GUI applications.

Visual Basic was an attempt to bring out a 4GL-like development environment that would allow programmers and even non-programmers to develop GUI applications in the Windows environment by visually painting the user interface, and then creating the rest of the application by writing small snippets of code using a proprietary scripting language which was loosely based upon the BASIC programming language.

Initially Visual Basic was lacking for serious business application development. Microsoft has continued to evolve Visual Basic over the years into a serious development environment. Today it is a strong development tool for a wide number of professional developers who want to rapidly create GUI applications for the Windows family of platforms.

For the users of COBOL applications, however, Visual Basic presents a couple of substantial challenges:

- Having no relationship to COBOL, Visual Basic presents an alien development environment to COBOL programmers. They must learn an entirely new development methodology and language.
- Visual Basic does not provide all of the facilities found in the COBOL language.
- Prior to Fujitsu COBOL, Visual Basic could not easily be inter-mixed with COBOL programs to combine the best of both worlds, and to allow the re-use of existing COBOL programs.

Many COBOL programmers have requirements to re-host applications (move COBOL applications from another platform such as the mainframe) to the PC environment. They want to at the same time re-design traditional text based interfaces to take advantage of GUI.

While some COBOL vendors attempted to provide “bridges” to allow Visual Basic to intermingle with COBOL, these approaches typically fell short.

One of the major inhibiting factors in inter-mixing other vendor’s COBOL with Visual Basic related to the fact that each had it own separate and complex run time system. Mixing two very proprietary run time systems is a substantial undertaking. Most of these legacy run time systems were architected and developed by their respective vendors long before PC’s, Windows and GUI’s were even invented.

Fujitsu Addresses the COBOL Need

Microsoft over the years has recognized the need to support COBOL applications and has made multiple attempts to provide products to the COBOL community. Their original COBOL compiler was replaced by Micro Focus' COBOL/2 product in 1988. The COBOL/2 product suffered from many of the above mentioned pitfalls, and Microsoft withdrew it from the market in June of 1993.

Fujitsu recognized this dilemma and set out to design and build a new generation COBOL development environment from the ground up. A special emphasis was placed upon the design of the run time system to alleviate the above noted problems.

Fujitsu additionally developed a tool to aid in the conversion of COBOL/2 applications to Fujitsu COBOL (while COBOL is a standard language, Micro Focus has added a significant number of proprietary extensions).

Fujitsu COBOL 3.0

The Fujitsu COBOL 3.0 is a complete COBOL development environment that allows you to create standalone COBOL applications and/or COBOL components for use with Microsoft Visual Basic and Visual C++ applications.

The Fujitsu COBOL PROGRAMMING-STAFF (P-STAFF) facility is a 32 bit project-oriented development environment which provides integrated access to an Editor, Compiler, Debugger, Execution, and other supporting tools. A sophisticated data file editor is also included. It supports all COBOL data file types, and provides editing, conversion, printing, sorting, reorganizing and recovery capabilities.

For previous users of Micro Focus' or Microsoft's COBOL/2, the MFTO85 utility is included to aid in converting COBOL programs utilizing some of the Micro Focus proprietary extensions to run under Fujitsu COBOL.

Fujitsu PowerBSORT OCX is also included. PowerBSORT OCX is callable from any Windows application that supports OCX's, for example, Microsoft Visual Basic.

PowerBSORT OCX online help is provided along with sample applications for Visual Basic 4.0 and 5.0. The online help and sample files are automatically installed along with PowerBSORT OCX. **Note**: PowerBSORT OCX must be registered. Double click on Register PowerBSORT OCX (in the Fujitsu COBOL 3.0 folder) to register the OCX to Windows.

Event Driven Programming

In order to understand how to develop GUI COBOL applications in the Windows environments, one needs to understand the concept of Event Driven Programming.

While the term “event driven” may be somewhat new in the computer industry, the concept has been around for a long time, and chances are that if you are a COBOL programmer, you already understand it.

When COBOL programmers first began developing mainframe-based applications that interacted with users (e.g. non-batch style applications), they were constrained to simple line-oriented ACCEPT/DISPLAY syntax.

Later, on-line transaction monitoring systems such as IBM’s CICS and IMS/DC enhanced this capability to handle full screen-oriented interaction (as opposed to line oriented). Some PC COBOL vendors later enhanced the COBOL ACCEPT/DISPLAY verbs in non-standard ways to provide full screen interaction.

While one could argue that none of the above mentioned techniques qualify as event driven programming, some were in fact, a very primitive form of this approach.

Event driven programming breaks the user interface portion of a GUI application (screen I/O, keyboard I/O, mouse I/O, and a few other possibilities), into a series of possible events.

These events include such user actions as pressing a key on the keyboard, moving the mouse, clicking on some screen object such as a button with the mouse, holding the mouse button down and dragging the mouse to move a window, etc.

Writing a proper Windows GUI application entails creating all of the potential windows and objects that the user may interact with

(including output objects as well), determining all of the possible events that may occur, and creating a link between each potential event and your application code to handle the event.

What you eventually hope to wind up with is an application that registers its user interface with Windows (in much the same approach as CICS and IMS/DC applications register themselves in the mainframe world). Once the application is registered, Windows recognizes events and sends event driven message to the application for resolution.

The registration and event handling processes are based upon the Windows message queue paradigm.

The Windows Message Queue

The Windows message queue is an exceptionally complex topic, and will be discussed only briefly here to give you a basic understanding.

Much like CICS or IMS/DC (via VTAM) on a mainframe, Windows controls all input from the screen, mouse and keyboard. It must then implement a technique to manage all of this and ensure that any event that takes place gets routed to the proper application. Likewise for applications that wish to update system I/O resources such as the screen, Windows needs a mechanism to enable applications to alert it to do so.

Windows provides the above noted services by implementing a message queue. Every Windows application must register itself with Windows when it activates and messages are then routed into each application as appropriate as the user interacts with the system. Because Windows supports multiple applications running concurrently, this proves to be a very busy part of the operating system.

This means that any Windows application basically registers itself with Windows, and then goes into a recurring “message

loop", simply reading in messages sent to it by the operating system, and reacting as appropriate, until the application receives whatever it has defined to be a terminate type of message and terminates itself.

It's important to understand that an application can easily receive thousands of Windows event messages in a short period of time. For example, when a user moves the mouse around the screen, multiple messages are generated to alert the application whose window(s) the mouse passes over. These messages not only alert the application that the mouse has moved over the application's window, but additionally provide information such as the exact location (x, y coordinates) where the mouse moved - even though the user did not press one of the mouse buttons!

If your application does not contain code to handle a certain message, then it may lock up or be abended. Fortunately, a common programming technique is to provide a catch-all routine that simply ignores all other messages than the ones you've defined that you care about.

Hopefully, you can begin to envision the Windows application environment where messages are being sent to applications as the user interacts with the system. The applications perform actions based upon the messages received and dispatch messages back to Windows to perform user interface operations (e.g. close a window, create a new window, update the screen with the new data being passed along, etc.).

In a low level Windows application, developers are forced to write extensive amounts of code to deal with all of the possible events that may occur, manage the internal message loop, and take case of a great deal of tedious work. This tedious work may even include being forced to re-paint the screen because another application's window was placed on top of your application's window (Windows does not automatically keep track of the layering of application windows and will not automatically restore the prior contents of a window that was briefly overlaid).

How PowerCOBOL and Visual Basic Simplify Application Development

One of the goals of Fujitsu's PowerCOBOL and Microsoft's Visual Basic is to effectively hide the tedious task of managing the Windows message queue and the application's message loop from the developer.

Using PowerCOBOL or Visual Basic, a developer need only paint the application's windows and contained objects (i.e., buttons, list boxes, text fields, etc.). PowerCOBOL users may program this in intuitive COBOL code, while Visual Basic developers define in a high level scripting language, how they are to interact with the user and the application. The developer then decides which potential events he or she actually cares about (for example, a button being pushed by a mouse click), and writes the code to handle each such event.

PowerCOBOL actually generates a great deal of the COBOL source code for GUI events and places it automatically into the COBOL program.

This approach frees the programmer from the tedious nature of low level Windows API message queue programming, so he or she can concentrate on higher level aspects of the application.

Both PowerCOBOL and Visual Basic actually generate code for the application to handle any potential events that the developer forgets or simply does not care to define event code for.

Because some of this code can be quite tedious and complex (e.g. re-sizing a window and all of its objects re-positioned), both PowerCOBOL and Visual Basic become great allies to the application developer who needs to field solid, full functional GUI applications.

When developing these types of applications using Visual Basic, users must be very familiar with Visual Basic's scripting

language, which is very extensive and can be quite complex at times.

Additionally, developers sometimes run into development scenarios where Visual Basic may not provide a certain capability or provides it in a somewhat tedious manner versus another programming language such as “C” or COBOL.

Lastly, developers often have legacy applications available that they would like to encompass in a GUI application. These legacy applications are often written in something other than Visual Basic script. This means that Visual Basic needs to be able to call programs written in languages other than its own scripting language. This is exactly where Fujitsu COBOL comes into play.

By providing the capability to associate potential events in a Visual Basic controlled application to COBOL programs, the development environment is greatly extended. COBOL programmers can write sophisticated GUI applications with a minimal amount of knowledge of Visual Basic’s scripting language. They may additionally re-use existing COBOL code with little or no change within a Visual Basic application as well.

Using Fujitsu COBOL 3.0

Creating COBOL applications to run standalone or under Visual Basic is a flexible and straightforward process. Developers typically follow the steps below:

- Application analysis and design.
- For GUI applications, creation and testing of the user interface using Visual Basic's development environment.
- Creation and testing of the COBOL component(s) of the application using Fujitsu's P-STAFF COBOL development environment.
- Testing and continued development of the overall application that is run under the initial control of Visual Basic if it is a GUI application.

It is worth noting that both Visual Basic and Fujitsu COBOL offer extensive dynamic development environments, making it easy to move between the two when creating and testing applications.

Fujitsu COBOL includes the P-STAFF project oriented development environment, from which you perform work on the COBOL side of the application.

Additionally, both environments provide extensive development support tools to aid the process.

Fujitsu COBOL standalone .EXE (executables) or .DLL files (dynamic link libraries). DLL files do not have to be physically linked into a Visual Basic application and may be called dynamically at run time by Visual Basic.

Doing away with the requirement to link all of this together allows the developer to work off-line on either end of the application. That is, the developer may work on and test the user

interface under Visual Basic, or he may work on the COBOL component(s) in Fujitsu COBOL's P-STAFF without having first gone through Visual Basic.

Because of the dynamic nature of the two development environments, you do not necessarily have to follow the above noted steps in order. You could first develop and test the COBOL components and then design the user interface in Visual Basic. The key point is that when it comes time to execute the overall application, you should have the interface and the COBOL components available.

Chapter 2. Installation

Installing the Fujitsu COBOL 3.0 program is a very simple and straightforward process. This chapter covers the following topics:

- Requirements
- Running the setup program
- Product registration
- Completing the installation
- Where to get help

Requirements

The Fujitsu COBOL 3.0 program is a full 32-bit product. It requires that you install it under either the Windows 95 or Windows NT operating Systems.

Minimum requirements for Fujitsu COBOL 3.0 are as follows:

- 80386 with a 80387 coprocessor or Higher CPU
- 16 MB or higher RAM
- 21 MB or higher Disk Space

The recommended configuration for Fujitsu COBOL 3.0 is as follows:

- High End 80486 or any Pentium CPU
- 32 MB RAM
- 65 MB Disk Space
- 100 MB Disk Space with the CD book

Note: The book can be viewed from the CD.

Running the Setup Program

Insert Fujitsu COBOL 3.0 CD-ROM into your CD-ROM drive and execute the SETUP program in the root directory in one of the following ways:

- From the Windows Explorer (Windows 95 and Windows NT version 4.0 or later) or Windows File Manager (Windows NT version 3.51 or earlier), double click on the icon representing your CD-ROM drive, and then double click on the SETUP.EXE program.
- Click on the Start button (Windows 95 or Windows NT 4.0 or later) or from the File menu in Program Manager (Windows NT version 3.51 or earlier), and select Run and enter the appropriate directory where Fujitsu COBOL 3.0 is located. For example:

  ```
  d:\COBOL32\setup.exe
  ```

  (where “d:” represents the drive letter assigned to your CD-ROM drive.)

  Then click on the OK button.

The following window appears:

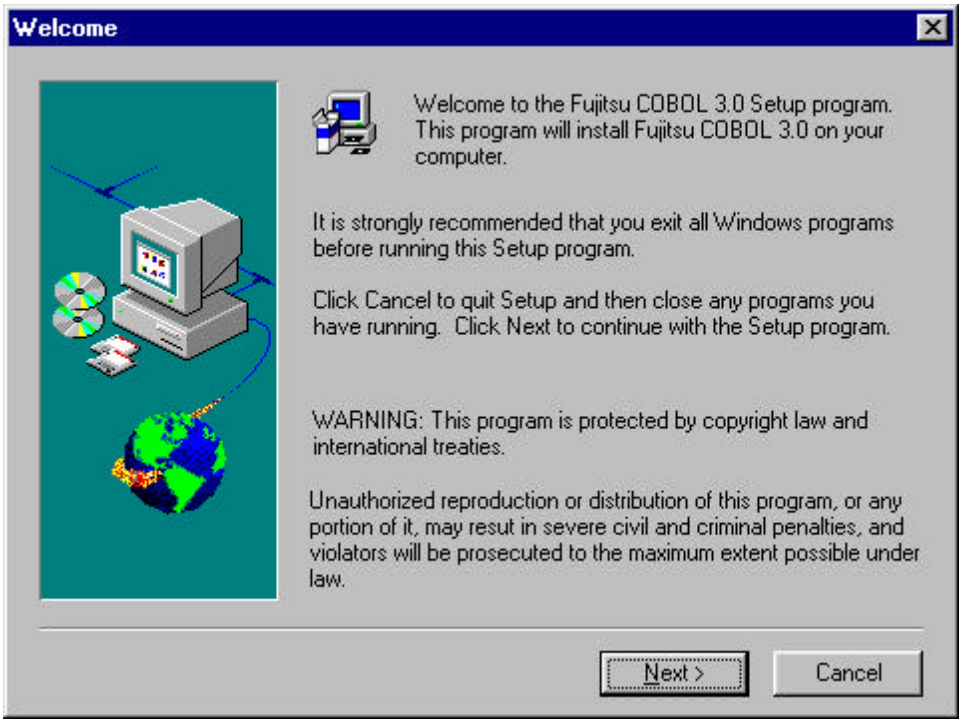


Figure 1. The Fujitsu COBOL 3.0 Welcome window

After reading the text in the window, click on the Next button, or click on the Cancel button if you wish to abort the installation.

The Fujitsu Program License Terms window appears. Click on the Yes button if you accept the terms of the license agreement. Otherwise click on the No button and you will exit the setup program.

The next window to appear may require you to enter a product serial number. The product serial number is located on the back of the product envelope.

The next window asks you to choose which options you would like to install. Click on the options you would like to install and then click on the Next button.

The following window appears:

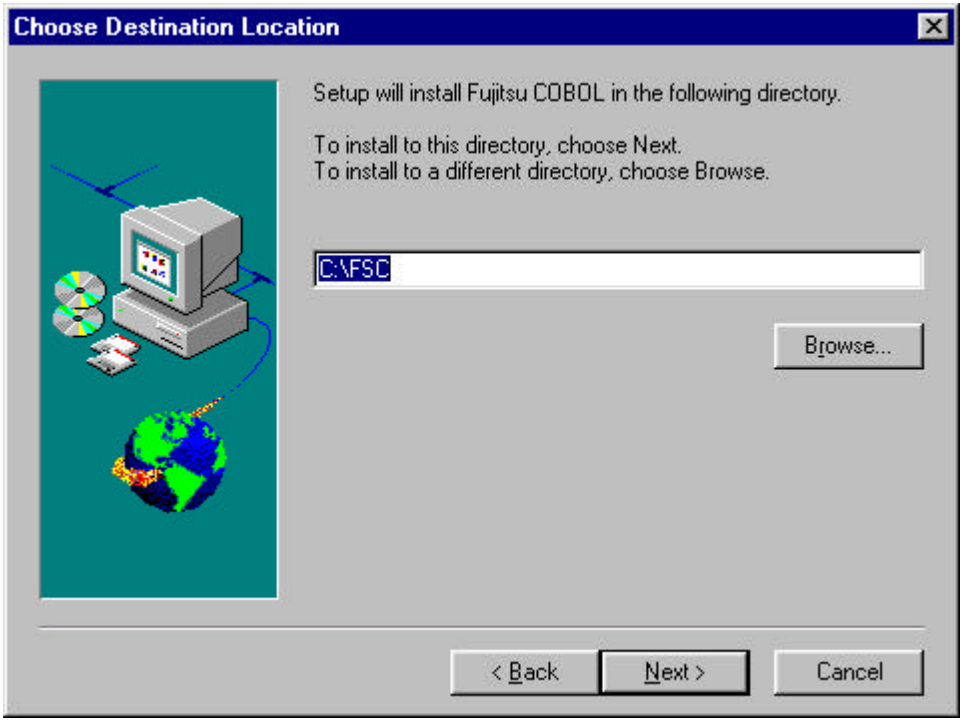


Figure 2. Choose Destination Location for the Fujitsu COBOL Tools window

If you do not wish the product to be installed in the default location (C:\FSC), click on the Browse button and change the installation drive and/or path in the Choose Directory dialog box.

You will be asked if you want to create a new directory if the path you enter into the Path dialog box does not exist.

Click on the Next button to continue with the installation.

The following window appears (note that the list box under "Existing Folders" will vary for each machine, depending on prior software installed):

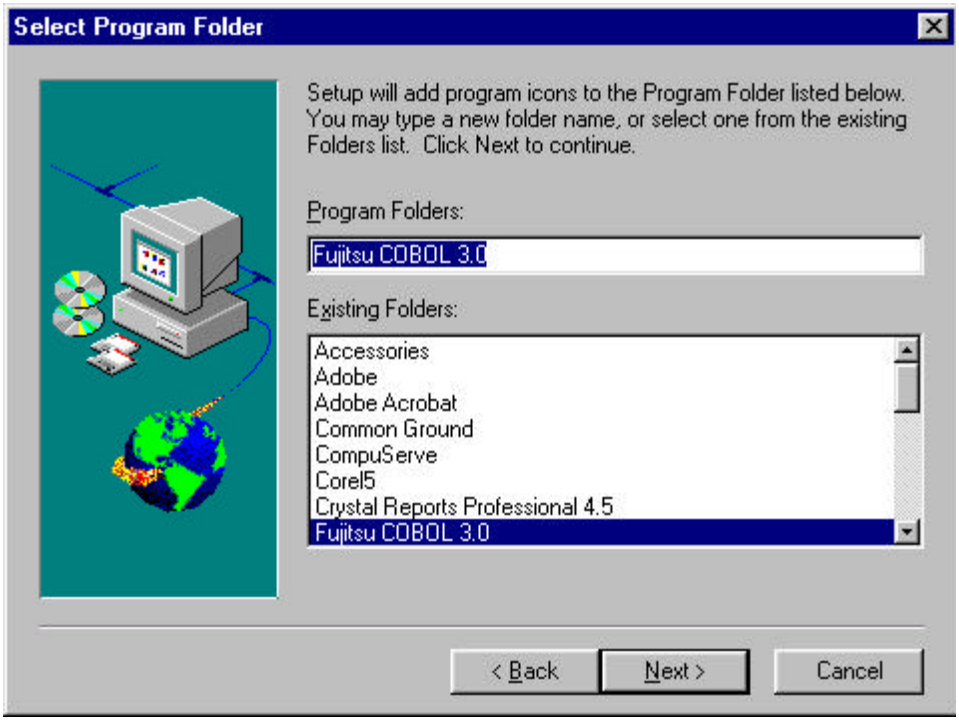


Figure 3. Select Program Folder window

This window allows you to specify which program group folder will contain the Fujitsu COBOL 3.0 Programs. It is recommended that you create a new folder for Fujitsu COBOL, which is the default.

Click on the Next button to continue the installation.

The following window appears on the free version of Fujitsu COBOL 3.0:

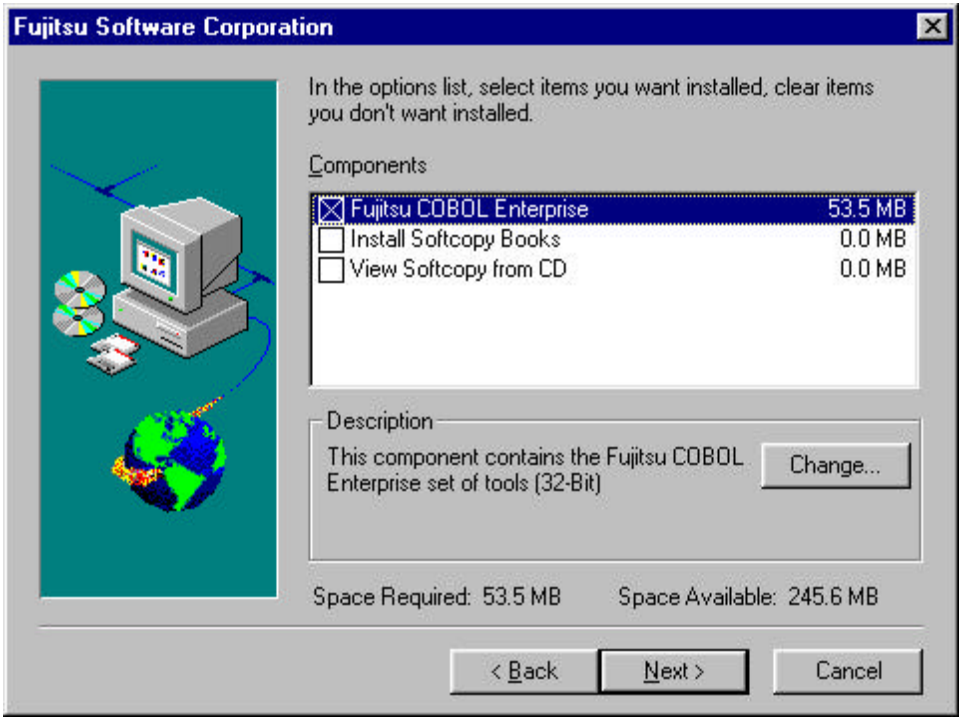


Figure 4. Fujitsu Software Corporation installation window

This window allows you to specify individual components, which may differ depending on the COBOL 3.0 product being used for installation,. and also calculates the required and available disk space. If you do not have enough disk space available, you may click on the Back button and go back and select a different location if available or free up some disk space.

If you are unable to free up enough disk space, you may deselect one or more components in the window to reduce the amount of disk space required. One potential choice would be the Fujitsu COBOL Books, as they may be accessed directly from the CD-

ROM drive when needed, if they are not installed on your hard drive.

Click on the Next button to continue the installation**.** The following window appears:

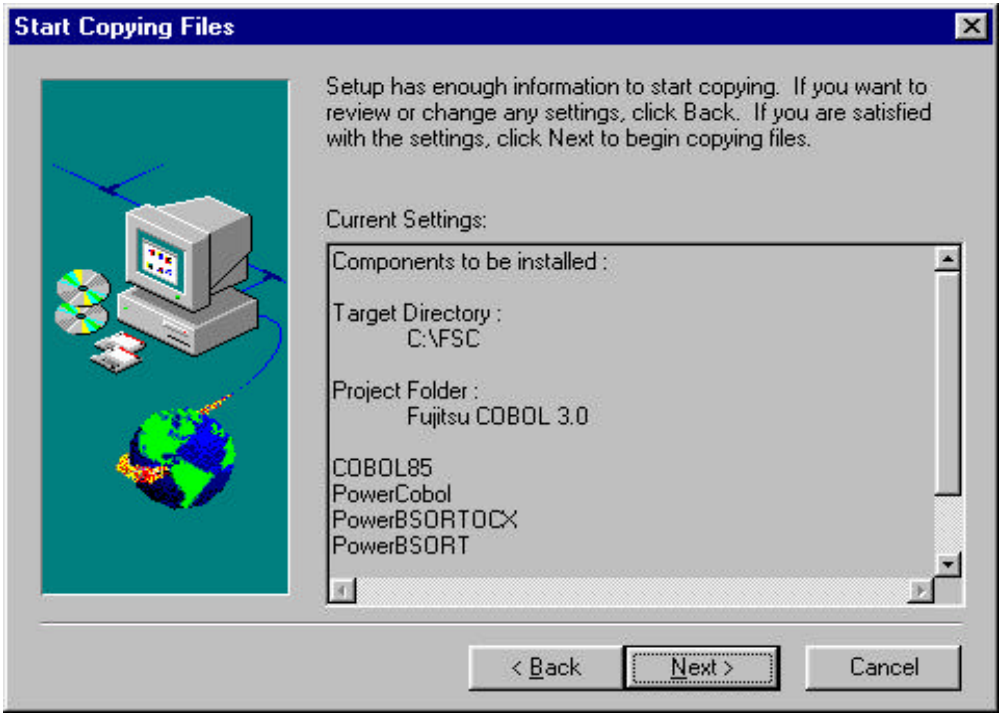


Figure 3. Start Copying Files window

This window presents all of the current installation options you have selected. If you wish to change something, simply click on the Back button until you come to the appropriate place to enter the change.

If you are satisfied with the current options, click on Next to begin part 1 of the installation. A status bar appears and the Setup program will begin copying files from the CD-ROM to your hard disk. Following installation, the product registration screen appears.

Product Registration

It is important that you register your product for the following reasons:

- To assure you are registered to receive technical support. Registered users will receive **FREE** technical support for 60 days (from the first technical support instance). The fastest way to obtain technical support is to send e-mail to **support@adtools.com.**
- To assure that you receive priority notification of new product releases (registered users will receive new product and upgrade information).
- To assure that you may be sent information on the latest improvements (fixes, new versions and related topics).
- To assure that you receive any automatic updates you may be due.

The registration process begins automatically near the end of the setup process.

For optimal use of the registration process, you should have a modem available that is connected to a phone line or have a networked connection to the Internet before you begin.

It is also possible to register via fax or U.S. mail if you do not have a modem or Internet connection.

At the start of the registration process, the following window appears:

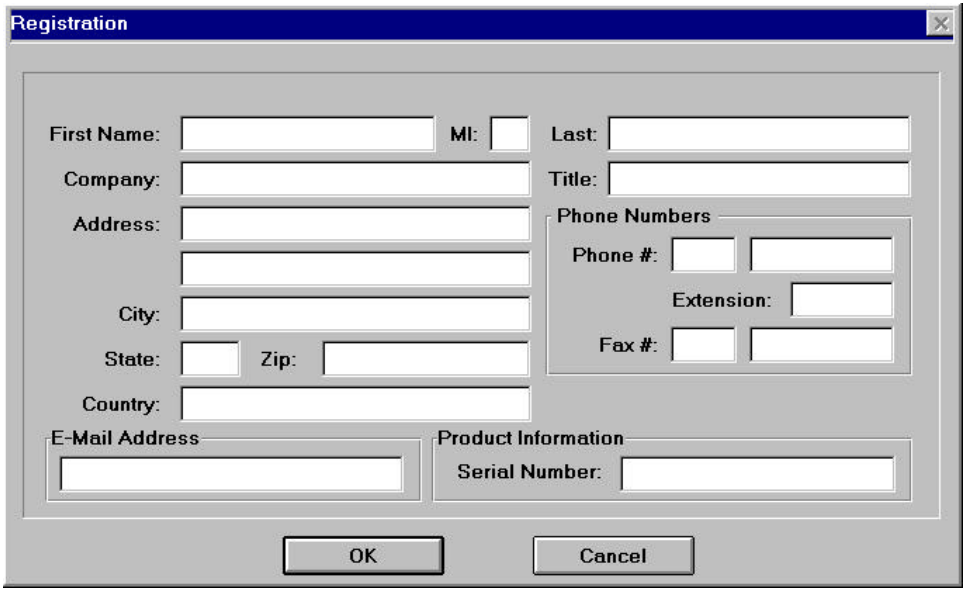


Figure 4. Initial Registration window

Please fill in as much of the information as is applicable. Your product serial number will be partially displayed in the lower right hand corner of the window. Fill out the remaining portion using the complete serial number found on the product envelope.

Once you have completed filling in the information, click on the OK button to continue with the registration process.

The following window appears:

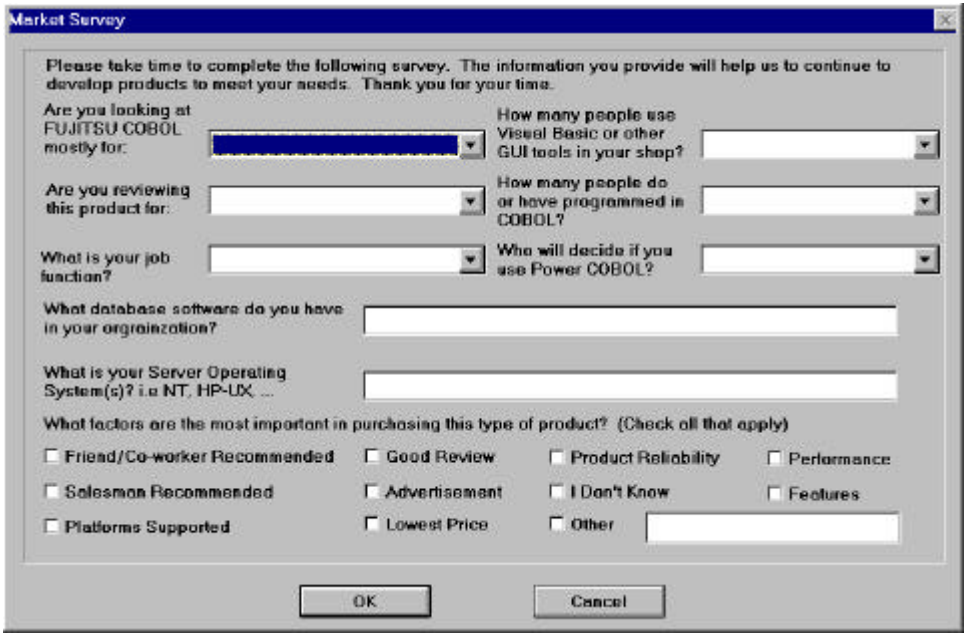


Figure 5. Market Survey window

Please fill out the applicable information in this window and click on the OK button when you are finished.

A window appears asking you to select your current Country. The default value is the United States. Please select your country and click on the OK button to continue the registration process.

The registration process will check to see if you have a modem and phone line installed so that it may dial a toll free number at Fujitsu to upload the registration information automatically. The following window appears, as the registration process verifies your modem:

Figure 6. Modem verification window

If a valid modem and phone connection are found, the registration process will continue automatically.

If a modem is successfully detected in your machine, a telephone setup window will appear:

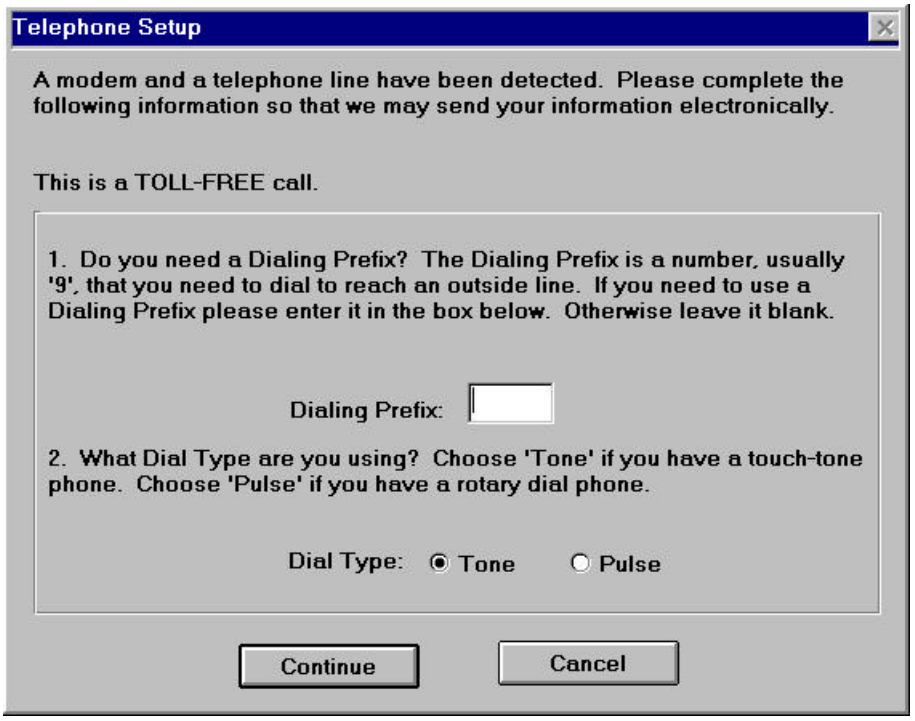


Figure 7. Telephone Setup window

If the phone line to which your modem is connected requires a dialing prefix to reach an outside line (e.g. the requirement to dial a “9” first before getting a dial tone for an outside line), please enter it into appropriate field in the window.

Click on the Continue button to complete the registration process. The modem should then be accessed to dial the toll free number at Fujitsu and the data you have entered for registration will be automatically uploaded and registered.

Once the registration data has been successfully uploaded, your modem should be disconnected, and the following window will appear:

Figure 8. Registration verification window

Click on the OK Button to continue.

If the check for a modem and phone line fails, the following window appears:

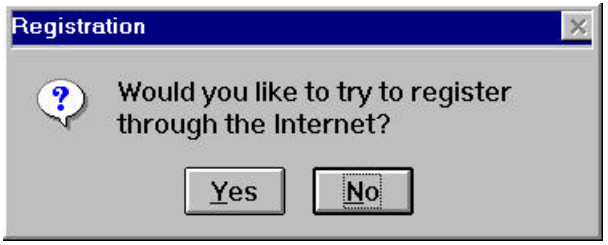


Figure 9. Internet Registration window

Internet registration will take place in the form of an E-Mail sent to the Fujitsu COBOL Web site. If you click on the Yes button, your machine will be checked for a valid Internet connection and the E-Mail will be sent.

If any of the following events takes place during the registration process:

- Program is unable to connect via modem to Fujitsu
- Program is unable to verify Internet Mail connection
- You select **"**No**"** in response to the question directly above regarding registering through the Internet.

The following window appears:

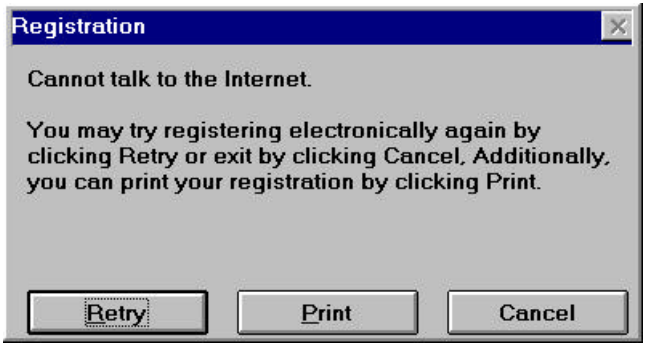


Figure 10. Internet connection failure window

You may either retry the operation, or you may select the Print button to print out the registration information you have entered. Once printed, you may either fax or mail this information into Fujitsu to complete your registration.

Completing the Installation

Once the setup program has completed copying files successfully and you have gone through the registration process, a dialog box will appear to inform you that your autoexec.bat file has been updated. Click on the OK button. The following window appears:

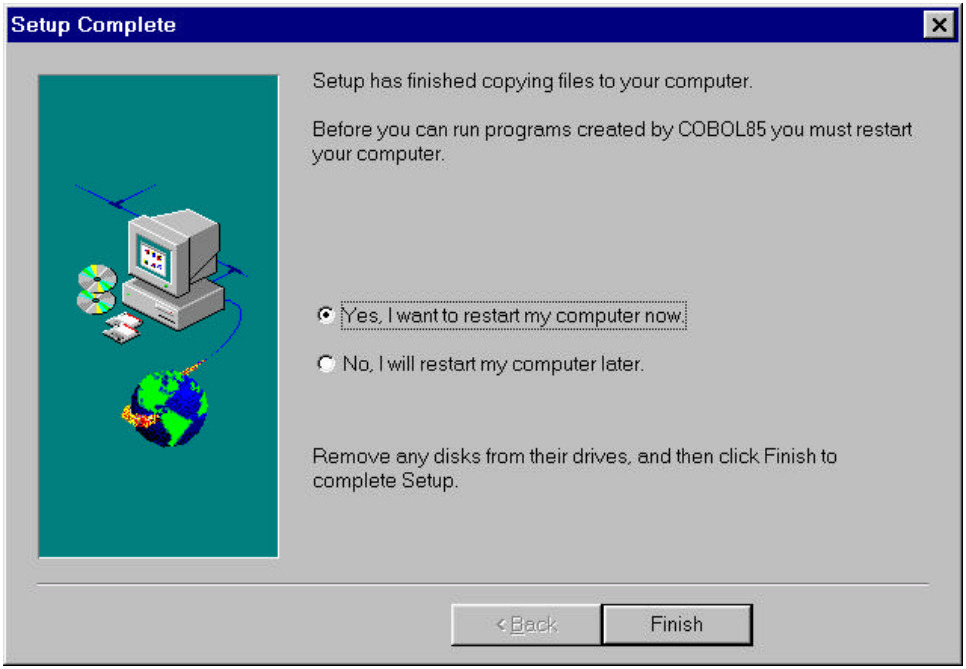


Figure 11. Setup Complete window

This will complete the installation and setup process. Before you click on the Finish button to close the setup program and to re-boot your computer, ensure you have closed all other applications.

If for some reason you choose not to re-boot your computer, please remember to perform a re-boot sometime before attempting to utilize the Fujitsu COBOL 3.0 product.

Note: It is *very* important to review the README files that are shipped with the Fujitsu COBOL 3.0 program. To access the README files under WINDOWS 95 and NT 4.0, click on the Start button, then Programs. Click on the Fujitsu COBOL 3.0 folder. Click on either COBOL Family Readme or COBOL Readme 32.

Where to Get Help

Fujitsu provides world class technical support for Fujitsu COBOL as noted below.

Please note one important point first. If you have a question or problem regarding the usage of Visual Basic (if you choose to utilize it to design your user interface), you should contact Microsoft's technical support department for Visual Basic. Refer to the Visual Basic documentation for information on contacting Microsoft.

If you have a question of problem regarding COBOL, its development environment or the supporting tools, please contact Fujitsu as noted below. **Note**: E-mail is the fastest method.

We sincerely hope that you enjoy Fujitsu COBOL!

Fujitsu COBOL Support Contact Information

Sales:	**(800) 545-6774**
	(408) 428-0300
Tech Support:	**(408) 428-0500**
Registration Fax:	**(408) 428-0600**
E-Mail:	**support@adtools.com**
Web Site:	**www.adtools.com**
Address:	**Fujitsu Software Corp.**
	Attn: Language Products Group
	3055 Orchard Drive
	San Jose, CA. 95134-2022

Chapter 3. A Quick Tour

This chapter takes you on a quick tour of the Fujitsu COBOL development environment.

Because you may produce full fledged COBOL applications using this environment, this chapter will concentrate solely on developing COBOL applications, and will not discuss Visual Basic. Refer to Chapter 4, “Creating Your First Visual Basic COBOL Application” for details on developing Visual Basic COBOL applications.

Fujitsu COBOL ships with several sample COBOL applications which cover a wide array of Fujitsu COBOL functions. Refer to Appendix A for a description of the included sample applications.

The COBOL Development Environment

PROGRAMMING-STAFF (P-STAFF) is an integrated 32 bit development environment that provides project level management for developing COBOL applications. P-STAFF includes:

- A sophisticated, yet easy to use source code editor
- A full ANSI 85 and ANSI 74 compliant 32 bit COBOL compiler
- A 32 bit linker
- A sophisticated COBOL source debugger
- Execution facilities
- Project management and Make facilities

In addition to the above noted tools for COBOL program development, P-STAFF also provides access to Fujitsu's integrated file management facility. This facility provides a number of services for managing COBOL data files of all types, including indexed files.

File management utilities are integrated together and include:

- Full data file editing - including adding, updating and deleting records
- Data file conversion (e.g. sequential to indexed)
- Data file printing
- Data file sorting
- Data file recovery of damaged files
- Data file reorganization

You will be introduced to this development environment by working with one of the sample applications included.

Working With A Sample Application

The sample application you are about to work with is a relatively simple single program, which reads in a sequential file and outputs an indexed file.

We will work on this application first because it creates an indexed file which we will then edit utilizing the data file editor.

Because this is a relatively simple one program application, we will not be utilizing the sophisticated project management facilities within P-STAFF. We will look at these later in this chapter.

The first step is to initialize the P-STAFF programming environment. Find the Fujitsu COBOL 3.0 directory on your machine.

Under WINDOWS 95 and NT 4.0, click on the Start button, then Programs. Click on the Fujitsu COBOL 3.0 folder (directory) and then click on Programming Staff 32.

Under Windows NT 3.51, open the Fujitsu COBOL container (directory) and double click on Programming Staff 32**.**

The following window appears:

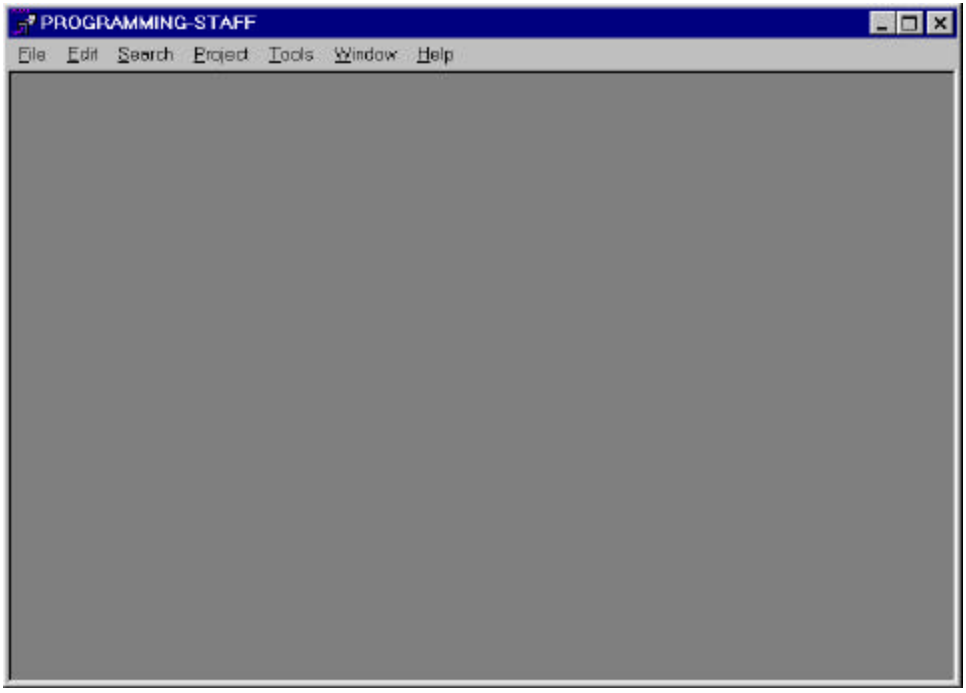


Figure 14. The PROGRAMMING-STAFF (P-STAFF) window

The installation process should have installed the sample programs. They can be found under the same directory that P-STAFF was installed in (typically C:\FSC\PCOBOL32\), in the Samples sub-directory.

Select Open from the File menu. Navigate through and find the above mentioned samples directory and then look within the SAMPLE2 sub-directory for the SAMPLE2.COB file. The fully qualified path name of this file (if you did not alter the default installation directory) is:

```
c:\FSC\PCOBOL32\SAMPLES\SAMPLE2\SAMPLE2.COB
```

Click on the OK button to load this file into the editor.

The following window appears:

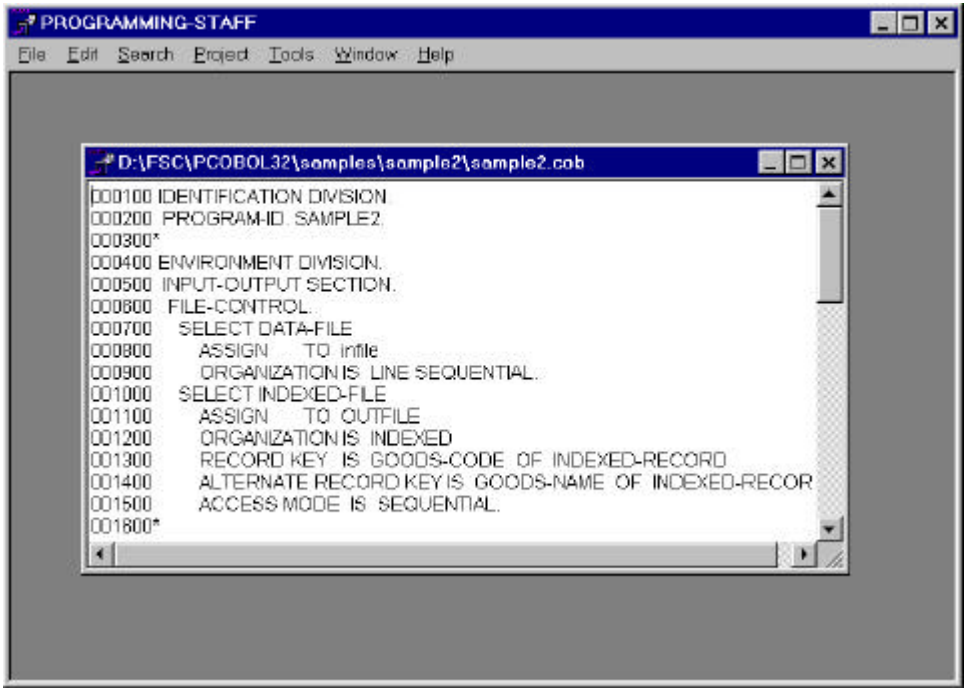


Figure 15. P-STAFF edit window

P-STAFF is a fully functional MDI (Multiple Document Interface) application. This means that within its main parent window, you can have any number of child windows active. This allows you to edit two files at the same time and to use the clipboard functions to copy text back and forth, for example. You may resize and re-position these windows however you like.

The SAMPLE2.COB program is a straight forward file conversion program. You are simply going to compile, link and execute it under P-STAFF. The result of this process will be an executable file (.EXE) that can be executed outside of the P-STAFF environment as well.

Compiling Your Program

To compile the Sample 2 program, select WINCOB [Compile] from the Tools menu. The following window appears:

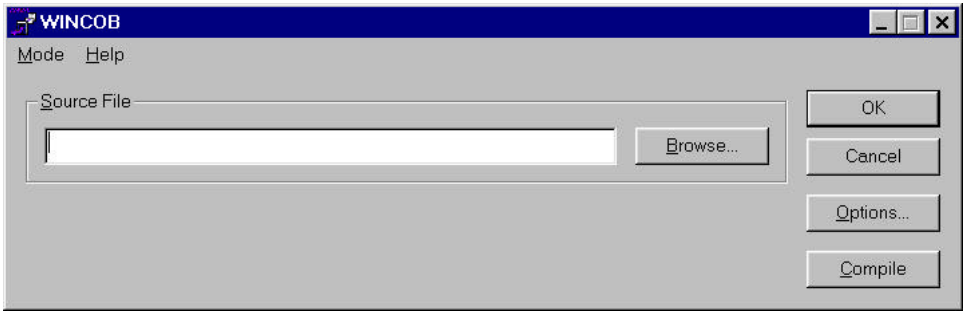


Figure 16. The WINCOB window

Use the Browse button to locate and specify the fully qualified path name of the SAMPLE2.COB program. Then click on the Options button. You need to set one compiler directive to tell the system that this is the main program (as opposed to being a sub-program of a larger project). The following dialog box appears:

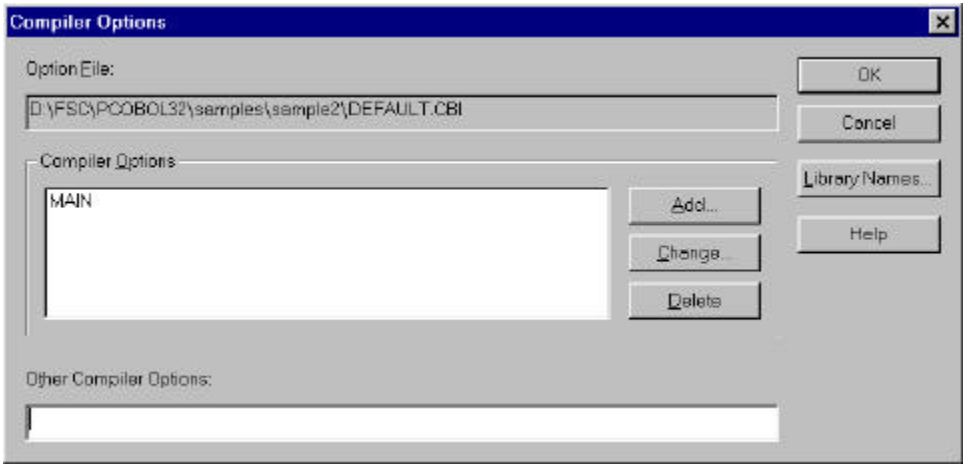


Figure 17. The Compiler Options dialog box

If the MAIN compiler option is not listed in the Compiler Options list box, click on the Add button and select MAIN from the pop-up list of available compiler options. Click on the Add button to close the (Add) Compiler Options dialog box. Select Compile Program as Main Program in the (Details) Compiler Option dialog box and then click on the OK button.

Once you have the MAIN option listed in the Compiler Options list box, click on the Cancel button. This will take you back to the WINCOB window.

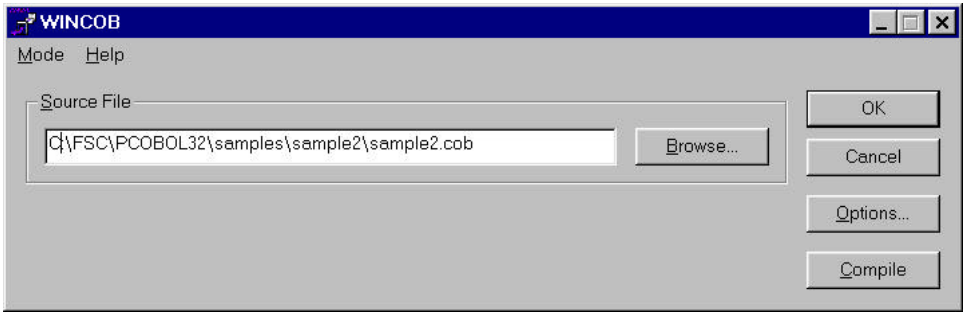


Figure 18. The WINCOB window

Click on the OK button to compile the program. You will see a countdown clock window which illustrates compile time progress. The program should compile without errors, and an message window will appear as follows:

Figure 19. Message window

You may close this window by clicking the close (X) button in the upper right hand corner.

For further details on using WINCOB, refer to the “COBOL85 User’s Guide.”

Linking Your Program

You are now ready to link the object module which the compiler produced to create an executable program. From the P-STAFF window, select WINLINK [Linking Files] from the Tools menu. The following window appears:

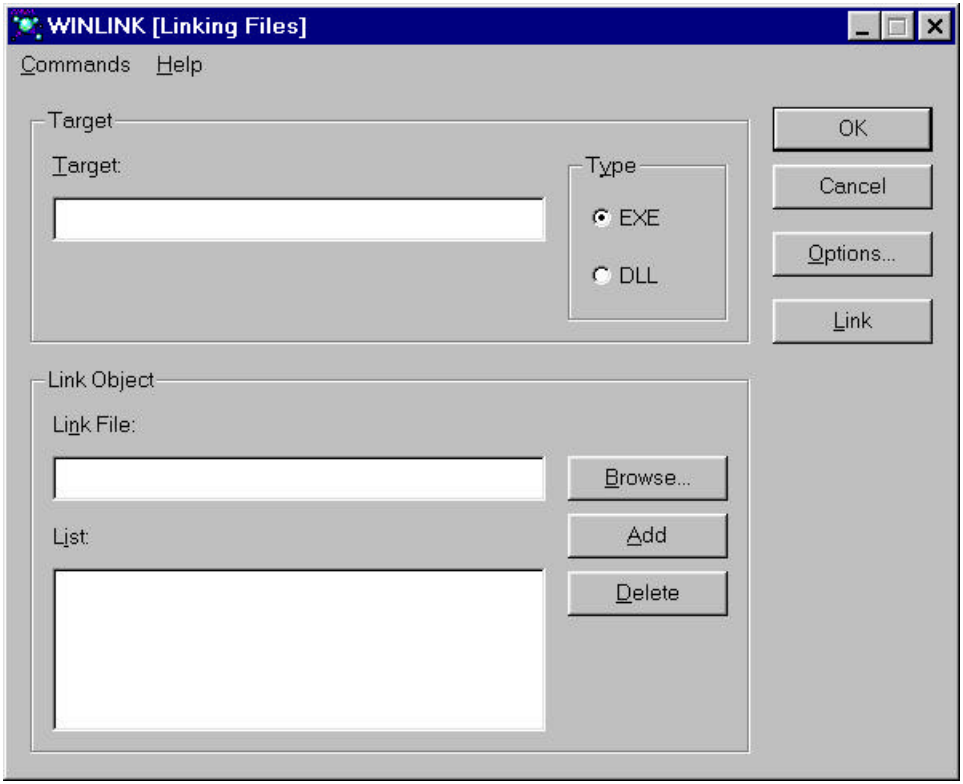


Figure 20. The WINLINK [Linking Files] window

Under the Link Object frame of this window, click on the Browse button, then locate and select the SAMPLE2.OBJ file. Once it is listed under Link File, click on the Add button to add it to the list of objects to be linked. A target name is displayed when the

object file is added. The WINLINK window should appear as follows:

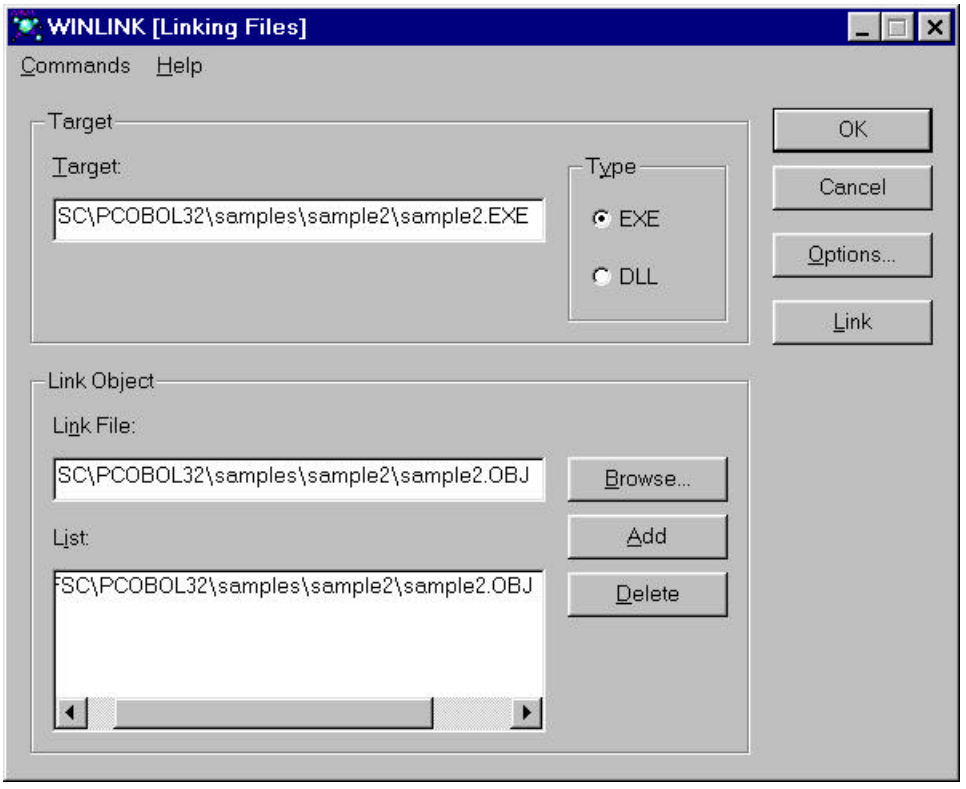


Figure 21. The WINLINK [Linking Files] window

Make sure that EXE is checked to ensure the linker produces a stand-alone executable file.

Cick on the OK button to begin linking the program and creating an executable file.

The linker is a DOS command line utility so the output will be displayed in an MS-DOS window which you should close when you are satisfied with the results.

The MS-DOS output window should look something like the following when linking is completed:

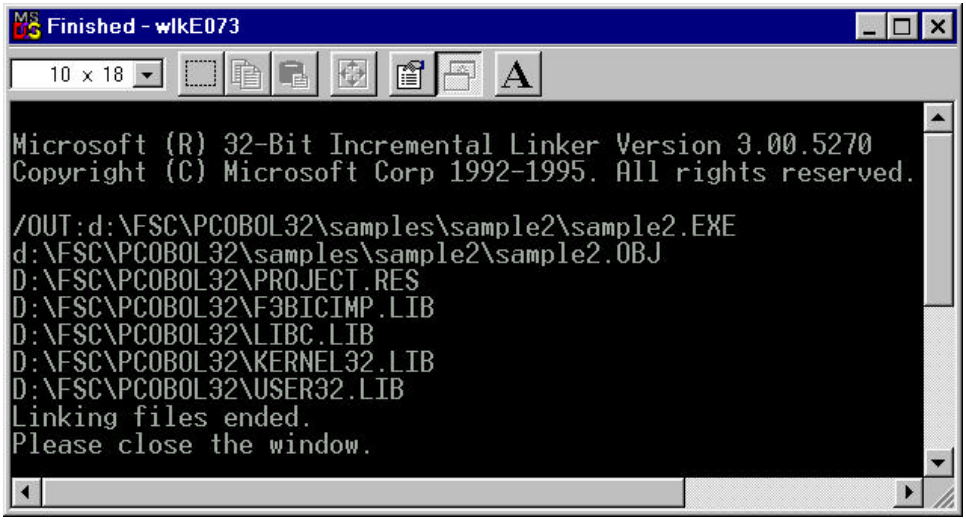


Figure 22. MS-DOS window showing output results

P-STAFF automatically creates a make file to provide an intelligent link process.

You do not have to spend time trying to determine which libraries need to be linked or where these libraries are located. Nor did you have to specify any link options. P-STAFF performs these tasks automatically as well.

If you have in the past utilized other vendor's COBOL development environments, you may now begin to appreciate P-STAFF's application management capabilities.

You should now have an executable program ready to run.

For further details on using WINLINK, refer to the "COBOL85 User's Guide."

Executing Your Program

You are now ready to execute the SAMPLE2.EXE program. You will utilize the WINEXEC facility to accomplish this.

WINEXEC allows you to execute programs from within P-STAFF. These programs may typically be executed directly from Windows itself, or from an MS-DOS command line under Windows.

However, WINEXEC offers a run-time environment management facility that greatly simplifies the execution process.

The sample program utilizes an input file and creates an output file. In the COBOL SELECT statements in the SAMPLE2.COB program you will find:

```
ASSIGN TO INFILE
```

and

```
ASSIGN TO OUTFILE
```

In the SAMPLE2 sub-directory, you will find a file named DATAFILE. We will utilize the environment run-time management facility within WINEXEC to equate the file names properly (much like DD JCL Statements in the mainframe environment, and like SET statements in the DOS and Windows environments).

In the P-STAFF window select WINEXEC [Execute] from the Tools menu.

The WINEXEC window appears. Click on the Browse button and select the SAMPLE2.EXE file. The WINEXEC window should display the following information:

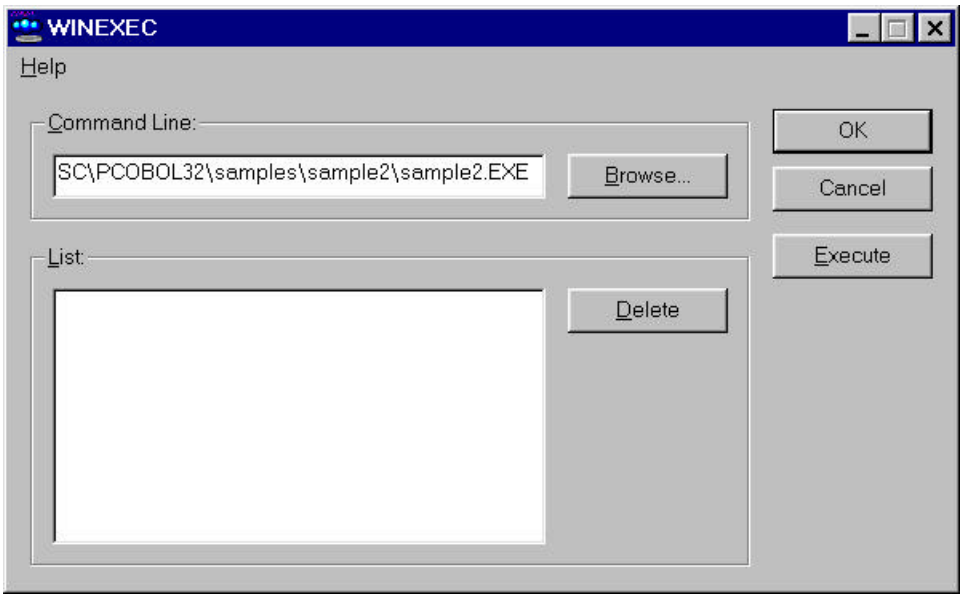


Figure 23. The WINEXEC window

Click on the OK button. The Run-time Environment Setup window appears. This window allows you to specify a wide number of run-time options for this specific application. You may use this facility to create customized run-time settings for any application, which can be saved and shipped along with the application.

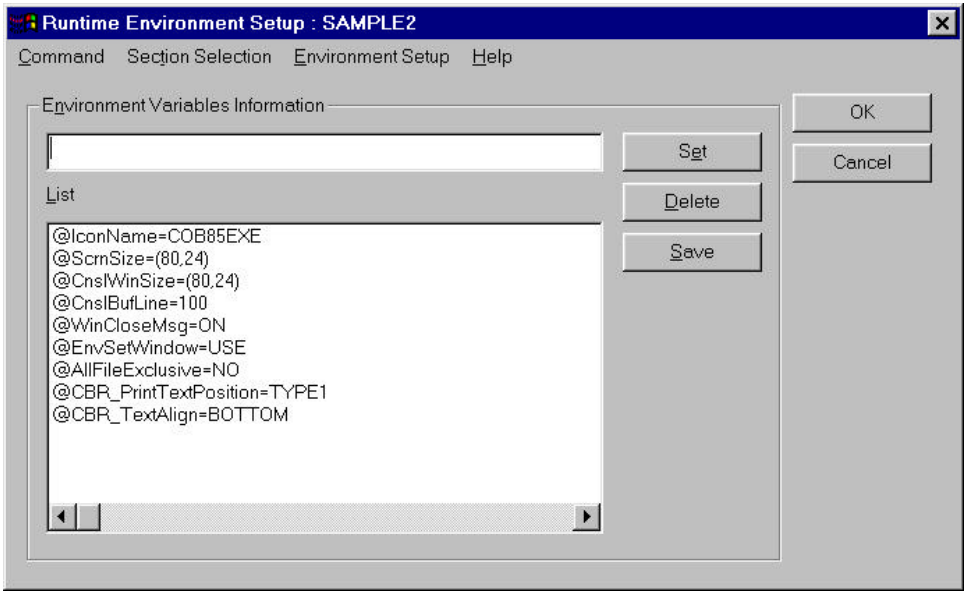


Figure 24. The Run-time Environment Setup window

You are going to add two new settings to the list currently displayed. The first will be for the input file. You need to equate "INFILE" to "DATAFILE".

To accomplish this, simply type:

INFILE=DATAFILE

under the Environment Variables Information field, and click on the Set button. The new entry will appear at the bottom of the list. Now do the same for the output file by typing:

OUTFILE=MASTER

and clicking on the Set button. This entry will appear in the bottom of the list box.

If you plan on executing this program again, you may want to click on the Save button to save the specific environment information. Otherwise, you must go through this process again each time you execute the program.

Click on the OK button to execute your program.

Because you are executing a batch-style program that does not perform any form of ACCEPT/DISPLAY (or other screen I/O), this program will execute in the background.

This means that you will not be presented with any additional information (such as an execution window) unless the program encounters a problem.

The output indexed file created by the program is called MASTER and it is placed in the SAMPLE2 sub-directory.

You are now ready to examine the powerful data file management facilities contained in Fujitsu COBOL. In the next section, we will utilize the index file we just created.

For further details on using WINEXEC, refer to the “COBOL85 User’s Guide.”

Using the Data File Management Facilities

Fujitsu COBOL comes with a powerful set of data file management facilities. We will utilize the data files from the SAMPLE2 sample program which you worked with in the previous section. The indexed file you are going to use here should have been created by the SAMPLE2 program as noted previously.

From the P-STAFF window, select COBFUT32 (the COBOL85 File Utility) from the Tools menu. The following window appears:

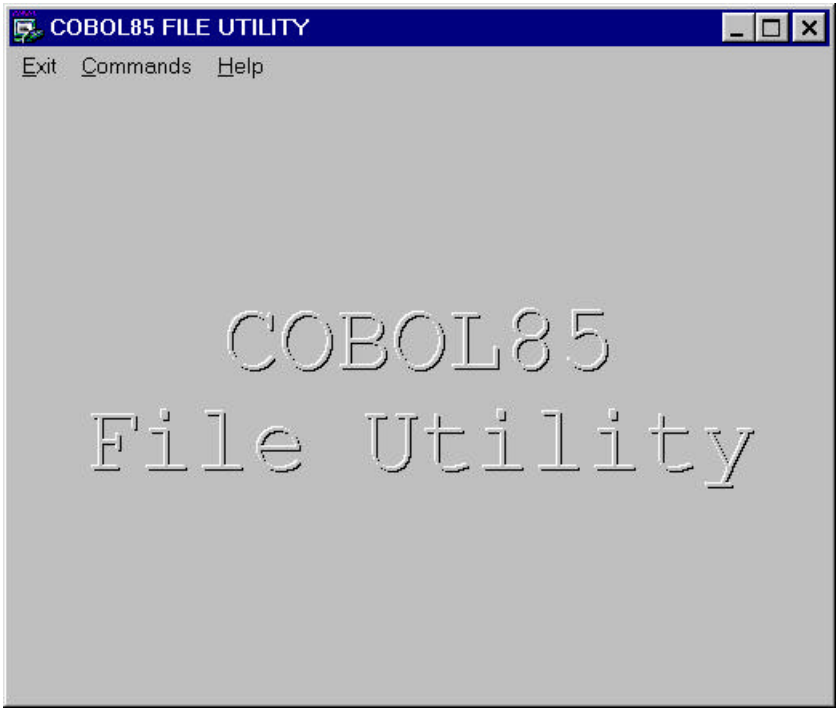


Figure 25. The COBOL85 File Utility

Select the Commands menu to access the COBOL85 File Utility commands.

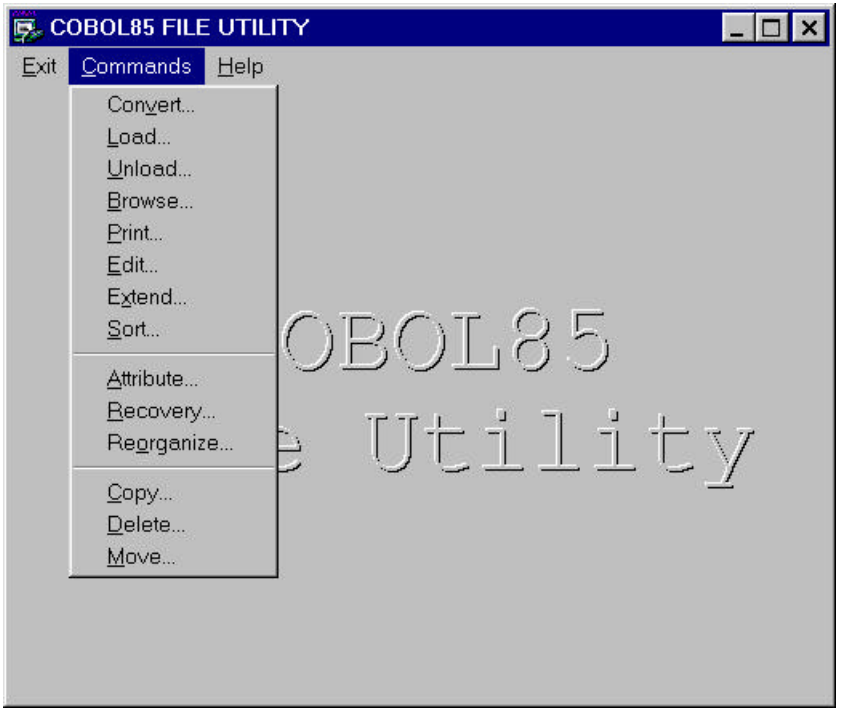


Figure 26. The Commands menu in the COBOL85 File Utility

The options listed in the Commands menu are as follows:

Convert

Converts text files to variable length, record sequential files and converts variable length, record sequential files to text files.

Load

Creates a record sequential file, relative file, or indexed file from a variable length, record sequential file. Using the EXTEND option, you can add records to any of these files using a variable length, record sequential file.

Unload

Creates a variable length, record sequential file from a record sequential file, relative file, or indexed file.

Browse

Displays each record's contents by character and hexadecimal expression.

Print

Prints record contents by character and hexadecimal expression. Non-alphanumeric characters (codes outside the range 0x20 to 0x7E) are printed as a period (".") in the character area. You can specify a range of records to be printed.

Edit

Allows you to browse, update, insert, and delete records.

Extend

Allows you to add records to an existing file.

Sort

Sorts the records of a record sequential, line sequential, relative, or indexed file by specified data items, and creates a new variable length, record sequential file.

Attribute

Displays indexed file attribute information (such as record length, record format, and key information).

Recovery

Restores indexed files which were not able to be accessed. Data which cannot be restored is output to the unrecoverable data store file in binary form. The Recovery command rewrites the file before restoring.

Reorganize

Deletes unused blocks of an indexed file and makes the file smaller. When unused blocks of indexed files are deleted, file access performance may be improved.

There are also file copy, delete and move facilities available.

Using the COBOL85 File Utility Editor

Select Edit from the Commands menu. The following window appears:

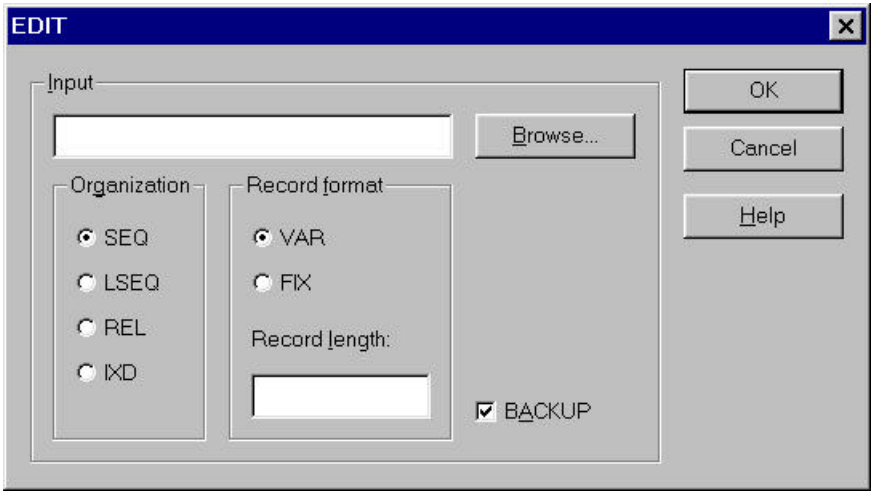


Figure 27. The Edit command window

Click on the Browse button and find the indexed data file (MASTER) you just created in the SAMPLE2 sample program This should be located in the c:\FSC\PCOBOL32\SAMPLES\SAMPLE2\ directory.

Once you have specified the file name, click on the IXD option button under the Organization frame box. This tells the editor that you are about to edit an indexed file. The Backup check box

tells the editor to make a backup copy of the file in case you decide you don't want your changes to take effect.

Click on the OK button to start the editor. Notice that you do not have to specify any record length or key information. The editor determines this automatically.

The editor window will appear and the data file will be automatically loaded. The window should look like:

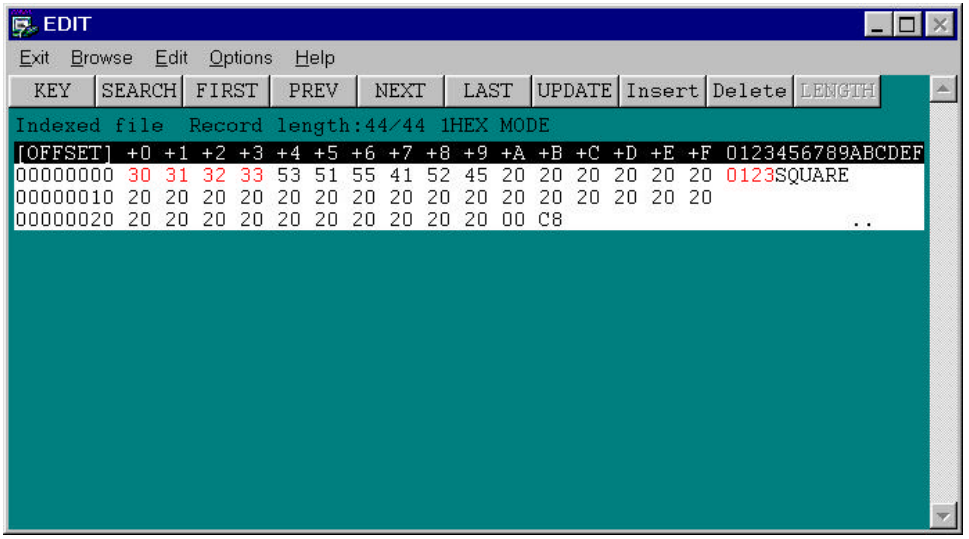


Figure 28. The COBOL85 File Utility editor

If the editor window appears very small, you may change this by selecting Select Font from the Options menu. You may pick a different font style and size and the window will be automatically resized.

Notice the layout of the record, with a Hex dump style format on the left, and the character equivalent on the right. The current key field is highlighted in red. The record is wrapped automatically, and the offset is noted in the far left hand column.

You may use the buttons on the tool bar to move through the file record by record, jump to the first or last record, insert, update,

or delete records, instigate a search, or specify a key. This provides you with greater flexibility in data editing.

A Sample Debug Session

Fujitsu COBOL comes with an extremely powerful native code source debugger.

This debugger actually works on the application's native object code. This is a key feature for serious, production minded application developers.

Historical Problems with other COBOL Debuggers

In the past, other COBOL vendors have delivered development environments which do not debug native object code. Instead, these compilers compile COBOL applications into a pseudo object code, much of which is then interpreted at run-time.

While this approach has had reasonable success for developing applications targeted at other environments, it has presented a litany of problems for PC production minded application developers who wish to field the production COBOL applications on PC platforms.

The past success of the pseudo code-based compilers has largely been for developers who want to use the PC for editing and some level of testing applications which are then destined to be re-compiled with someone else's production COBOL compiler on another target machine.

The best example of this is an IBM Mainframe COBOL application. The mainframe is a slow and expensive environment for development. If a developer can transfer mainframe

applications to the PC for editing, compiling and debugging, a great deal of benefit can be realized.

Thus, there are COBOL development products on the market which contain large legacy run-time systems and are tuned for this sort of development, known as “off-loading”.

These same systems, however, create a myriad of problems for developers who wish to utilize them to create true PC-based production applications.

Pseudo code, while somewhat portable to other platforms, suffers greatly from performance degradation. This comes from the fact that much of it has to be interpreted at run-time by these environments.

Some vendors have attempted to address this problem by supplying native code generators, which attempt to convert this pseudo code into something closer to native object code.

The main problem with this approach is that developers are forced to debug their applications in pseudo code, which the debuggers require. When it comes time to deliver the application, developers then re-compile the pseudo code into some form of native code.

The problem with this approach is that problems may occur wherein something works in pseudo code but does not work in native code. To make matters worse, the same COBOL debugger in these environments typically does not work on native code, and a kind of “Catch 22” situation develops.

A Serious Production Environment

One of the main goals of Fujitsu COBOL was to deliver the world's first serious production-oriented COBOL development environment for Windows-based PC's.

To achieve this goal, Fujitsu developed from the ground up a COBOL compiler that creates native object code directly, with no pseudo or intermediate code step.

The Fujitsu COBOL debugger works directly off of this native object code, so the developer debugs what is targeted for production.

Like most other serious language development environments such as C and C++, Fujitsu COBOL requires a single compiler directive (Test), to produce native object code with additional debugging information within.

Once the application is tested, it is typically re-compiled without the Test option to remove the debugging information. This reduces the size of the object code and increases performance.

As a result, there is one set of native object code, no pseudo or intermediate code, and no run-time interpretation required. More importantly, the developer utilizes a single powerful debugger to debug the application in native object code.

Preparing to Use the Fujitsu COBOL Debugger

In this section, you will take a quick tour of the powerful 32 bit Fujitsu COBOL debugger. You will utilize the same Sample2 application executed earlier in this chapter.

In order to prepare this application for use with the debugger, you will first need to compile it with the appropriate debug compiler directive (Test).

First, ensure that the P-STAFF development environment is active (Execute Programming Staff 32 from the Fujitsu COBOL 3.0 directory). Now follow these steps to prepare the SAMPLE2.COB program for debugging:

1. Select WINCOB [Compile] from the Tools menu to compile the program.
2. Click on the Browse button and locate the SAMPLE2.COB program.
3. Click on the Options button in the WINCOB window. You need to ensure that the compiler options Main and Test have been set in the Compiler Options list box. Click on the Add button and add both options if they are missing.

The dialog box should now appear as follows:

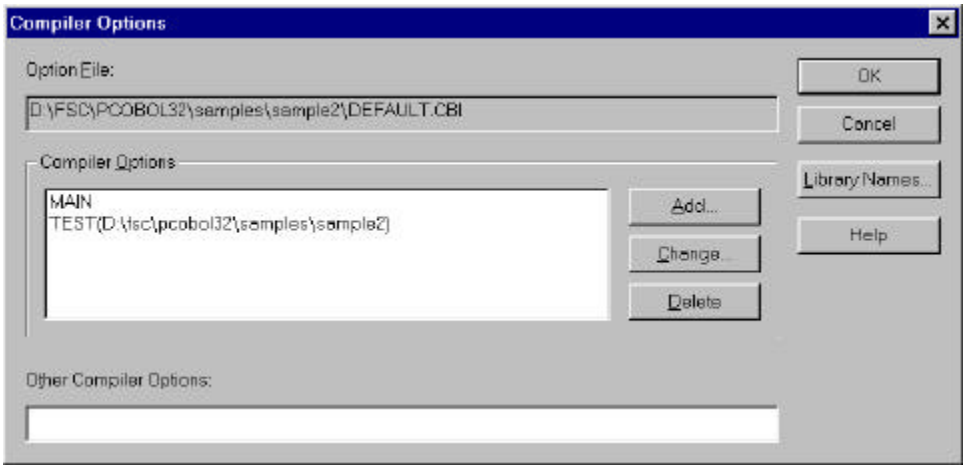


Figure 29. The Compiler Options dialog box with MAIN and TEST specified

4. Click on the OK button. This takes you back to the WINCOB window, which should appear as follows:

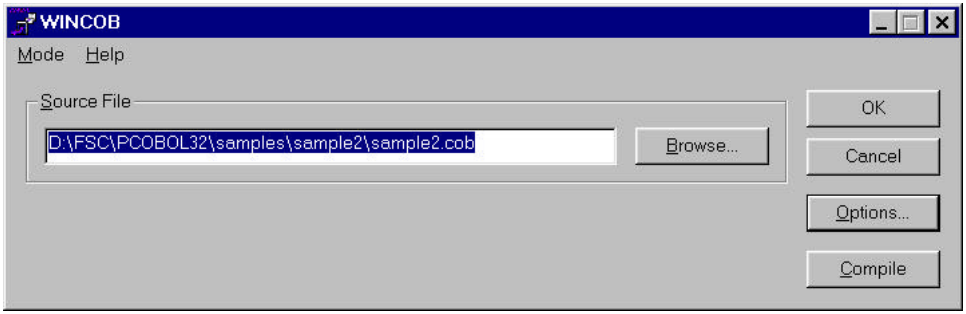


Figure 30. The WINCOB window

5. Click on the OK button. The program will now be compiled and you will see a countdown clock window to indicate compilation progress.

6. The program should compile without errors and you will see the following message window, which you may close by selecting the close icon (X) in the upper right hand corner:

Figure 31. The Message window

7. You now need to link your program with the appropriate debug linker options. From the Tools menu, select WINLINK [Linking Files]. This brings up the WINLINK window.
8. In the WINLINK window, click on the Browse button and find the SAMPLE2**.**OBJ object file. Then click on the Add button to add it to the list of files to link.
9. Next, ensure that the path-qualified SAMPLE2.EXE file is listed in the Target edit box, and that EXE is checked**.**
10. Now click on the Options button. This brings up the Linker Options window. Click on the Debug button. This specifies that you wish to link this application for use with the Debugger. Click on the OK button.

11. This will bring you back to the WINLINK window, which should now appear as follows:

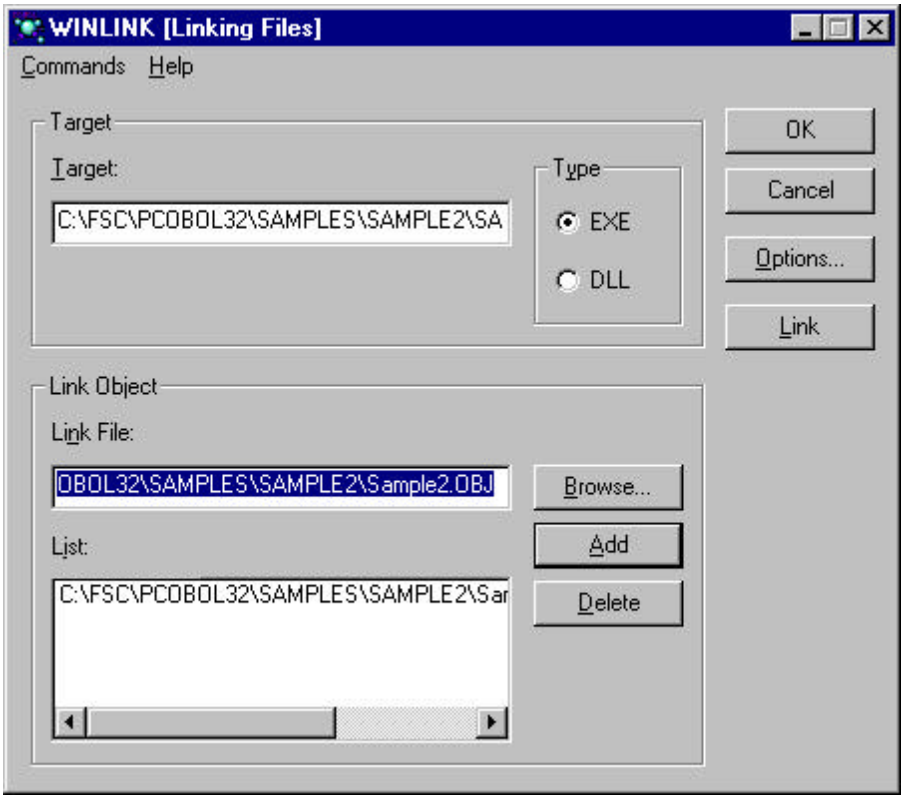


Figure 32. The WINLINK [Linking Files] window

12. Now click on the OK button to link the program for execution with the debugger. You are now ready to debug the program. Close the MS-DOS window showing the link results.

13. From the main P-STAFF window, select WINSVD [Debug] in the Tools menu. The main debug window appears. The size of the window may vary depending on your system resolution and currents defaults. It can be easily resized at will.

Figure 33. The COBOL85 Debugger window

14. Select Start Debugging from the File menu. Make sure the cursor is in the Application field and click on the Browse button. Find and select the SAMPLE2.EXE program and click on the OK button.

    The Run-time Environment window will be displayed. Before you click on the OK button to begin execution under the debugger, ensure that your current environment settings include the file assignments as follows:

    ```
    INFILE=DATAFILE
    ```

    and

    ```
    OUTFILE=MASTER
    ```

    These two settings should be displayed in the List window if you saved them from the prior execution, detailed previously in this chapter. If they are not present, simply type them in one at a time and click on the Set button to add them.

Now click on the OK button to begin execution and load the program into the debugger.

The debugger will load the program and bring up a source code window. The current execution point will be highlighted. The cursor will be positioned on the first executable statement in the program The window should appear as follows, depending on the current window size:

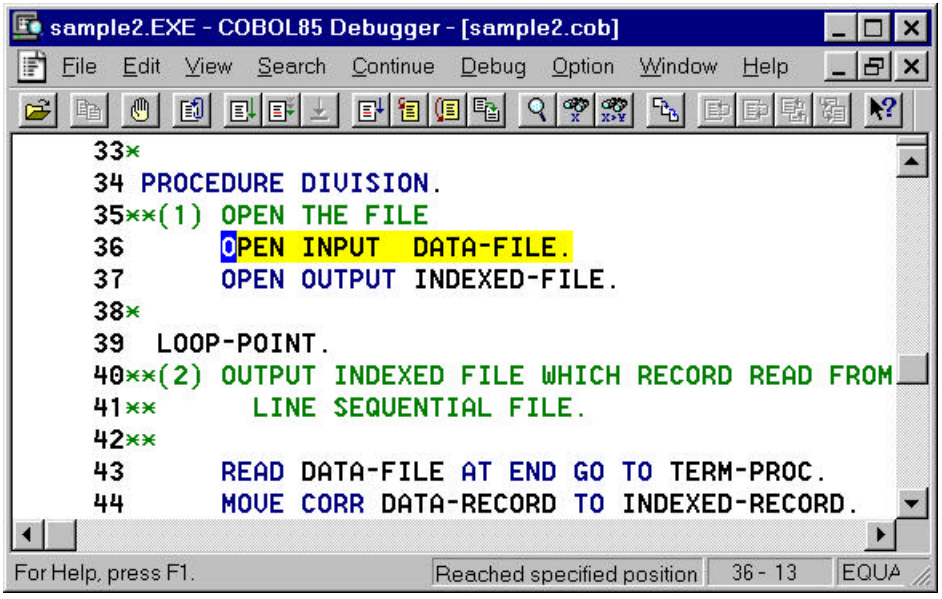


Figure 34. A source code window in the COBOL85 Debugger

You are now ready to explore the powerful Fujitsu COBOL native source code debugger.

In the following section, you will be guided through some of the major features of the debugger. There are a significant number of powerful features that you should additionally read about in the “Fujitsu COBOL Debugging Guide.”

Using the Fujitsu COBOL Debugger

The Fujitsu COBOL debugger offers a large number of significant features for the COBOL developer. You will be taken through some of the more significant features in this section.

Refer to the “Fujitsu COBOL Debugging Guide” for a more comprehensive discussion and usage of the debugger than is found here.

The debugger offers the following commonly found options:

- Ability to step execution line by line through a program
- Ability to set breakpoints and execute at full speed up to those points
- Ability to interrogate, monitor, and to change any data item, including Linkage Section data items

In addition to the above noted standard features you would expect in any debugger, the Fujitsu COBOL debugger offers a wealth of advanced features such as:

- COBOL source code sensitivity. Color coding of various parts of your program takes place automatically to help distinguish COBOL verbs from data items, etc.
- Ability to debug entire application suites
- Ability to set breakpoints within any program in an application suite
- Ability to set conditional breakpoints dependent upon certain conditional events taking place. For example, if you are not sure which specific line to set a breakpoint on, you can instead specify a conditional breakpoint such as “Break when Employee-ID Equals 9999.” This allows you to execute at

machine speed until your program meets the specified condition.

- Execution tracing-When enabled, this feature records the path of the debug session through your application. This assists in identifying logic flow problems, non-executed code, and other problems related to the application flow through your program(s). This feature requires that you turn on the TRACE compiler option.
- Passage counts may be turned on to report how many times specified statements are executed on a given run.
- The debugger will check subscripts automatically for out of range conditions and warn you. This feature requires that you turn on the CHECK compiler option.
- Recording and playback of debug sessions for regression testing. Automatic Debugging actually executes lists of debug commands directly out of a previously created file.
- Split screen debugging provides the ability to view two separate parts of your source code at the same time.
- Ability to monitor which programs of your application are currently loaded into memory, and the ability to easily switch back and forth between them.
- Ability to dynamically allocate Linkage Section items and set them on the fly when debugging sub programs without their parent programs.

The combination of advanced debugging features coupled with the fact that this is a native source code debugger illustrates the serious nature in which Fujitsu COBOL was designed for production applications development.

You will now be guided through some of the basic debugging capabilities. It is assumed that you have previously utilized a debugger. Depending upon your past debugging experience,

you should find this debugger to be quite straightforward and intuitive.

As you go through this section, please feel free to experiment and try any features that interest you.

You will notice that debugger commands may be accessed via menus. Additionally, you will notice a tool bar directly under the menus that appears as follows:

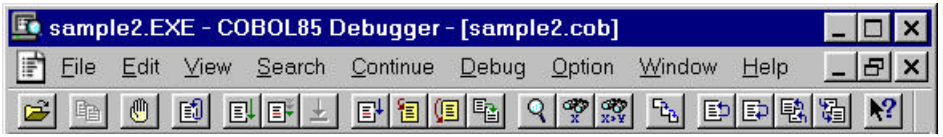


Figure 35. The COBOL85 Debugger tool bar

If the toolbar is not present on your system, simply select Toolbar from the View menu. The tool bar buttons provide an accelerated method of accessing many of the debugger commands that are available via the menus.

If you want to know which command each icon represents, simply move the mouse pointer over the button (do not depress the mouse button, or you will execute the actual command!). You will see a small pop-up text box that explains which command the icon under the mouse pointer represents.

Some of these icons may be grayed out and thus non-selectable, depending upon the current state of your debugging session.

The following is a brief overview of some of the main debugging commands available on the Continue, Debug and Option menus.

Controlling Execution While Debugging

The Continue menu contains some of the execution control commands. Click on the Continue menu to access the commands. We encourage you to try out the commands explained below. If you encounter the end of the program, or a problem, select Re-debug to start afresh.

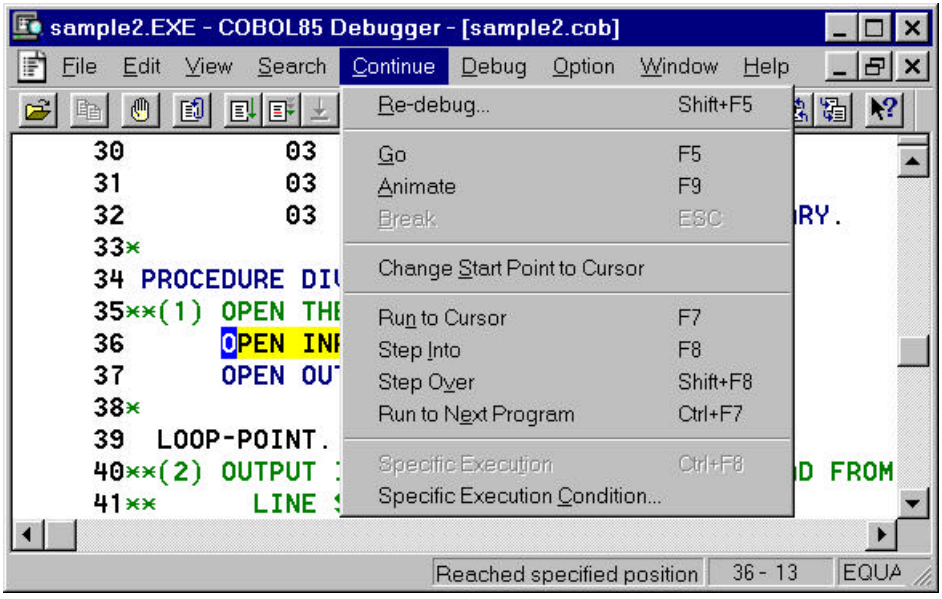


Figure 36. The Continue menu in the COBOL85 Debugger

The following commands are available on the Continue menu:

Re-debug

This command ends the current debug session and begins a new one by restarting execution.

Go

This command starts executing at machine speed, and stops only when a break point is encountered, an error takes place, or the application completes execution. You may also select Break to suspend execution.

Animate

This command causes the debugger to execute individual statements, highlighting each line as it executes.

Execution continues until a break point is encountered, an error takes place, or the application completes execution. You may also select Break to suspend execution. Use this command to visually watch your program source code execute.

Break

This command suspends execution if you are in Go or Animate mode.

Change Start Point to Cursor

This command allows you to move the cursor to any executable statement in your program and restart execution from there. This allows you to branch execution freely around your program. Be cautious about branching around Exits, out of Performs that have not completed, or any other loops which impact logic flow.

Run to Cursor

This command executes your program at machine speed from the current execution point, stopping on the line you currently have the cursor positioned on.

Step Into

This command will execute the current statement and typically stop on the next executable statement. If the

statement you've asked it to execute forces a branch somewhere (such as a CALL to another program, or a PERFORM), you will be branched to that section of code and positioned on the next executable statement.

Step Over

This command will execute the current statement. If the current statement forces a branch somewhere (such as a CALL to another program, or a PERFORM), all of that code will be executed automatically at machine speed. You will then be placed on the next executable statement physically following the statement you've just executed. This allows you to execute an entire PERFORM or a CALL to another program as if it were a single statement.

Run to Next Program

This command causes execution to take place at machine speed from the current statement until control is passed to the next program (for example, via a CALL). Execution will then be suspended on the first executable statement in the next program encountered.

Specific Execution

This command allows you to execute until a specific condition is met. The following conditions can be specified as specific execution conditions:

- Executing up to a statement of a specified type
- Executing up to a specified position
- Executing up to an entry
- Executing up to an exit
- Executing up to a file access

Specific Execution Condition

This command allows you to set one of the above mentioned specific execution conditions.

Try out a few of these commands. Try executing single statements with the Step Into command, for example. Select Animate and watch the program begin executing before your eyes. Select Break to stop animation.

Note that many of the commands are available via the icons on the tool bar. Commands are also available via shortcut menus. Shortcut menus are small pop-up windows of commands made available by clicking on the right mouse button.

Move the mouse out over the program's source code somewhere and click on the right mouse button. A shortcut menu appears with command options.

You have complete power over all aspects of execution while using the debugger.

Data Commands and Advanced Debugging

As noted above, the Fujitsu COBOL debugger provides a number of advanced options. Significant functionality was also built in to handle program data.

Click on Debug to bring up its menu. The Debug menu offers a number of options and commands.

The menu should appear as follows:

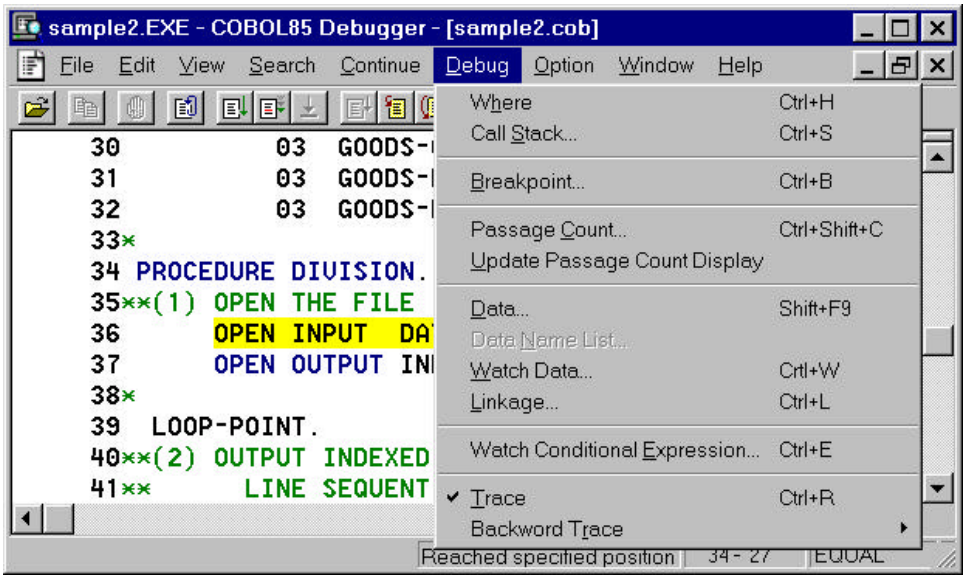


Figure 37. The Debug menu in the COBOL85 Debugger

The following commands and options are available on the Debug menu:

Where

This command will position the source code window to wherever the next executable statement is contained.

Call Stack

This command displays the current call stack. This information is useful in determining application program flow (e.g. where did I get into this particular program from?). You may also use this to allow the debugger to switch to any programs within your application that may be currently active in memory.

Breakpoint

This command allows you to set breakpoints to force the debugger to suspend execution on any given executable statement. You may set simple or complex conditional breakpoints.

Passage Count

This command brings up the passage count window, where you may select lines of code to have statement execution counts tracked. When you select a line of code to be tracked, that line is highlighted in color.

Update Passage Count Display

The statement in which a passage count point is set in the source file window is colored. A statement not passed and an already passed statement can be distinguished by different color. This command allows the display color to be updated in accordance with the latest information.

Data

This command allows you to interrogate and/or change the current value of a data item. You may also set up a watch data window from here to monitor a data item visually as the program executes.

Data Name List

This command opens the Data Name List dialog box to display a list of data names used in a presentation file or network database.

Watch Data

This command allows you to specify data items to be monitored (watched) during execution.

Linkage

This command acquires the data area that was defined with the Linkage Section of the currently stopped program.

Watch Conditional Expression

This command allows you to set and watch a conditional expression related to your program's execution.

Trace

This command causes the debugger to begin collecting trace information which can be utilized for examining execution paths and tracing backwards.

Backward Trace -

This option allows you to specify one of the following trace commands:

- Previous Statement
- Next Statement
- Previous Procedure
- Next Procedure

Try out some of these commands and options. If you get into trouble, select Re- debug from the Continue menu.

For example, try setting a breakpoint on the READ statement in the program and then select Animate or Go. The program will be executed until the READ statement is reached.

Turn on a Passage Count for the READ statement, and another for the CLOSE statement. When you are finished executing the program, check the Passage Count window for execution counts for these two statements.

Maximize the Debugger screen to fill the entire screen if you have not done so already. Create a Data Watch Window for the

DATA-RECORD data item. Watch it change as you step into the READ statement each time.

Change the value of the DATA-RECORD data item on the fly.

Other Debugger Options

The Options menu contains environment setting options, as well as automatic debugging and log file options. Click on Option to bring up its menu.

The menu should appear as follows:

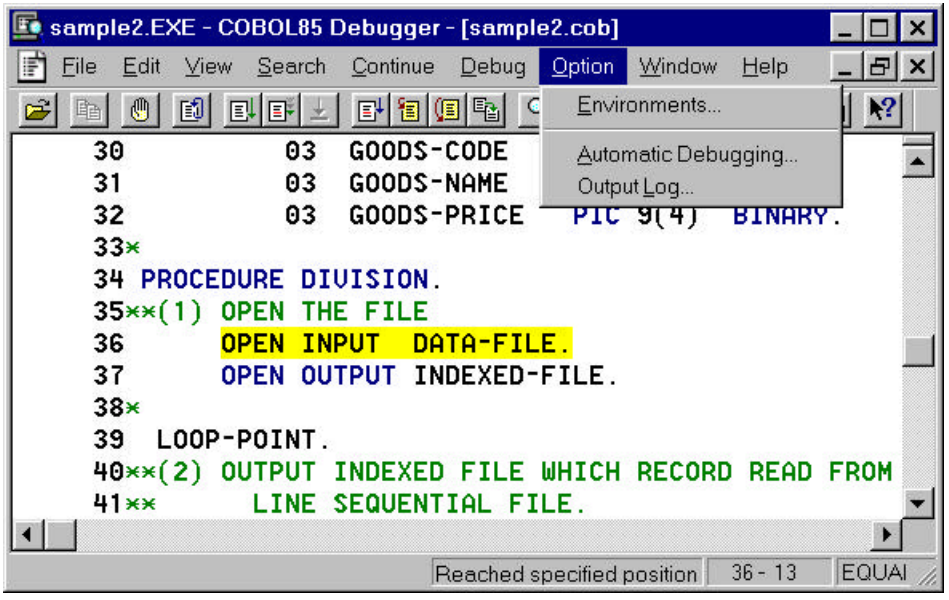


Figure 38. The Options menu in the COBOL85 Debugger

The following commands are available in the Options menu:

Environments

This command brings up an option window where you can set the following options:

Animation Speed

Perform Nesting Display

Display Sub-Program Text

Statement Identification Mode (Editor or Relative Line Number).

Debugger windows initialization.

Sound (beep on each execution break). If you don't like the debugger beeping at you after it executes and stops on each statement, you can turn this off here.

Automatic Debugging

This command allows you to specify a previously recorded log file to be read in and acted upon. The debugger will play back the log file, thus reenacting prior execution steps.

Output Log

This command allows you to create an output log file and start and stop recording to the log file for potential playback later. This log file is also referred to as a History File. You may examine it to determine key execution information from a prior run.

One other useful feature that may not be obvious is the debugger's split screen facility. This allows you to split the screen and view two separate portions of your source code.

To split the screen, locate the vertical scroll bar on the right side of the debugger window. Now look at the top of this scroll bar

and just above its top arrow, you will see a very small horizontal bar.

Move the mouse pointer to this small horizontal bar and hold the left mouse button down and move it into the source code window. This will split the screen at the point you drag it into. You may drag it around and create various sized split screens.

To leave split screen mode, simply grab the bar with the mouse again and drag it back to where it came from.

A Quick Look at Project Management

Ensure that the P-STAFF window is active. You are now going to examine some of the project management features available in Fujitsu COBOL.

Previously in this chapter, we have examined how to edit, compile, execute and debug a single program COBOL application.

Unfortunately, few COBOL applications consist of a single small COBOL program.

Managing COBOL applications often entails managing a variety of programs and sub-programs. COBOL Development environments offered by other vendors in the past have provided little in project management capabilities for the developer.

This is surprising, given the age, complexity and price of some of these products. It’s even more surprising when one notes that other language developers (e.g. C and C++ programmers) take these facilities for granted in their development tool sets.

Fujitsu quickly recognized the need to assist COBOL developers in managing their application suites during development.

Intelligent management and make facilities were thus designed into the P-STAFF development environment.

To fully appreciate these facilities, it is important that you understand the concept of a “make” facility.

The Make Facility Concept

Most applications, regardless of the programming language, are created via the same steps:

- Find all of the needed program source files, copy files (include files), and proper versions of each.
- Compile all of the individual programs into object code (remembering any specific compiler directives required - sometimes on a per program basis).
- Determine which additional object code libraries need to be linked in, based upon functionality utilized in the programs. For example if you utilize an indexed file, you need to be sure to link in the indexed file I/O library, etc.
- Link edit the application together with appropriate linker options to create the executable(s).
- Test and/or deliver the application. This process was typically complicated by the fact that stepping between debugging and production versions of an application required recompiling and re-linking everything.

What developers needed was some sort of intelligent management utility that would keep track of all this and allow the developer to simply enter a command or click on a button and rebuild everything automatically.

To make this even more desirable, it would be even better if an extra level of intelligence was added to look at each piece of the

overall puzzle and determine if it even needed to be recompiled or re-linked.

For example, you works with a 25 program application and changes a single line in only one of the 25 programs, you should not be forced to re-compile all 25 programs before re-linking. Rather, you should only have to re-compile the program that changed, and re-link with all of the other previously compiled object modules.

Developers were also forced to remember application specifics such as “which one is the main program?” and link in the correct order to produce the proper executable.

Based on this rather obvious need, the concept of an intelligent make facility was born.

Most make facilities are based upon the fact that application components are dependent upon each other. For example a main executable (EXE) file is dependent upon the object files and libraries that were linked together to create it.

These object files and libraries are themselves dependent upon the program source files that were compiled to create them.

Its thus seems both logical and straightforward that someone could develop a utility that simply checks date and time stamps on files to and re-builds an application based upon the results.

For example, if a certain object file is found to be newer than the current executable program, it’s clear that the program needs to be re-linked to include the newer version of the object code.

If a source program is found to be newer than the current object file, it’s clear that this program needs to be re-compiled and the application needs to be re-linked to create a new executable that encompasses the new program changes.

By maintaining a small history file for an application, an intelligent make utility can greatly reduce redundant compiles

and links, as it will not have to guess if something has changed since the last build.

Surprisingly enough, make utilities have been common for languages such as C, C++, Pascal, Basic, etc., for many years.

Most COBOL vendors, however, have failed to provide this rather obviously needed utility or have only provided part of what's needed. Instead, users of these environments are forced to manage this manually. One of the largest challenges in these environments is determining how to build the application together once it is tested.

Stepping from the pseudo code world into native code and building a stand alone application can be a daunting task using one of these products. Many PC COBOL developers have spent time building, executing, noting down errors (primarily of missing programs which they were unaware had to be linked in), and going back to linking and executing over and over again.

One way around this was to give up on the idea of building a self-contained application, and instead installing megabytes upon megabytes of legacy run-time systems on any machine destined to run a COBOL application.

Developers then had to be concerned about potential updates to either their applications, or the COBOL vendor's legacy run-time system, creating new problems.

Needless to say, this has proven to be a less than ideal COBOL development scenario.

A Look at the P-STAFF Project Management Facilities

In this section, we will look at the P-STAFF Project Management Facilities by creating a sample project file. **Note**: This activity is also covered in Sample 5 in Appendix A.

Sample 5 supports creation of a new project file (SAMPLE5.PRJ). The project file is based on the assumption that C:\FSC is specified as the COBOL85 installation directory. In order to create a new project file, do the following:

1. Create the project file (SAMPLE5.PRJ).

   Select Open from the Project menu of the P-STAFF window, then type the project file name (SAMPLE5.PRJ) in the Open Project dialog box. Click on the Open button.

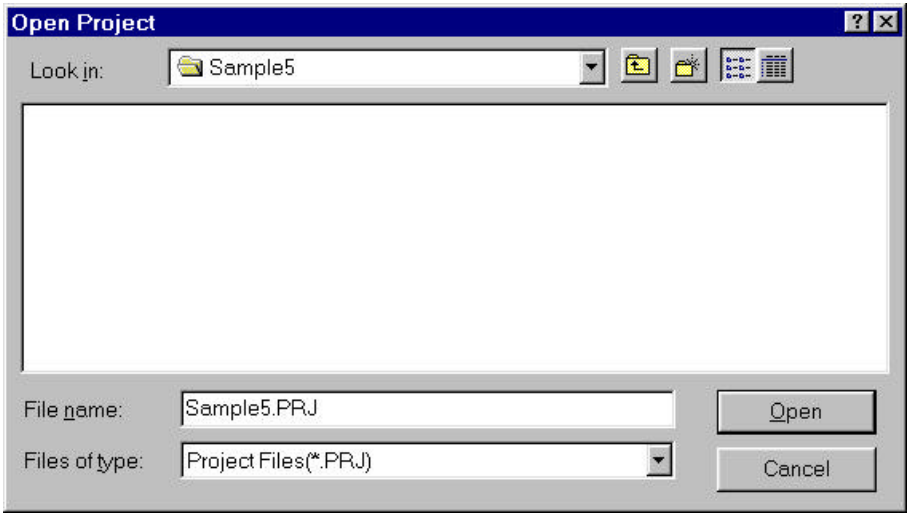


Figure 39. The Open Project dialog box

The following window is displayed.

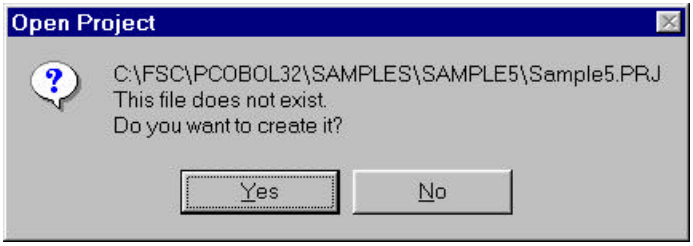


Figure 40. The Open Project creation window

Click on the Yes button to create the SAMPLE5.PRJ file.

The Sample5.prj window (inactive) and the Target Files dialog box (active) are displayed.

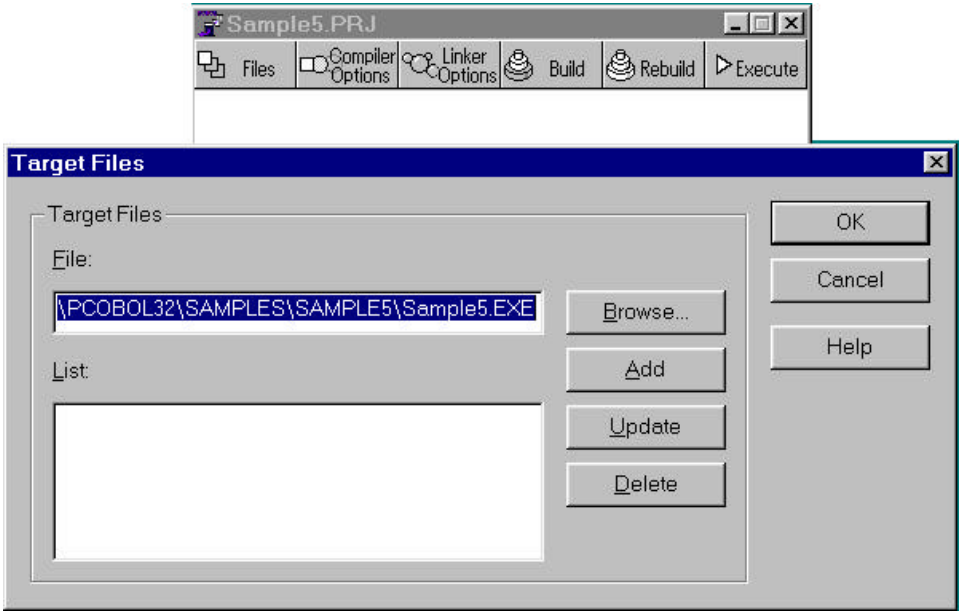


Figure 41. The Target Files dialog box and the Sample5.PRJ window

2. Register a target file.

   By default, the executable program name appears in the File edit box. Click on the Add button to specify the executable program name (SAMPLE5.EXE) of the main program to be created.

   You can type over the executable program name to specify the DLL name (PRINTPRC.DLL) of the subprogram to be created. Then click on the Add button to add the DLL name to the list.

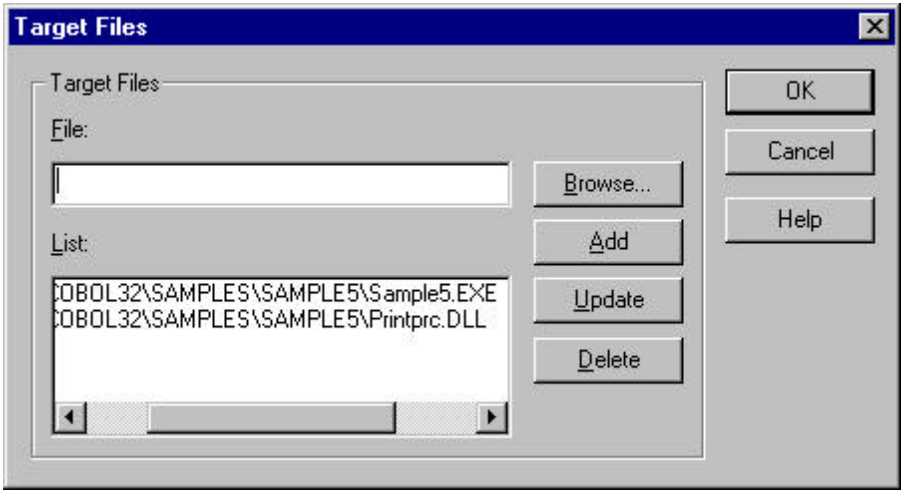


Figure 42. The Target Files dialog box

3. Register dependent files.

   Click on the OK button of the Target Files dialog box. The Dependent Files dialog box opens.

   By default, the SAMPLE5.EXE file appears in the Targets edit box in the Dependent Files dialog box. Click on the Browse button to locate the source file name (SAMPLE5.COB) of the main program. Highlight SAMPLE5.COB in the Browse Files

window and click on the Open button. Then click on the Add button to add SAMPLE5.COB to the list.

Highlight SAMPLE5.COB in the list and click on the Main Program button. The SAMPLE5.COB icon is displayed in red.

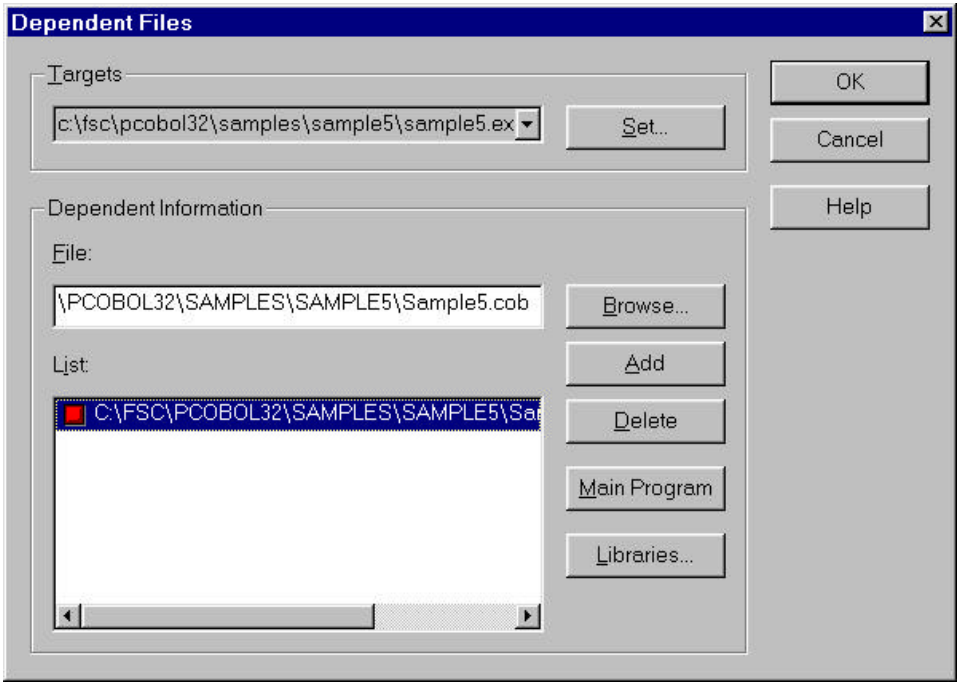


Figure 43. The Dependent Files dialog box

4. Register a library file.

   Highlight SAMPLE5.COB in the list and click on the Libraries button. The Library Files dialog box is displayed.

   Click on the Browse button to locate the library file (S_REC.CBL) and then click on the Add button to add it to the list.

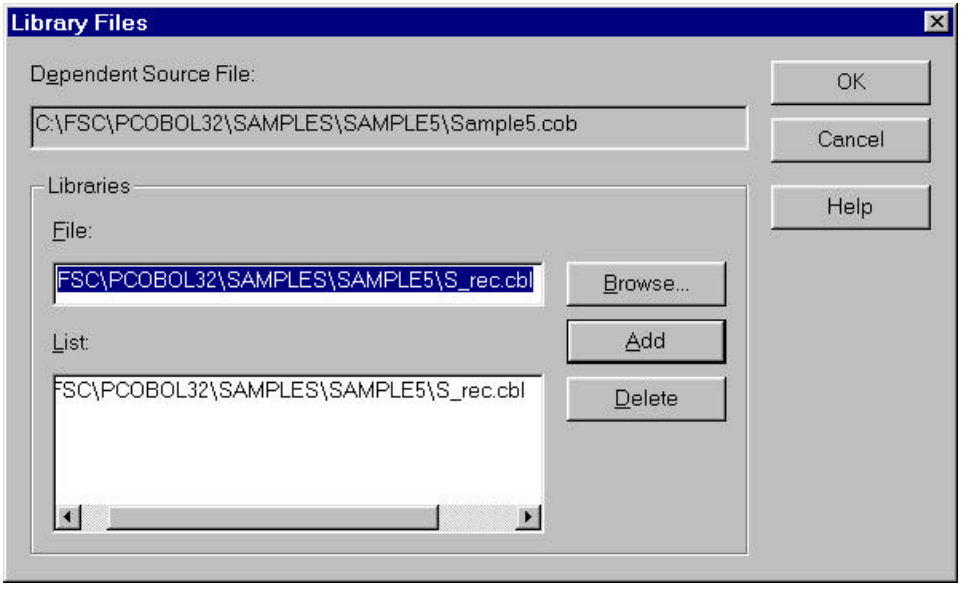


Figure 44. The Library Files dialog box

Click on the OK button to return to the Dependent Files dialog box.

5. Specify the import library name (PRINTPRC.LIB). This file will be created by P-STAFF when the PRINTPRC.DLL file is created.

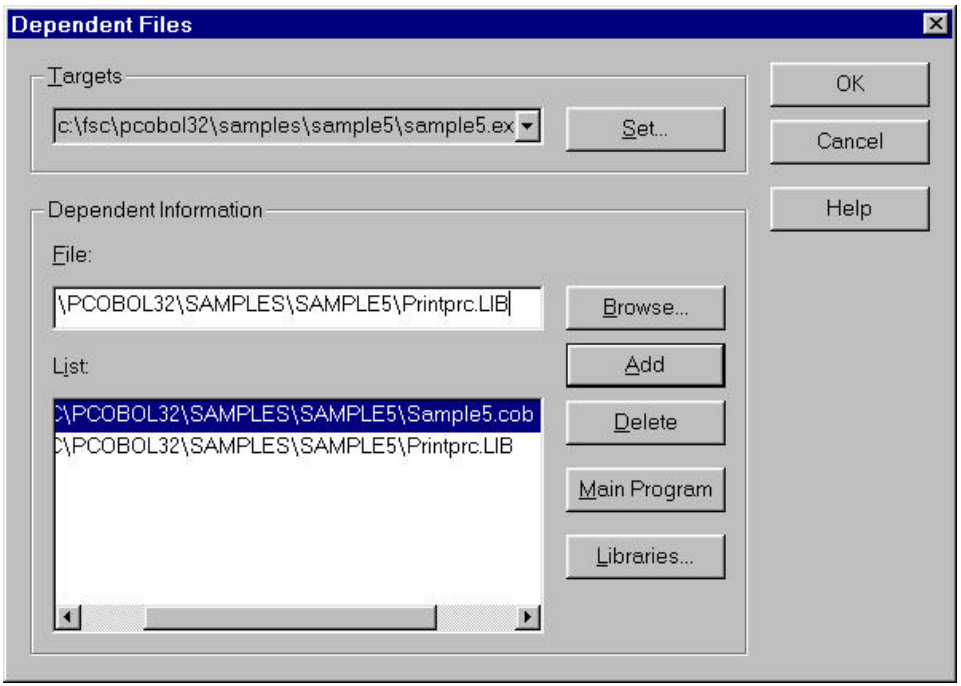


Figure 45. The Dependent Files dialog box

6. Change the target to PRINTPRC.DLL.

   Specify the source file name (PRINTPRC.COB) of the subprogram in the same manner as described above. **Note**: DLL's do not have a main program so you cannot specify PRINTPRC.COB as a main program.

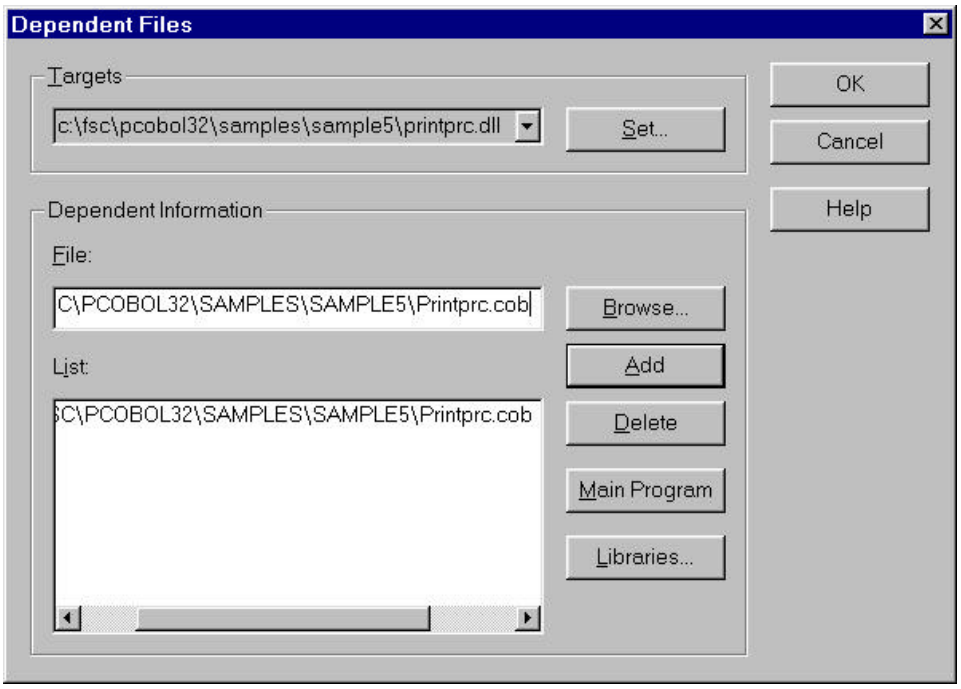


Figure 46. The Dependent Files dialog box

7. Register a library file.

   Highlight PRINTPRC.COB in the list and click on the Libraries button. The Library Files dialog box is displayed.

Click on the Browse button to locate the library file (S_REC.CBL) and then click on the Add button to add it to the list.

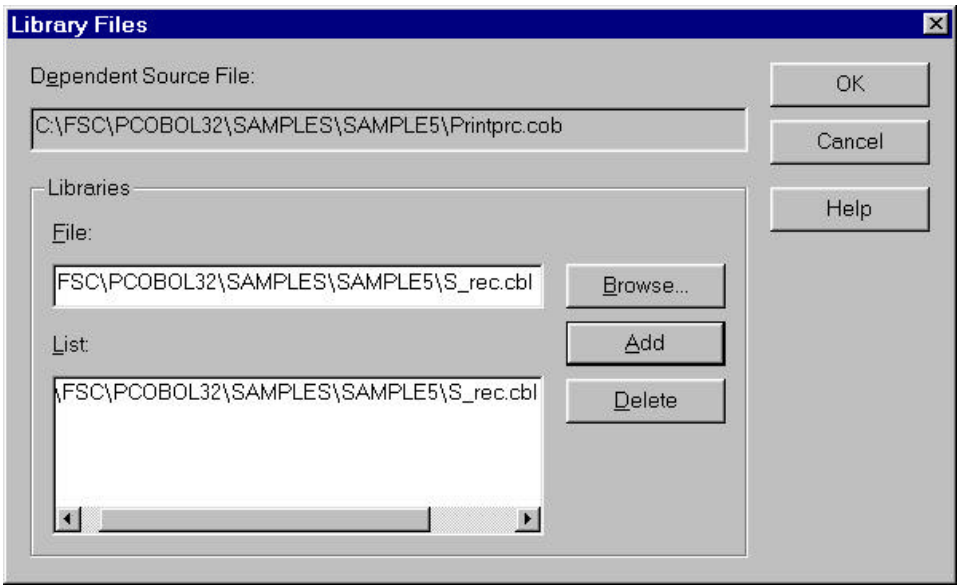


Figure 47. The Library Files dialog box

Click on the OK button to return to the Dependent Files dialog box.

Click on the OK button at the completion of file registration.

The programs (SAMPLE5.COB, PRINTPRC.COB) fetch the library text (S_REC.CBL) with the COPY statement. Registering library files in this manner allows P-STAFF to automatically recompile programs when library files have been modified.

The registered file names are displayed in the project window. The files listed may vary in physical location depending upon where you installed the product.

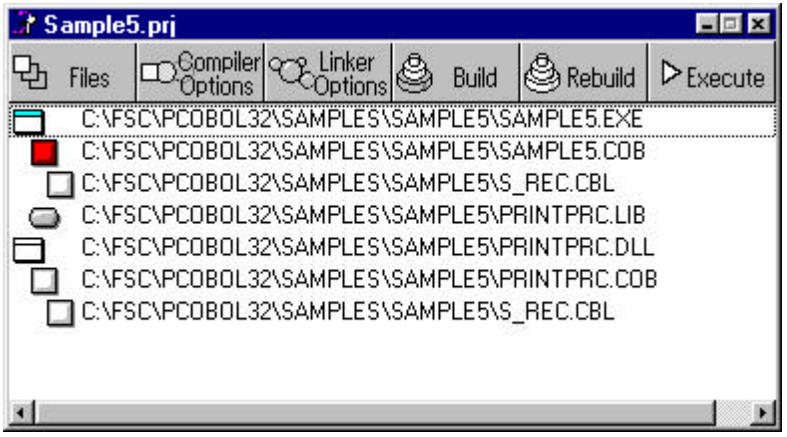


Figure 48. The project window

You will note that the project window lists the files that comprise the Sample 5 project. These files are presented in a hierarchical form with different types of icons to visually illustrate their inter-dependencies and relationships.

The two left most icons (next to SAMPLE5.EXE and PRINTPRC.DLL), represent executable files which are dependent upon the files indented below them.

The SAMPLE5.EXE executable is dependent upon SAMPLE5.COB (the COBOL source program for it).

SAMPLE5.COB has a dependency upon the S_REC.CBL copy file, and the red box icon next to it identifies it as producing the main application executable (SAMPLE5.EXE).

The PRINTPRC.LIB file establishes the proper entry point information for calling the PRINTPRC sub-program.

The PRINTPRC.DLL file is a dynamically callable executable, which will be called at run-time by the main SAMPLE5.EXE application.

PRINTPRC.DLL is dependent upon the PRINTPRC.COB source program, which is also dependent upon the S_REC.CBL copy file.

You may utilize the project window to move around the components of the Sample 5 application. If you double click on one of the source files, for example, that file will be loaded into a fresh edit session within P-STAFF.

Move around the project window and gain a better understanding of its usage. Clicking on the Rebuild button will cause the entire application to be compiled and rebuilt, including all of the sub steps required.

Note that project files are static in nature, and if you have altered the default installation directory, the SAMPLE.PRJ file will not reflect the current directory where its files are contained. This will cause errors when you attempt to rebuild the application.

You will utilize this facility in the next chapter as you build your first Visual Basic COBOL application, to create a new project and manage it.

Chapter 4. Creating Your First Visual Basic COBOL Application

In this chapter, you will learn how to create your first COBOL application which uses Visual Basic as a graphical front end.

It is assumed before you begin that you have already installed Visual Basic, version 4.0 or higher on your machine. If you have not, please do so before continuing this section.

It is also assumed that you have gone through the Visual Basic Tutorial entitled “Your First Visual Basic Application,” located in the “Visual Basic Programmer’s Guide.” This will help you understand the terminology used later on in this chapter.

An Overview of the Visual Basic COBOL Application

The application you are going to develop will be a simple graphical window with a text box and two command buttons.

It will be named HICOBOL (short for "Hello From COBOL").

The text box will be assigned a text value by Visual Basic. One of the command buttons, when selected, will call your HICOBOL program passing two parameters.

One parameter is a numeric value. The COBOL program will set this value to 100 each time, but it will be ignored by the Visual Basic part of our application (this parameter is defined to illustrate the technique). You may enhance the application on your own to use this value somehow.

The second parameter is a character string. The COBOL program will always set this to "Hello From COBOL. " When the COBOL program exits, this value will be returned to Visual Basic and our graphical text box will be updated with this new string to illustrate the interaction between Visual Basic and COBOL.

This will happen each time the user clicks on the command button. The second command button will cause our graphical text box to be reset to display a different string.

As noted previously in this manual, you will develop the COBOL portion of this application using the Fujitsu COBOL P-STAFF development environment. You will develop the GUI portion of this application using Visual Basic's development environment. You may develop the two separate portions of the application in any order.

How Fujitsu COBOL Interacts with Visual Basic

Fujitsu COBOL was created to work well with other language environments from a run time perspective.

Other COBOL vendor's products have traditionally taken a separate approach, being forced to do so primarily by their legacy run time systems.

Most programming API's (application programming interfaces) available on the Windows platforms support a C-style programming interface. This includes not only operating system services such as GUI interface components, but additionally a vast array of other programming interfaces. These include data base systems, client server messaging and transaction systems, and even end user applications.

Because of the design of most COBOL development products, developers have in the past often times been precluded from accessing these programming API's effectively because of poor or non-existent mixed language support from COBOL vendors.

The arrival of Fujitsu COBOL has changed all of this and now levels the playing field for COBOL developers. Visual Basic integration is but one example of the excellent mixed language programming support found in Fujitsu COBOL.

The integration point between Fujitsu COBOL and Visual Basic is the creation of a DLL (dynamic link library) file from Fujitsu COBOL.

Instead of producing a standalone executable (EXE) file within the COBOL development environment, you instead produce a DLL file. You can think of a DLL file as a dynamically callable COBOL subprogram, which does not have to be physically linked to a program that calls it and may even pass data back and forth.

To create a proper DLL, the linker needs a side file known as a DEF file. This file is typically quite short. It contains the name of the DLL to be produced and also defines EXPORTS.

Exports are names of entry points contained within the DLL file which are to be made public to allow other applications to call them after the DLL has been identified to the operating system.

We are going to create a simple DLL file called HICOBOL.DLL, which will contain a single exported entry point - the program name. The program name entry point will cause a calling program to enter the program at its first executable statement in the Procedure Division.

You will also create a supporting DEF file called HICOBOL.DEF for the linker.

Once you have written the HICOBOL.COB program, the HICOBOL.DEF program, and have created the HICOBOL.DLL executable, you will exit Fujitsu COBOL and enter Visual Basic to design the graphical interface.

While in Visual Basic, you will create a graphical user interface (GUI) for your COBOL application, which will call the HICOBOL.DLL program. Data will be passed back and forth as well.

To identify the Fujitsu COBOL DLL to the Visual Basic program, you will code three simple Visual Basic Define statements. One is for the Fujitsu COBOL runtime load program. Another is for the Fujitsu COBOL runtime unload program. The final DECLARE is for the HICOBOL.DLL executable you are going to create.

In order to get a Fujitsu COBOL DLL program to work with Visual Basic, you need to first issue a single call from Visual Basic to ensure that the COBOL runtime support is properly loaded.

You then code a single Visual Basic statement to call the HICOBOL.DLL program and pass data to it.

Finally, you code a call to unload the COBOL runtime support when finished.

That's all there is to it.

From an architectural standpoint, when you develop future, more complex Visual Basic COBOL applications, you will arrange things slightly differently than in the example below.

The main reason for this is that you only need to load the Fujitsu COBOL runtime once at the start of your application, and then issue the unload call just before you exit your Visual Basic application entirely, for performance reasons.

To keep our example simple and straight forward, we are going to code all three calls together (load the COBOL runtime, call HICOBOL, unload the COBOL runtime).

Developing the COBOL Portion of the Application

In this section you are going to develop the HICOBOL.DLL executable which will be called by Visual Basic.

You are going to use the project management facilities within P-STAFF to assist you.

Bring the P-STAFF development environment up by going to the Fujitsu COBOL 3.0folder (directory) and executing Programming Staff 32.

The following window appears:

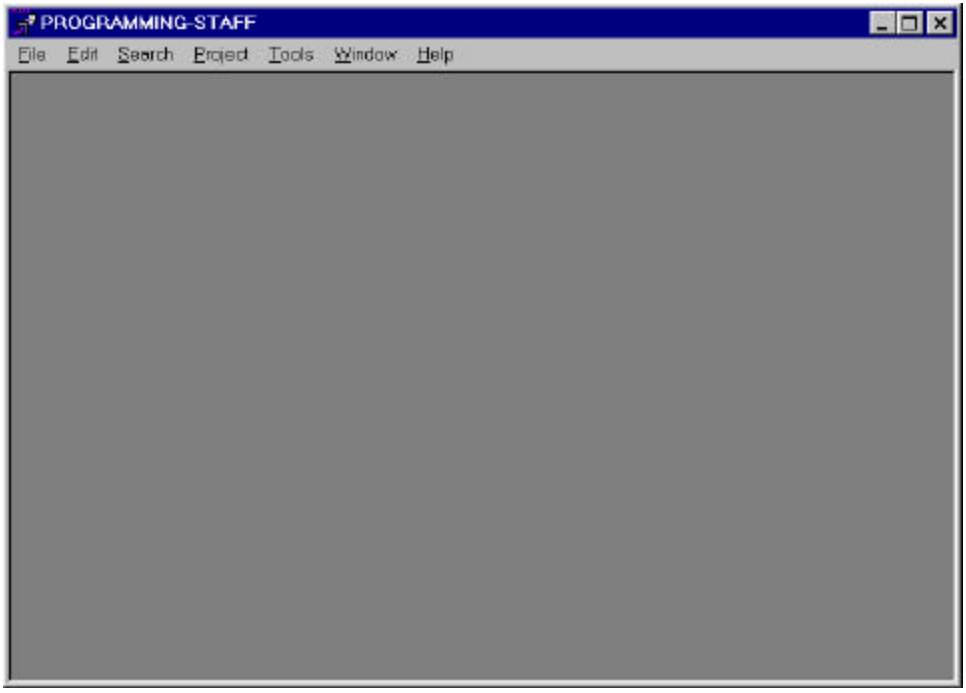


Figure 49. The PROGRAMMING-STAFF (P-STAFF) window

Select Open in the Project menu.

The Open Project window appears. You need to create a new folder (sub-directory) to store the application you are about to develop.

The Open Project window has a Create New Folder icon near the upper right hand side.

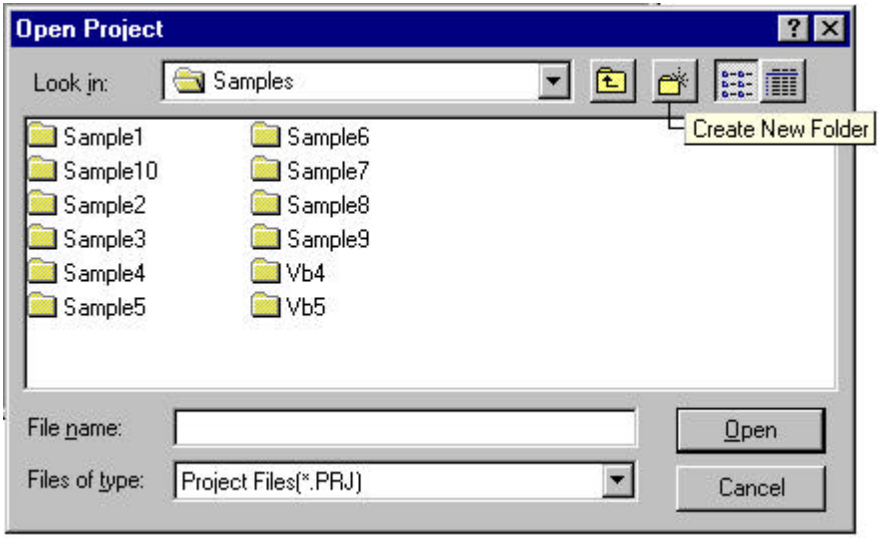


Figure 50. The Open Project window

Click on the Create New Folder icon. A new folder will be placed in the window and the cursor will be automatically placed into the text name field.

Change the name of the folder to HICOBOL, by typing:

HICOBOL

The Open Project window should now look like:

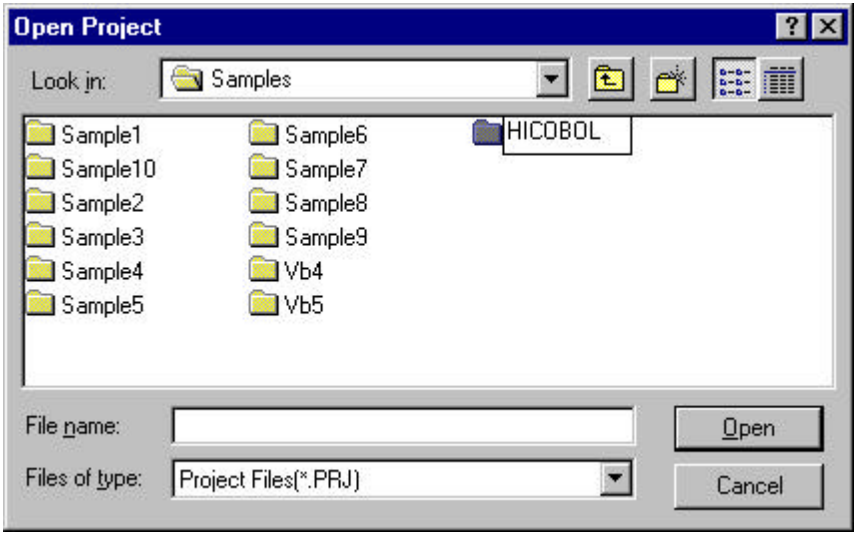


Figure 51. Adding the HICOBOL folder to the SAMPLES directory

Press the ENTER key. The new folder will be created.

Move the cursor to the File Name field, and type in the name of your new project, HICOBOL**.** Make sure that Project Files(*.PRJ) is visible in the Files of type field directly beneath the File name field.

Click on the Open button. The following window appears:

Figure 52. The Open Project creation window

Click on the Yes button, and the project file will be created.

The following dialog box appears:

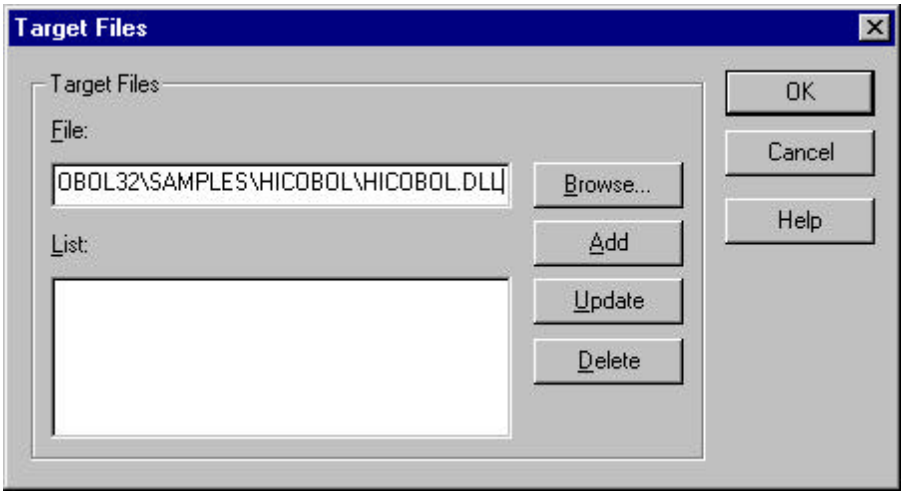


Figure 53. The Target Files dialog box

This dialog box is asking you to list all of the executable files you plan on producing in this project. It has already taken a guess as to one and listed its name in the File field.

Please note that on your machine, the file extension type may be .EXE instead of the desired **.**DLL for this project. If so change this in the File field, as noted in the above example.

Click the Add button to add this to the list of executables.

Because you are not planning to create any other executables, click on the OK button.

The following dialog box appears:

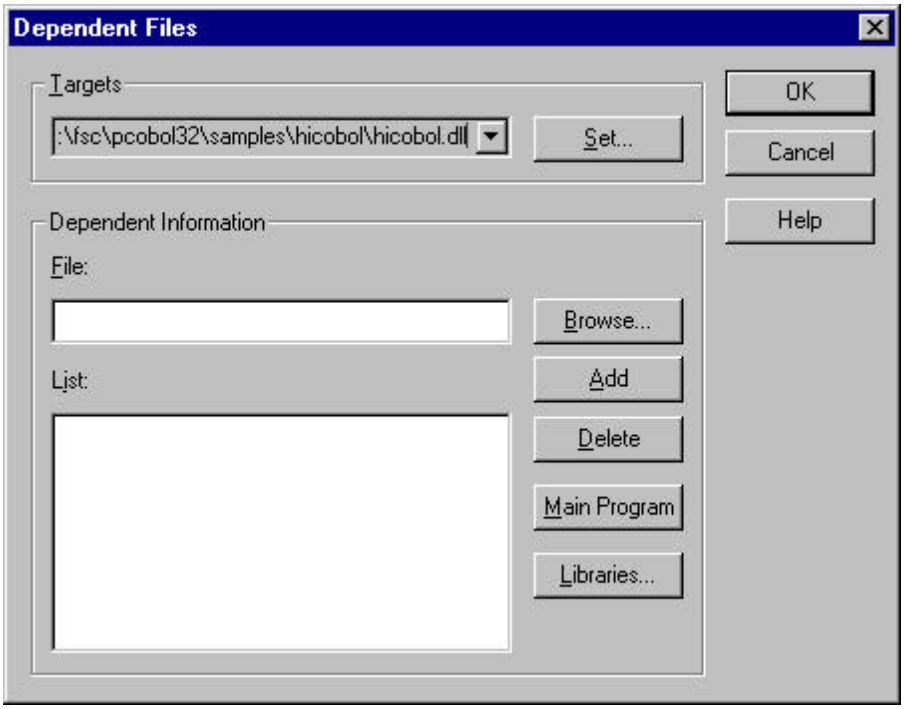


Figure 54. The Dependent Files dialog box

This dialog box is requesting that you list all of the files upon which the HICOBOL.DLL executable is dependent on.

You will need to list two files. One is the HICOBOL.COB source program you are about to create, and the other is the definition (DEF) file required for the linker to create a DLL.

Enter HICOBOL.COB in the File field and click on the Add button.

Next enter HICOBOL.DEF in the File field and click on the Add button again.

The Dependent Files dialog box should now look like this:

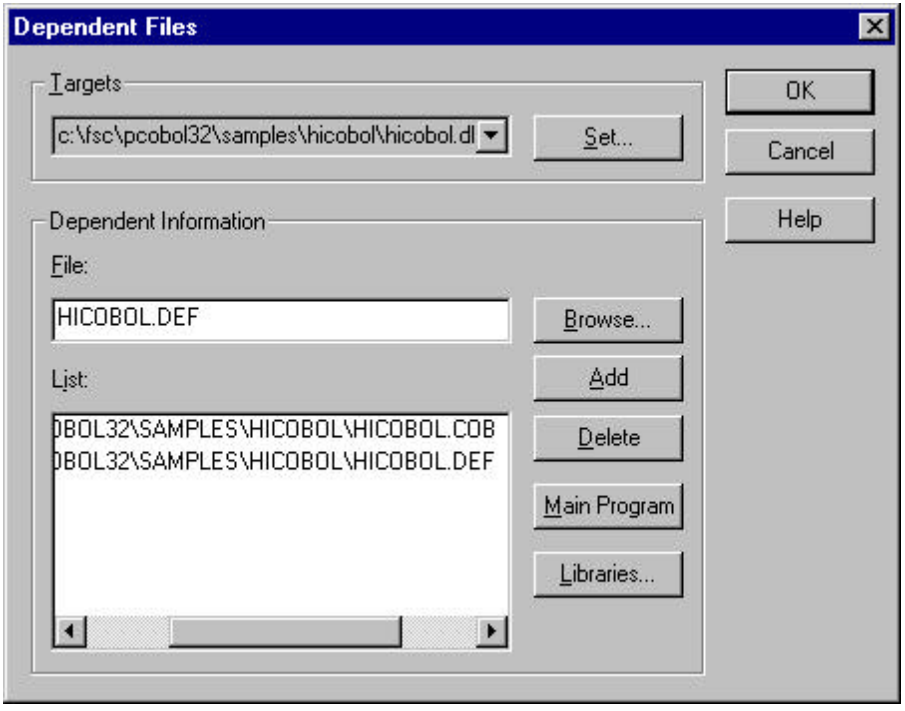


Figure 55. The Dependent Files dialog box

Click on the OK button. The following window appears:

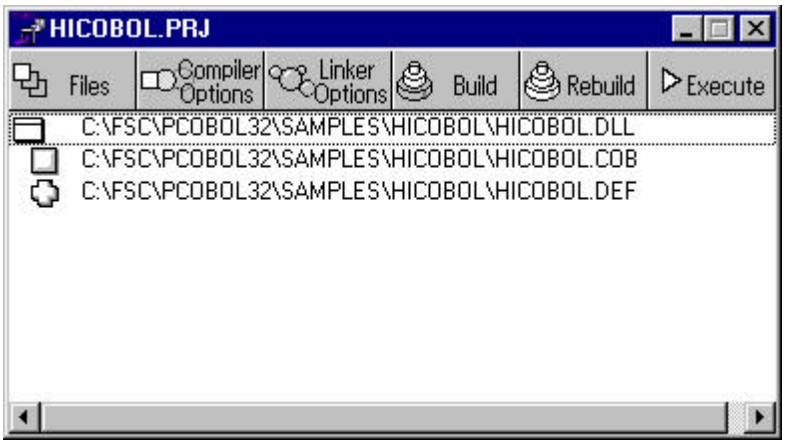


Figure 56. The Project window

You are now ready to create the source COBOL program and the DEF file.

You will also get a taste of how the project window interacts with the P-STAFF development environment to simplify your development tasks.

Within the Project window, double click on the line that contains the HICOBOL.COB file name. This will bring up a blank Edit session window inside the P-STAFF facility as follows:

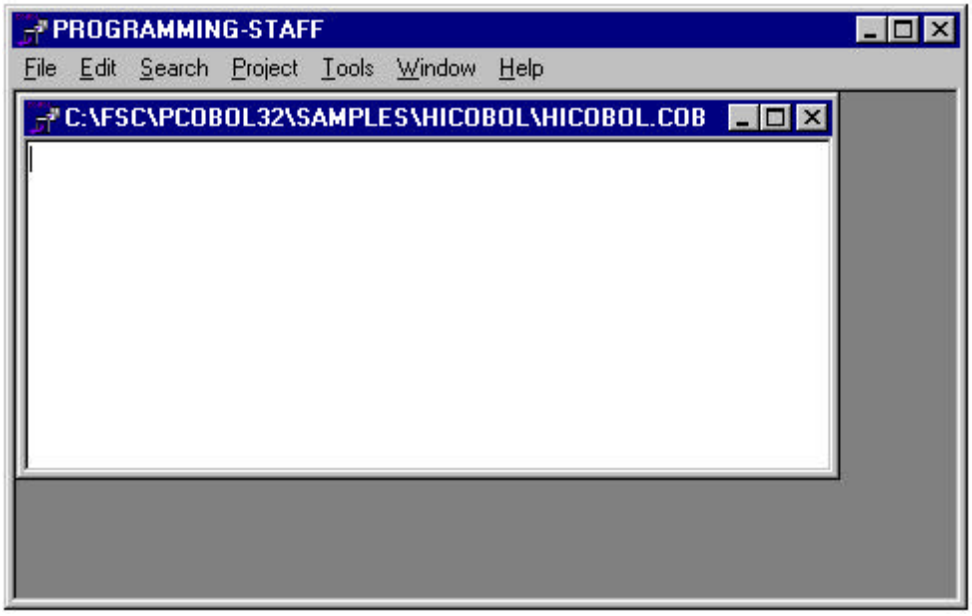


Figure 57. The P-STAFF Edit window

The cursor will automatically be placed at the start of the new Edit window. You are now ready to type in the HICOBOL.COB source program.

Type in the following short COBOL program in the Edit window:

```
000010 IDENTIFICATION DIVISION.
000020  PROGRAM-ID. HICOBOL.
000030*
000040 DATA DIVISION.
000050 WORKING-STORAGE SECTION.
000060 LINKAGE SECTION.
000070 01 vbInteger    PIC S9(4) COMP-5.
000080 01 vbString      PIC X(16).
000090 PROCEDURE DIVISION WITH STDCALL LINKAGE USING vbInteger, vbString.
000100     MOVE 100 TO vbInteger.
000110     MOVE "Hello From COBOL" to vbString.
000120     EXIT PROGRAM.
```

If you examine the HICOBOL program, you should be able to understand what it does quite easily. You will note that it contains a Linkage Section and thus gets two parameters passed to it, each time it is called.

The vbInteger parameter is always set to the value of 100. The vbString parameter is always set to the value of "Hello From COBOL".

These parameters will actually be defined with our Visual Basic application and passed into our COBOL program. They will be passed back when the program exits. Numeric parameters such as vbInteger are passed by reference, while strings such as vbString are passed by value. The string value will be passed back to Visual Basic and updated, however, because we are changing its Linkage Section value.

The following window appears:

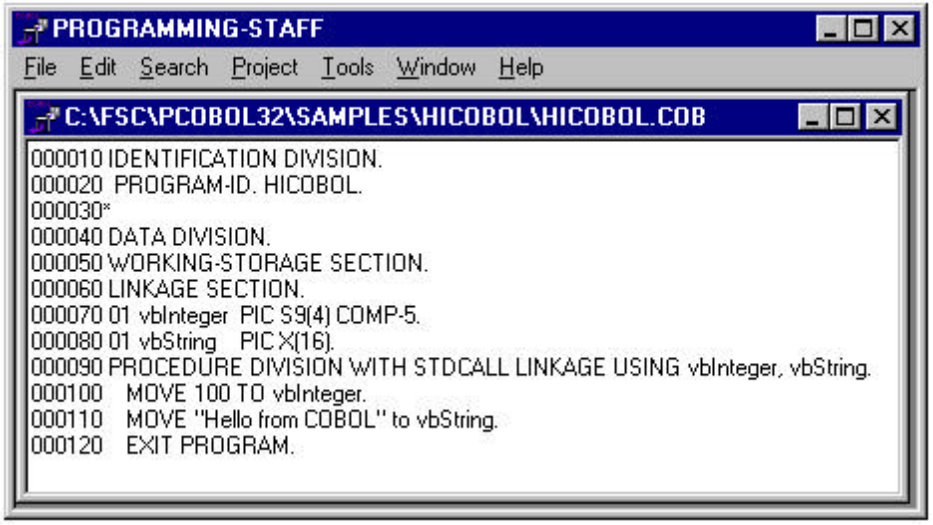


Figure 58. The HICOBOL program in the P-STAFF Edit window

Click on File in the P-STAFF menu to bring up the File menu. Select Save to save the program.

Click on File again and select Close to close the Edit session.

You are now ready to create the DEF file for the linker. **Note**: This is an optional step. If you choose not to create a DEF file, P-STAFF will automatically create one for you.

Move back to the Project Window and double click on the line that contains the HICOBOL.DEF file name. This will bring up a new edit window and position the cursor on the first line.

Enter the following two source lines in the Edit session:

```
LIBRARY "HICOBOL"
EXPORTS HICOBOL
```

The first line specifies the file name of the DLL file to be created.

The second line specifies any entry points that are to be made public and thus callable by any outside program that is made aware of this DLL file.

This is all you need in the DEF file.

The Edit window should appear as follows:

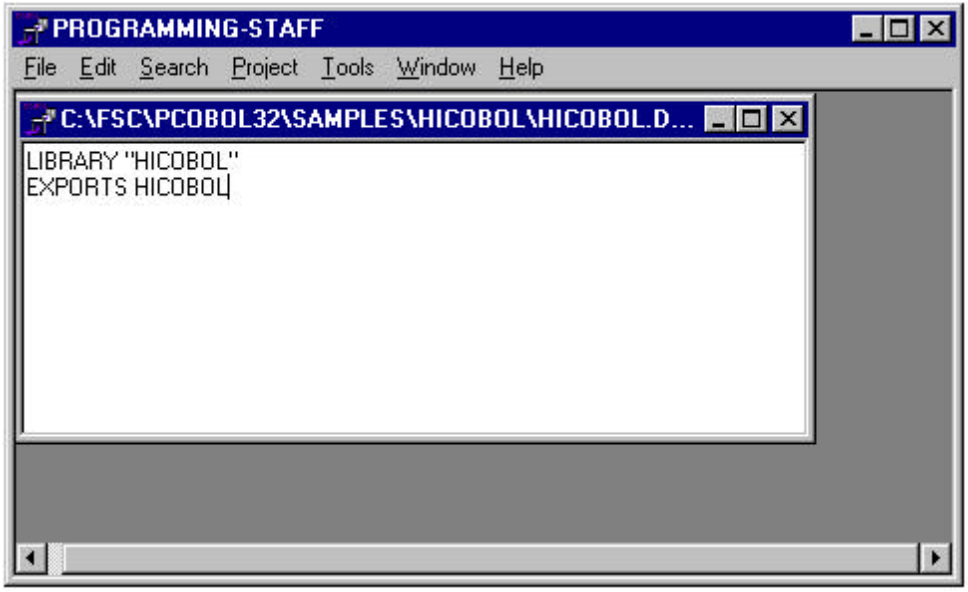


Figure 59. The HICOB.DEF file in the P-STAFF Edit window

Click on File in the P-STAFF menu to bring up the File menu. Select Save to save the DEF file.

Click on File again and select Close to close the Edit session.

You are now ready to compile, link and create the HICOBOL.DLL executable. You can do all of this by moving back to the Project window and clicking on the Build button. The program will be compiled and, if no errors are present, it will be linked into a proper DLL file.

If you receive any compile errors, they will be listed in a window within P-STAFF. Edit the HICOBOL.COB source file, and look carefully to correct any typing mistakes you may have made. Save the file, move back to the Project window and click on the Rebuild button to create the DLL file.

Note that there is a major difference between the results of the Build and Rebuild buttons. Build will simply check to see if there are valid object files present and will re-link them into an executable - even if the program source code has changed!

Rebuild forces a recompile of all changed programs, even if a valid object file already exists.

Upon each attempt to execute a build or rebuild, an MS-DOS window will appear. Close this window when you are finished viewing its contents.

Once you have a successful build (or rebuild) of your application, the following window appears:

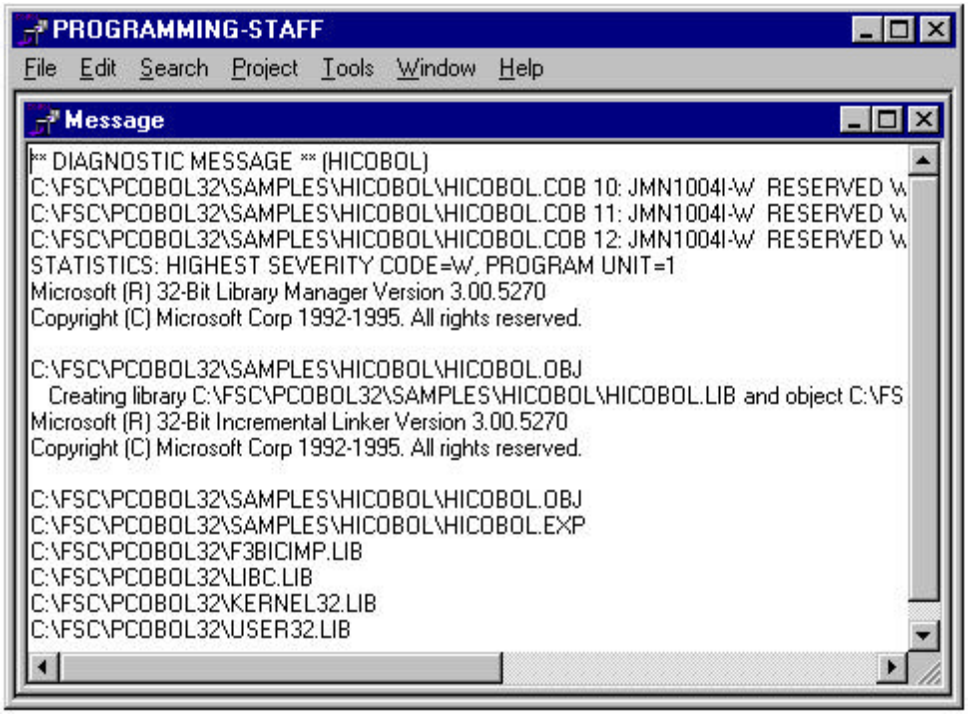


Figure 60. The Message window

You have now completed the COBOL development side of this application. It's time to develop the graphical user interface. This will be done using Visual Basic. Close down P-STAFF (either click on the Close button, or select Exit from the File menu). This will also close the project window automatically.

Developing the GUI Interface Using Visual Basic

You will now use Visual Basic version 4.0 or higher to develop the graphical user interface for the HICOBOL application.

Bring the Visual Basic development environment up. You can do this by selecting Visual Basic icon (4.0 or 5.0) from the Visual Basic folder.

The following window appears:

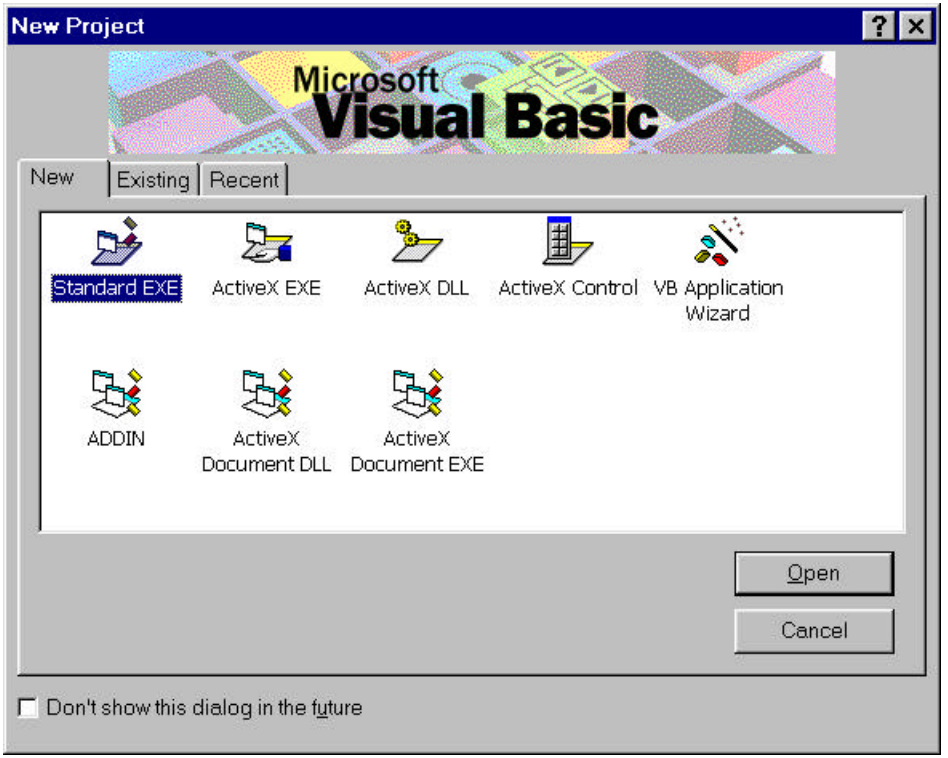


Figure 61. The Visual Basic New Project window

This window asks you to specify which type of application you are going to develop. You are going to develop a Standard EXE, so select this and click on the Open button.

The following window appears:

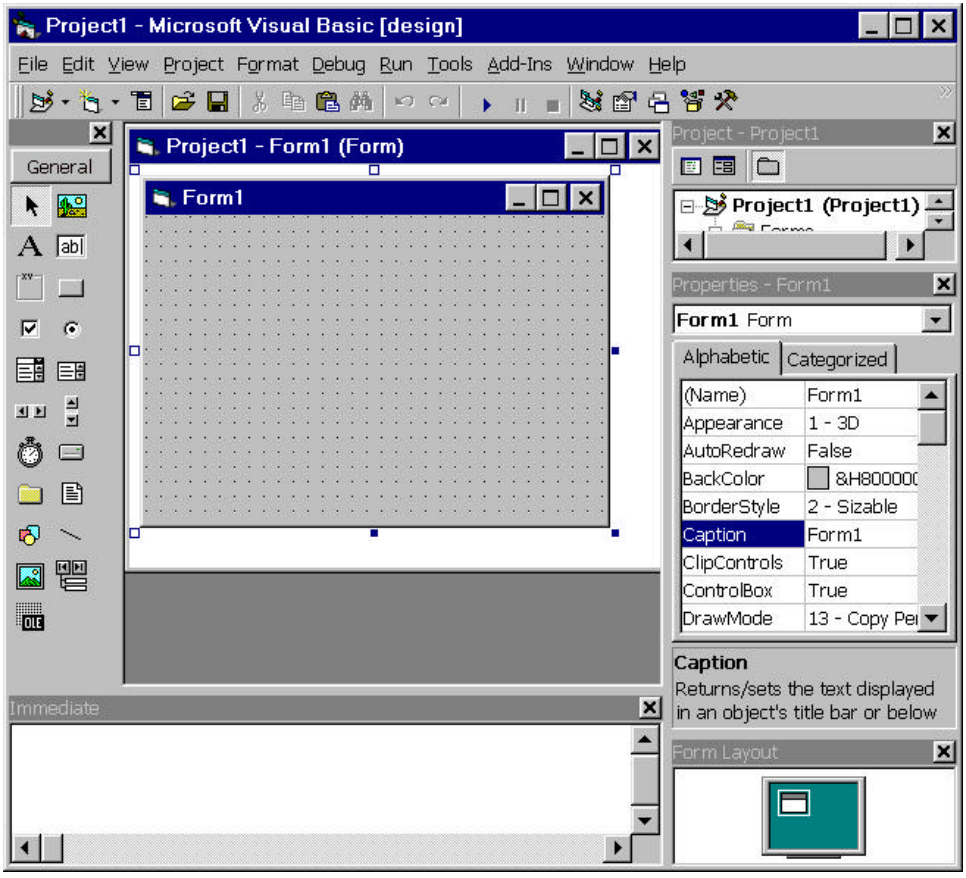


Figure 62. The Visual Basic development environment

You should already be familiar with this environment. If not, it is strongly suggested that you go through the Visual Basic "Your First Visual Basic Application" tutorial in the "Visual Basic Programmer's Guide."

In the center of this window is an object named Form1. This will become our main application window. You are going to create a text box and two command buttons on this form.

Creating the Graphical Interface Objects in Visual Basic

1. To create the text box, move the mouse to the object palette on the left hand side of the window. Click once with the left mouse button on the text box icon (the second icon down in the right hand column - it has an *ab* inside its box).

2. Move the mouse over to the Form1 grid. Starting near the upper left hand corner of the grid, hold the left mouse button down, and drag the mouse down a bit and to the right to draw a text box some like the following:

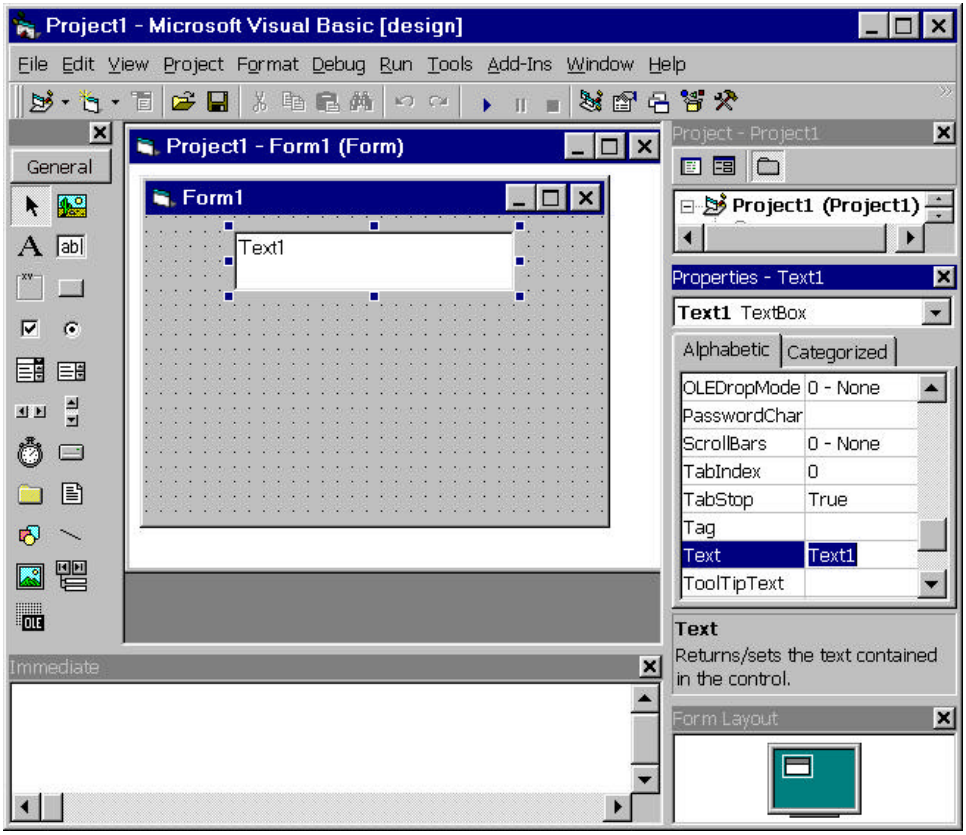


Figure 63. Creating a text box in Visual Basic

3. The cursor should have automatically been positioned in the Properties on the right side of the window. It should be in the Text value area. This contains the default value displayed within the text box. To the immediate right of the Text field change Text1 (the default value for a newly created text box),

to "Hello From Visual Basic". You will notice that it instantly changes in the text box on Form1.

4. Now you will place the first command button near the bottom of Form1. Move the mouse back to the object palette on the left and click once with the left mouse button on the command button icon. It is located immediately under the Text Box icon you previously selected.

5. Now move the mouse pointer near the bottom left of the Form1 window and draw a command button by holding the left mouse button down and dragging it.

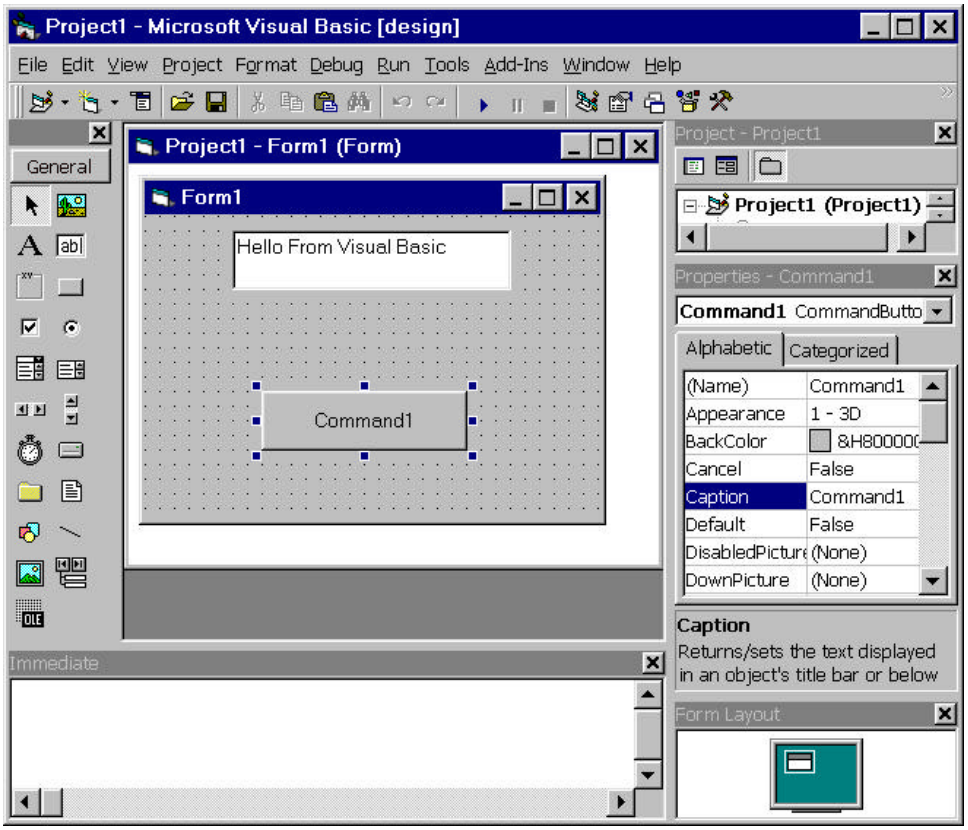


Figure 64. Drawing a command button in Visual Basic

6. Change the Caption value (the text displayed on the command button) from Command1 to Hello COBOL, (the cursor should have automatically been positioned here in the Properties Box), by typing it in.

   You may reposition the command button or the text box by simply selecting either with the mouse and dragging the selected object to the position you desire on the form.

7. You are now ready to place a second command button on Form1 between the text box and the first command button. Repeat the process noted in steps 4 and 5 above and create the second command button. Change the caption value to Reset. Form1 should now look something like the following:

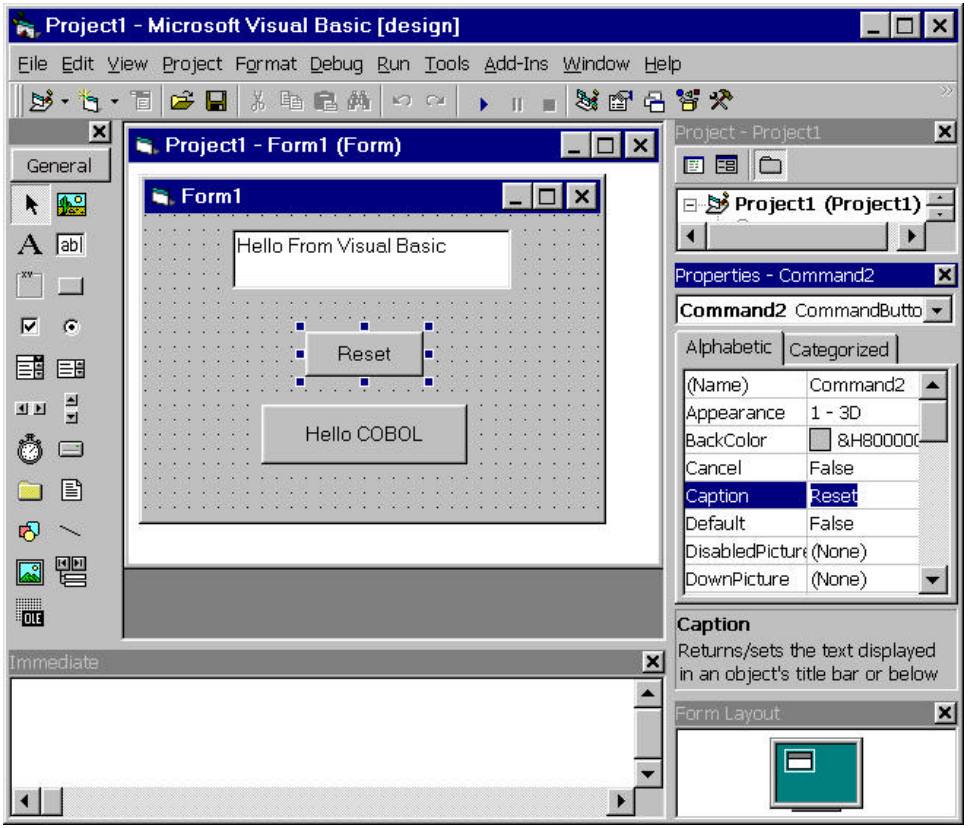


Figure 65. Drawing a second command button in Visual Basic

Creating the Programming Logic in Visual Basic

Now that you have created the application's main window (Form1) and its associated objects, it's time to wire this all together by creating some program logic to define the user interface interaction.

Visual Basic makes this process simple and straightforward in the way in which it has implemented the event driven programming paradigm.

In the following steps you will take care of the following needs:

- Writing the actual Visual Basic Script (programming language) to handle events and to call the supporting COBOL application.
- Declaring the components of the supporting COBOL application to Visual Basic.
- Execute the overall application under the control of Visual Basic.

You will begin by writing the program logic. Visual Basic makes this a very straightforward process.

The first requirement you will take care of is to implement the functionality that is desired for the Reset push button. The behavior that is desired is to have the text box string reset with a standard value each time a user clicks this command button.

Implementing the Reset Button

1. Move the mouse pointer over top of the Reset command button. Double click on this button with the left mouse button.

   This will bring up a program code window and will create a Visual Basic subroutine which will get executed every time a user clicks on the Reset button. Part of the subroutine code will be created automatically (its definition and exit).

   The cursor is automatically placed at the proper spot in the new subroutine to begin entering commands.

2. Enter the following line of code, which will cause the text string to be set to a new value:

   ```
   Text1.Text = "Text Reset"
   ```

The following window appears:

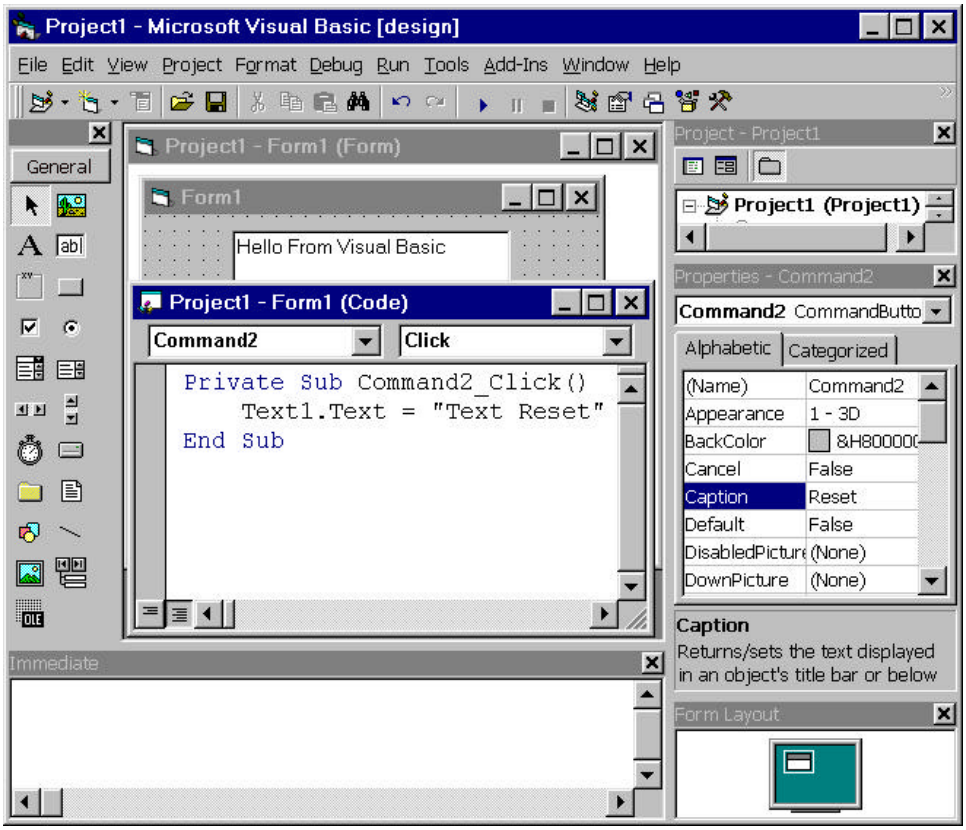


Figure 66. Creating a Visual Basic subroutine for the Reset button

Click on the Close button (the X in the upper right hand corner) of the program code box you just typed in.

Implementing The Hello COBOL Button

The Hello COBOL command button is where most of the action in this application will take place. When a user clicks on this button, Visual Basic will call the HICOBOL.DLL executable and pass two parameters to it.

The COBOL program will update both parameters and they will be passed back. Visual Basic will take the value of the string parameter (which the COBOL program will change to "Hello From COBOL") and update the text box value with it.

You need to perform the following steps at this point:

1. Move the mouse over the Hello COBOL command button and double click on it with the left mouse button to open up a program code window. A new subroutine will automatically be created to execute any time a user selects this button.
2. Enter the following code in the new code window between the Private Sub and End Sub statements:

```
Dim vbInteger as Integer
Dim vbString as String * 16
Call JMPCINT2
Call HICOBOL (vbInteger, vbString)
Text1.Text = vbString
Call JMPCINT3
```

3. Let's look at what each of the above noted statements does at run time:

   Dim vbInteger as Integer - this defines a numeric data item named vbInteger

   Dim vbString as String * 16 - this defines a character string of 16 characters named vbString

Call JMPCINT2 - This calls the supplied Fujitsu COBOL routine which initializes the COBOL run time support for Visual Basic. It need only be called once per application.

Call HICOBOL (vbInteger, vbString) - this calls the COBOL DLL file you previously created, and passes the two parameters to it.

Text1.Text = vbString - this command instructs Visual Basic to update the Text string in the Text1 text box with the value contained in vbString, which will be returned from the HICOBOL COBOL program.

Call JMPCINT3 - This calls the supplied Fujitsu COBOL routine which unloads (exits) the COBOL run time support for Visual Basic. It need only be called once per application.

The following window appears:

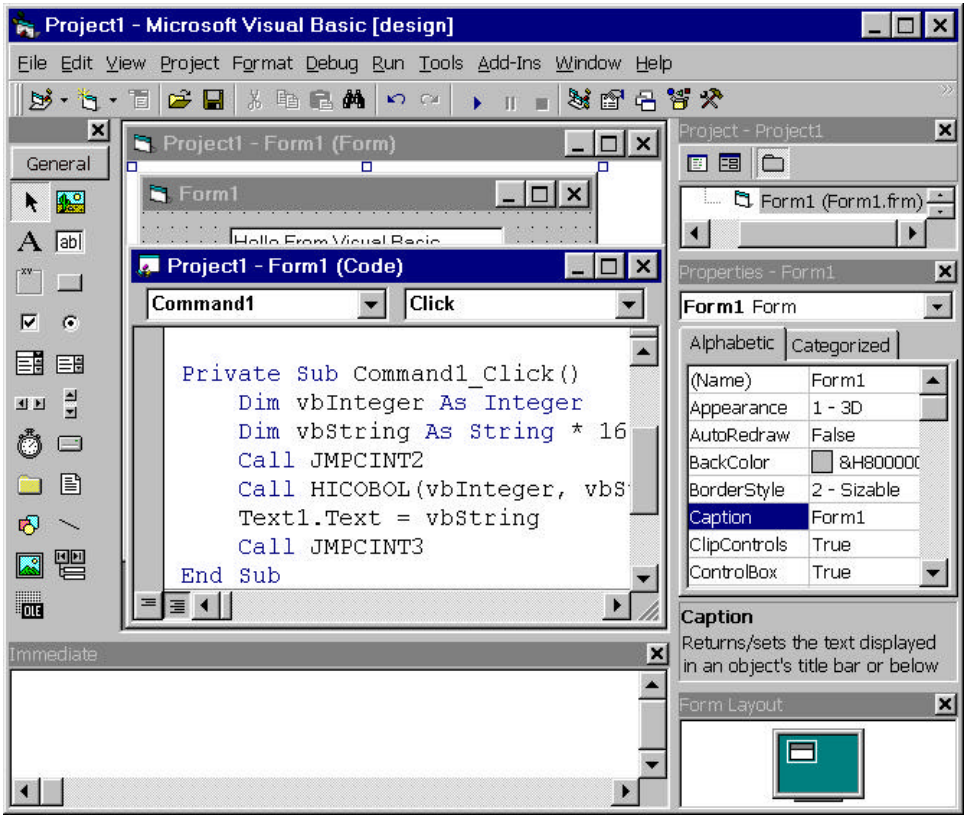


Figure 67. Creating a Visual Basic subroutine for the Hello COBOL button

Defining the COBOL Components to Visual Basic

The final step that needs to be performed is to define to Visual Basic the external COBOL executables that are going to be referenced in this application. Do the following:

1. While the program code window - entitled Project1 Form1 (Code) - is still open (double click on the Hello COBOL command button again if you have closed this window),

Click on the small down arrow to activate the drop down box which currently has the value Command1. This box is located in the upper left of the project code window. It should appear as follows:

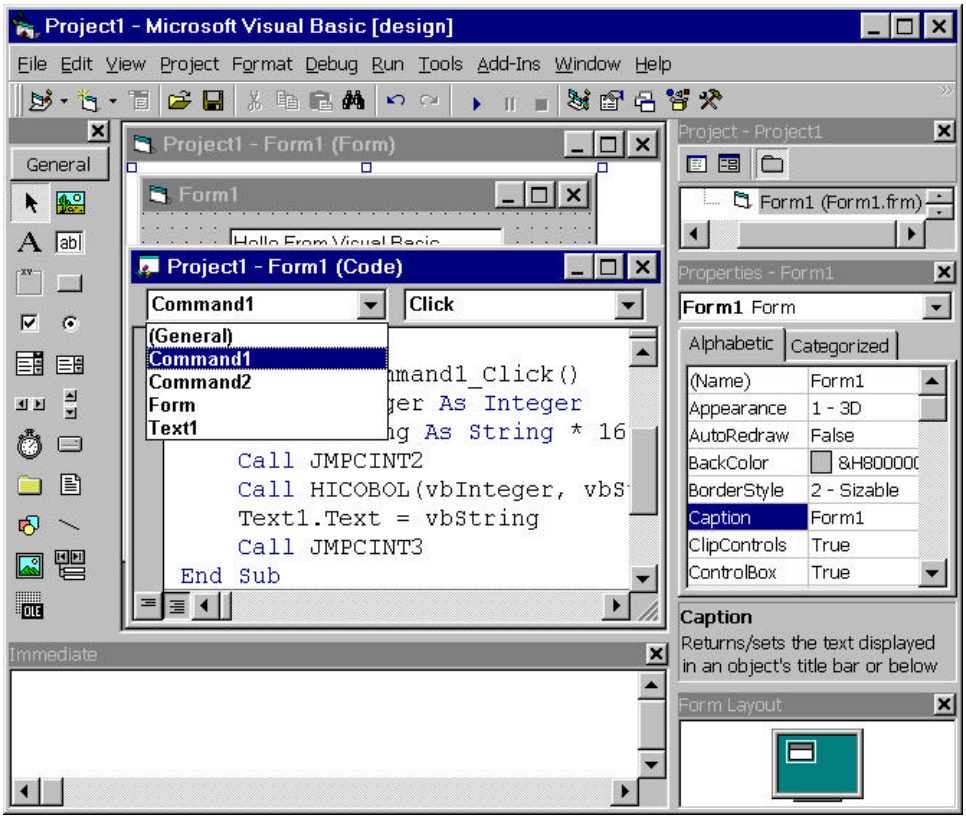


Figure 67. Defined subroutines for the Visual Basic COBOL application

This drop down list box allows you to navigate easily through your program source code for each event sub-routine you have defined. You now need to go to the General section of your program to create declarations for the external COBOL executables. Click on (General).

2. Enter the following 3 statements in any order (as one long string each) in the program code window in the General Section area:

```
Private Declare Sub HICOBOL Lib
"c:\FSC\PCOBOL32\SAMPLES\HICOBOL\HICOBOL.DLL"
(vbInteger as Integer, ByVal vbString as
String)

Private Declare Sub JMPCINT2 Lib
"c:\FSC\PCOBOL32\F3BIPRCT.DLL" ()

Private Declare Sub JMPCINT3 Lib
"c:\FSC\PCOBOL32\F3BIPRCT.DLL" ()
```

The three statements above have been wrapped to multiple lines because of the page size of this manual. They should be entered into the program code window as a single line each (not wrapped).

These statements define to Visual Basic where the external COBOL modules that will be referenced at run time reside. Be sure to alter the path settings above when entering these if you changed the default installation directory when you installed Fujitsu COBOL to reflect your machine's installation location.

The following window appears:

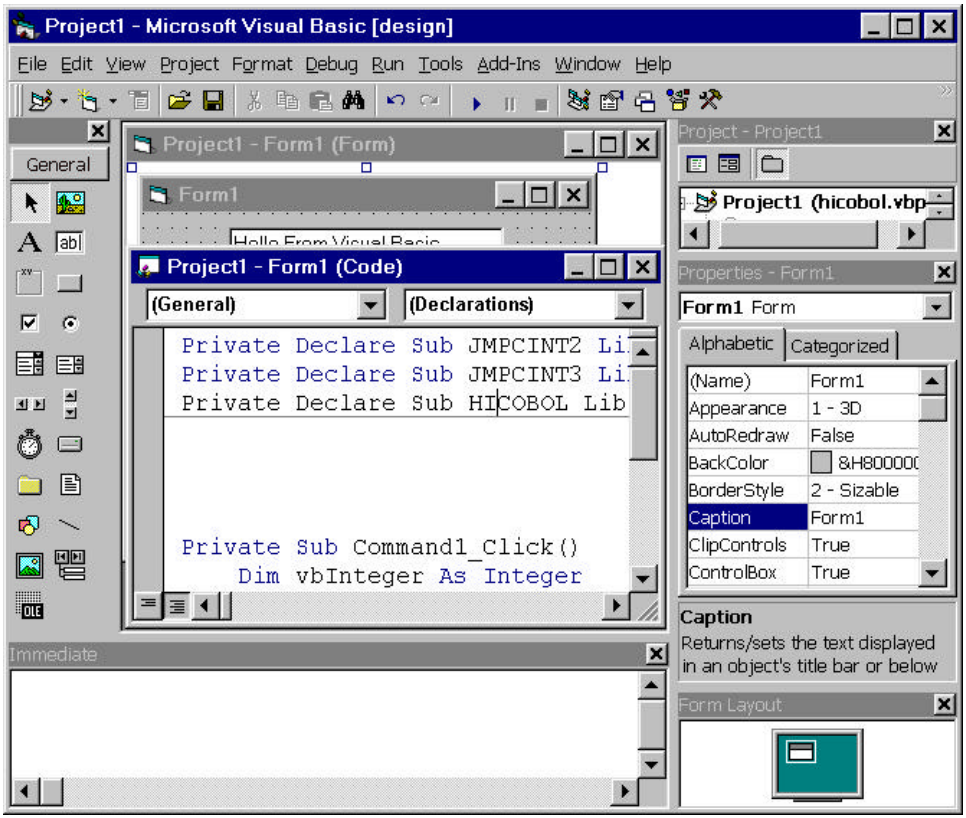


Figure 69. Defining the external COBOL modules in Visual Basic

Close the program code window to save it.

You have now completed all of the steps required to create this sample application. You should save Visual Basic portion you've just created. Click on the File menu near the upper left corner of the main Visual Basic window. Click on Save Project. You will be prompted to save the form. Change the Form name from Form1.frm to HICOBOL.frm, and save it. Likewise, change the project name to HICOBOL.vbp and save it.

Executing the HICOBOL Application under Visual Basic

You are now ready to execute the HICOBOL application. From the Run menu of Visual Basic, select Start.

The HICOBOL graphical window you just created should appear as follows:

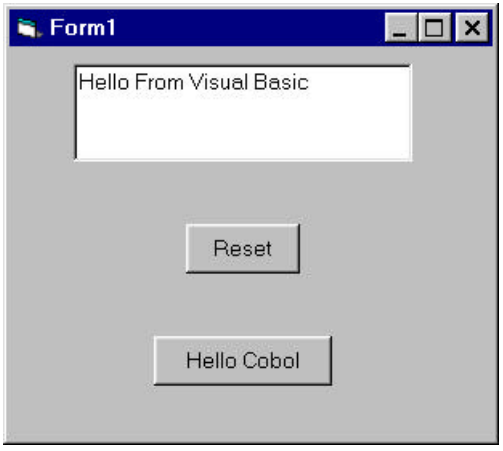


Figure 70. The HICOBOL graphical window

If you click on the Reset command button, the text in the text box should be changed to Text Reset.

If you Click on the Hello COBOL command button, Visual Basic will issue a call to the HICOBOL.DLL executable. The COBOL Run Time Environment Setup window will appear as follows:

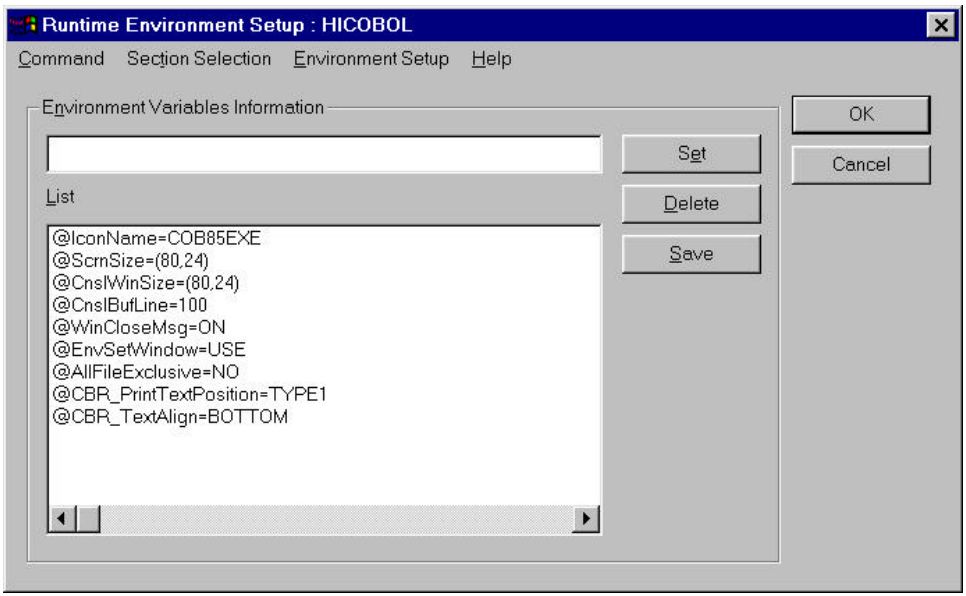


Figure 71. The Run Time Environment Setup window

This window will appear every time HICOBOL.DLL is called. You can turn it off by setting the environment directive EnvSetWindow=UNUSE and saving it. Be advised that this creates or updates a file called COBOL85.CBR and if you turn the run time environment setup window off, you must locate this file and manually edit it to contain the line @EnvSetWindow=USE to turn the window back on. Watch carefully where this file gets saved if you change it and save it.

Click the OK button on the Run Time Environment Setup window. The HICOBOL.DLL program will be called, will update the parameters, and the text box in your graphical window should be updated to read "Hello From COBOL."

If you click on the Reset command button, the text in the text box should be reset to Text Reset.

Closing the window exits the application.

If you desire to create a standalone EXE for your application, use the Make command under the Visual Basic File menu. Note that Visual Basic applications require you to distribute the Visual Basic Runtime DLL along with your EXE and the COBOL components required.

If you have additional questions related to Visual Basic COBOL applications, refer to Chapter 5, “Integrating COBOL Programs with Visual Basic.”

Chapter 5. Integrating COBOL Programs with Visual Basic

This chapter contains detailed information regarding the integration of Fujitsu COBOL programs to Visual Basic applications.

COBOL DLL Interaction with Visual Basic

To access any COBOL routines from Visual Basic, you need to first declare the COBOL PROGRAM-ID or the COBOL ENTRY routine using the Basic DECLARE statement with the special LIB keyword to specify the path name of the COBOL DLL. These declarations can be made in the declaration section of any form module or in the global module, and must be declared as Private. You can then call these routines from your Visual Basic code like any other function call.

When a Visual Basic program calls a COBOL routine, you should call JMPCINT2 before calling the first COBOL routine, and call JMPCINT3 after calling the last COBOL routine. JMPCINT2 is a subroutine that initializes the COBOL Run Time Environment and JMPCINT3 is a subroutine that exits the COBOL Run Time Environment.

Calling COBOL routines without calling JMPCINT2 and JMPCINT3 may degrade performance because the COBOL Run Time Environments will be initialized and exited on every COBOL routine called.

Visual Basic Declarations of COBOL Modules

```
Private Declare Sub PROG Lib "c:\mycobol.dll" (vbInteger as Integer,
ByVal vbString as String)

Private Declare Sub JMPCINT2 Lib "c:\FSC\PCOBOL32\F3BIPRCT.DLL" ()

Private Declare Sub JMPCINT3 Lib "c:\FSC\PCOBOL32\F3BIPRCT.DLL" ()
```

Visual Basic Call to a COBOL Program

```
Sub Form_Click ( )
Dim vbInteger as Integer
Dim vbString as string * 15
      Call JMPCINT2  ‘ Initialize COBOL Runtime Environment
Call PROG (vbInteger, vbString)
Call JMPCINT3 ‘ Terminate COBOL Runtime Environment
End Sub
```

COBOL LINKAGE SECTION and PROCEDURE DIVISION

```
Identification Division.
Program-ID.   "PROG".
Data Division.
Linkage Section.
01  vbInteger pic s9(4) comp-5.
01  vbString  pic x(15).
Procedure Division with STDCALL Linkage Using vbInteger, vbString.
move 100 to vbInteger
move "Fujitsu COBOL" to vbString
exit program.
```

Parameter Passing Between Visual Basic and Fujitsu COBOL

Visual Basic treats a COBOL DLL as a “black box.” It only needs to know the COBOL program's LINKAGE SECTION parameter list to pass data to and from the COBOL DLL. Parameters can only be passed from Visual Basic to COBOL BY REFERENCE, except strings which must be passed using the ByVal keyword in the DECLARE statement. Passing parameters BY REFERENCE is the default in Visual Basic. Parameter names need not be the same between Visual Basic and COBOL, however, attribute, length, and number of corresponding data items must be identical.

Visual Basic declarations must exactly match the COBOL parameter lists defined in the USING statement of the COBOL PROCEDURE DIVISION or ENTRY statements. No parameter checking is done because the two are separate compilation units.

Visual Basic incorporates a rich assortment of data types, some of which are currently not supported by Fujitsu COBOL. These data types include variable-length strings, *Currency*, *Variants*, and objects.

Visual Basic Strings

All strings passed between Visual Basic and COBOL must be declared as fixed-length strings and passed using the ByVal keyword in the DECLARE statement. To avoid possible memory corruption, ensure that all strings passed between Visual Basic and COBOL occupy the same size.

```
Dim vbString as string * 15     ‘ Equivalent to PIC X(15)
```

Visual Basic Correspondence Data Types

When passing parameters, associate the data types as follows:

Table 1. Visual Basic correspondence data types

Visual Basic		**COBOL**
Type Name	**Storage Size**	**PICTURE Clause**
Boolean (16bit)	2 Bytes	S9(4) COMP-5
Boolean (32bit)	4 Bytes	S9(9) COMP-5
Byte	1 Byte	X
Double	8 Bytes	COMP-2
Integer	2 Bytes	S9(4) COMP-5
Long	4 Bytes	S9(9) COMP-5
Single	4 Bytes	COMP-1
String	1 Byte per Character	X(n)

Passing Arrays

Visual Basic and COBOL can pass numeric arrays. This works because numeric array data is always laid out sequentially in memory. A COBOL routine, if given the first element of an array, has access to all of its elements.

Returning Control and Exiting Programs

To return control from COBOL to Visual Basic, execute the EXIT PROGRAM statement. When the EXIT PROGRAM statement is executed, control returns immediately to the calling program.

Compiling and Linking the COBOL Programs

To build the COBOL DLL, a module definition file (.DEF) that lists the attributes of the DLL library must be created. A proper DEF file will be automatically created by P-STAFF during the build process, or you can modify the following example below changing the MYCOBOL to match the COBOL PROGRAM-ID.

Sample COBOL.DEF File

A DEF file is required by the linker to produce a valid DLL executable. The following is an example of a valid DEF file for a COBOL DLL.

```
LIBRARY  "PROG"
EXPORTS  PROG
```

Chapter 6. COBOL 32 Bit Run-time Files

This chapter lists the 32 bit run-time files required by Fujitsu COBOL.

The following table lists the Fujitsu COBOL 32 bit run-time files.

Table 2. The Fujitsu COBOL 32 bit run-time files

DLL Name	Description
F3BICMSE.DLL	A resource file for the Run-time Environment Setup window (English)
F3BICMSJ.DLL	A resource file for the Run-time Environment Setup window (Japanese)
F3BIBTRV.DLL	DLL for accessing Btrieve using READ/WRITE statements
F3BICICL.DLL	A client runtime system for simplified inter_application communication function
F3BICMUE.DLL	Message resources for the simplified inter_application communication function destination definition utility (COBCIU32.EXE) (English)
F3BIDBG.DLL	Interactive Debugging (TEST) option
F3BIETBP.DLL	One of the DLLs required for the mainframe presentation file test function
F3BIFLTE.DLL	Floating runtime system
F3BIFRM.DLL	COBOL85 File Manager
F3BIIFNC.DLL	A DLL for intrinsic functions
F3BIFUPE.DLL	One of the COBOL85 File Utility DLLs (English)
F3BIFUTC.DLL	One of the COBOL85 File Utility DLLs (Recovery function)
F3BIFUTE.DLL	One of the COBOL85 File Utility DLLs (Messages)
F3BIILNG.DLL	A DLL for a mixed language environment
F3BIICON.DLL	Runtime icon
F3BIIO.DLL	I/O runtime DLL
F3BILANP.DLL	License control DLL
F3BILCL.DLL	One of the presentation file function DLLs
F3BILCLM.DLL	One of the presentation file function DLLs (Destination DSP)
F3BILKTC.DLL	Interactive debugging runtime system
F3BILPIO.DLL	Printing runtime system DLL
F3BIMSGE.DLL	Runtime system messages (English)
F3BINUC.DLL	Nucleus runtime system DLL
F3BIEMSG.DLL	One of the DLLs for the mainframe presentation file test function

Table 2. The Fujitsu COBOL 32 bit run-time files (cont.)

DLL Name	Description
F3BIODSJ.DLL	A message DLL for the ODBC function (Japanese)
F3BIODBC.DLL	ODBC run-time system DLL for embedded SQL
F3BIODSE.DLL	A message DLL for the ODBC function (English)
F3BIOVD0.DLL	DLL for overlay printing
F3BIOVD1.DLL	DLL for overlay printing
F3BIOVD2.DLL	DLL for overlay printing
F3BIOVDE.DLL	DLL for overlay printing
F3BIOVDF.DLL	DLL for overlay printing
F3BIOVDG.DLL	DLL for overlay printing
F3BIOVDL.DLL	DLL for overlay printing
F3BIOVLD.DLL	DLL for overlay printing
F3BIPRCT.DLL	Run-time Unit/Environment Control DLL for including ACCEPT/DISPLAY statements
F3BIPREHE.HLP	Help file for Run-time Environment Setup window - not needed if window is not shown (English)
F3BIPREHJ.HLP	Help file for Run-time Environment Setup window - not needed if window is not shown (Japanese)
F3BIPRIO.DLL	Required when using presentation files
F3BIPRSE.DLL	A resource file for the Run-time Environment Setup window (English)
F3BISCRN.DLL	SCREEN SECTION runtime system DLL
F3BISMDA.DLL	One of the DLLs for the mainframe presentation file test function
F3BISORT.DLL	COBOL SORT
F3BISQL.DLL	A control DLL for embedded SQL
F3BISQLK.DLL	A DLL for SequeLink access for embedded SQL
F3BISTMG.DLL	SORT/MERGE statement runtime system DLL
WBTRCALL.DLL	Btrieve DLL licensed by Pervasive Software (**See Note below**)

Note: You must obtain a valid license from Pervasive Software in order to use Btrieve. This file is required only to access the Btrieve database.

Appendix A. Sample Programs

The sample programs shipped with Fujitsu COBOL are intended to give an overview of the capabilities of COBOL85 and P-STAFF. Refer to the "COBOL85 User's Guide" for further details on using Fujitsu COBOL. Table 3 shows the sample programs available with Fujitsu COBOL 3.0.

Table 3. Fujitsu COBOL Sample Programs

	Functions Used	**File Name**
Sample 1	- ACCEPT/DISPLAY (console window) - Debugger	SAMPLE1.COB (COBOL source program) SAMPLE1.SVD (debugging information file) DBGSAMP1.EXE (executable program for debugging)
Sample 2	- Line sequential file (reference) - Indexed file (creation)	SAMPLE2.COB (COBOL source program) DATAFILE (data file)
Sample 3	Screen Input-Output Operation Using the Presentation File Function	Available in the Fujitsu COBOL Enterprise Edition
Sample 4	- Screen handling - Indexed file (reference)	SAMPLE4.COB (COBOL source program)
Sample 5	- Project management - Inter-program communication - Library fetch - ACCEPT/DISPLAY (message box) - Print file - Indexed file (reference) - Line sequential file (creation) - Passing parameters for execution - Free format	SAMPLE5.COB (COBOL source program) PRINTPRC.COB (COBOL source program) PRINTPRC.DEF(module definition file) S_REC.CBL (library text) SAMPLE5.PRJ (project file) SAMPLE5.CBI (option file for compilation) SAMPLE5.LNI (option file for Linkage)

Table 3. Fujitsu COBOL Sample Programs (cont.)

	Function Used	**File Name**
Sample 6	- Fetching command line arguments - Internal program	SAMPLE6.COB (COBOL source program)
Sample 7	- Environment variable handling	SAMPLE7.COB (COBOL source program)
Sample 8	- Print file - ACCEPT/DISPLAY (console window)	SAMPLE8.COB (COBOL source program)
Sample 9	- Inter_application communication	SAMPLE9.COB (COBOL source program) TSUUSHIN.COB (COBOL source program) PRM_REC.CBL (Registration collection original) SAMPLE9.INI (Logical destination definition file)
Sample 10	- Remote database access	SAMPLE10.COB (COBOL source program)

Sample 1: Data Processing Using Standard Input-Output Destination

Sample 1 shows a program that inputs or outputs data from or to the console window using the COBOL ACCEPT/DISPLAY function. Refer to the "COBOL85 User's Guide" for details on how to use the ACCEPT/DISPLAY function.

Function

Inputs an alphabetic character (lowercase character) from the console window, and outputs a word beginning with the input alphabetic character to the console window.

COBOL Statements Used

ACCEPT statement, DISPLAY statement, EXIT statement, IF statement, and PERFORM statements are used.

Program Compilation

Refer to the "COBOL85 User's Guide."

Program Linkage

Refer to the "COBOL85 User's Guide."

Program Execution

Refer to the "COBOL85 User's Guide."

Program Debugging

Sample 1 supports a debugging information file and the executable program (DBGSAMP1.EXE) created by specifying the debug option. To start Sample 1 using the interactive debugger, do the following:

1. Select WINSVD [Debug] from the Tools menu in P-STAFF.
2. Select Start Debugging from the File menu.

The Start Debugging dialog box opens. Specify the executable program name (DBGSAMP1.EXE) on the Command line edit box. Alternatively, you can use the Browse button to navigate to the directory where the file is stored. Click on the OK button.

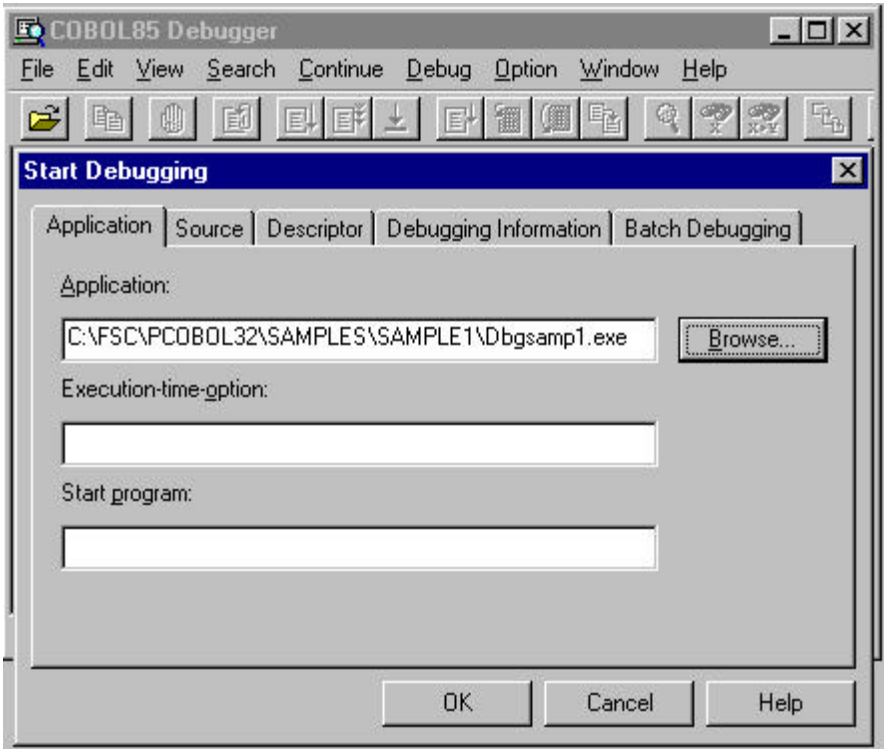


Figure 72. The Start Debugging dialog box

3. The Run-time Environment Setup window is displayed. Sample 1 does not require specification of run-time environment information.

   Click on the OK button to return to the COBOL85 Debugger window.

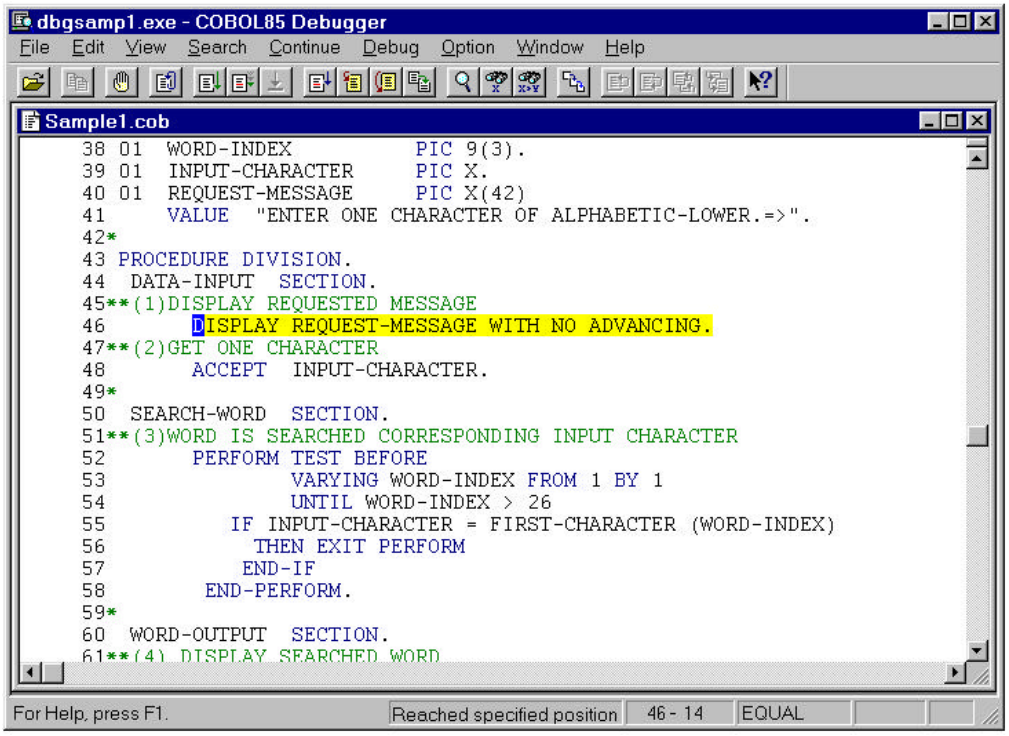


Figure 73. The COBOL85 Debugger window

4. Step through the program.

   The program automatically stops at the first executable statement (Line 46). Select Step Into from the Continue menu or press the F8 key to step (advance) to the next executable statement (Line 48). Step again to execute the Accept statement on Line 48.

   Note: The Accept statement requires user input.

5. Input the data name.

   Make the console window active. **Note**: The console window may be hidden behind the COBOL85 Debugger window. Type in any lowercase alphabetic character and then press the ENTER key. The debugger automatically returns to the next executable statement (Line 52).

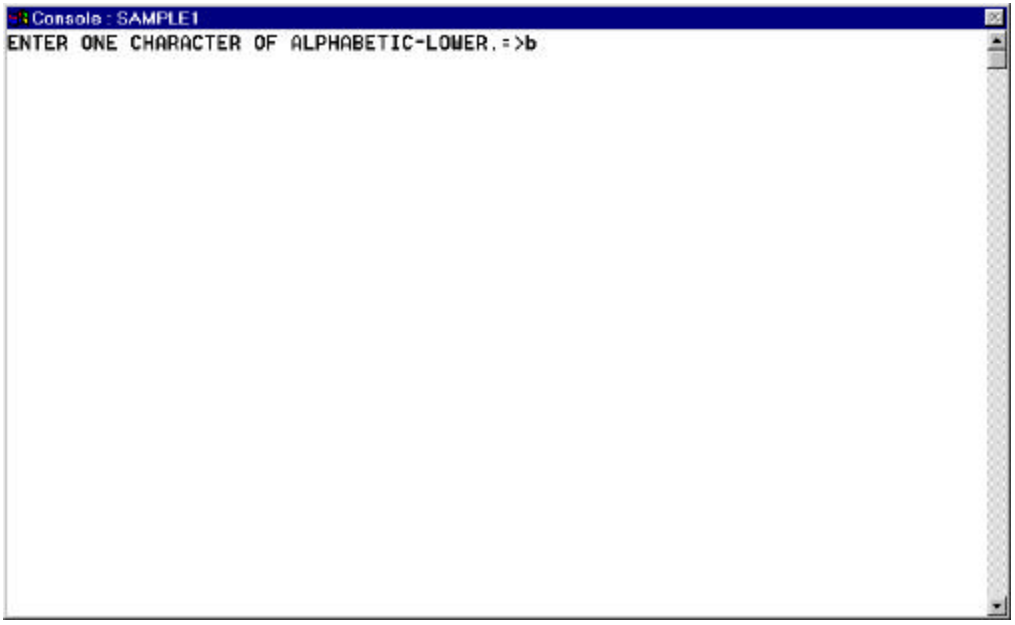


Figure 74. The console window

6. Change the data name (input-character) to "A".

   Move the cursor to Line 48. Double click on INPUT-CHARACTER to highlight it. Select Data from the Debug menu. Alternatively, you can click on the right mouse button to select Data in the shortcut menu. The Data dialog box is displayed.

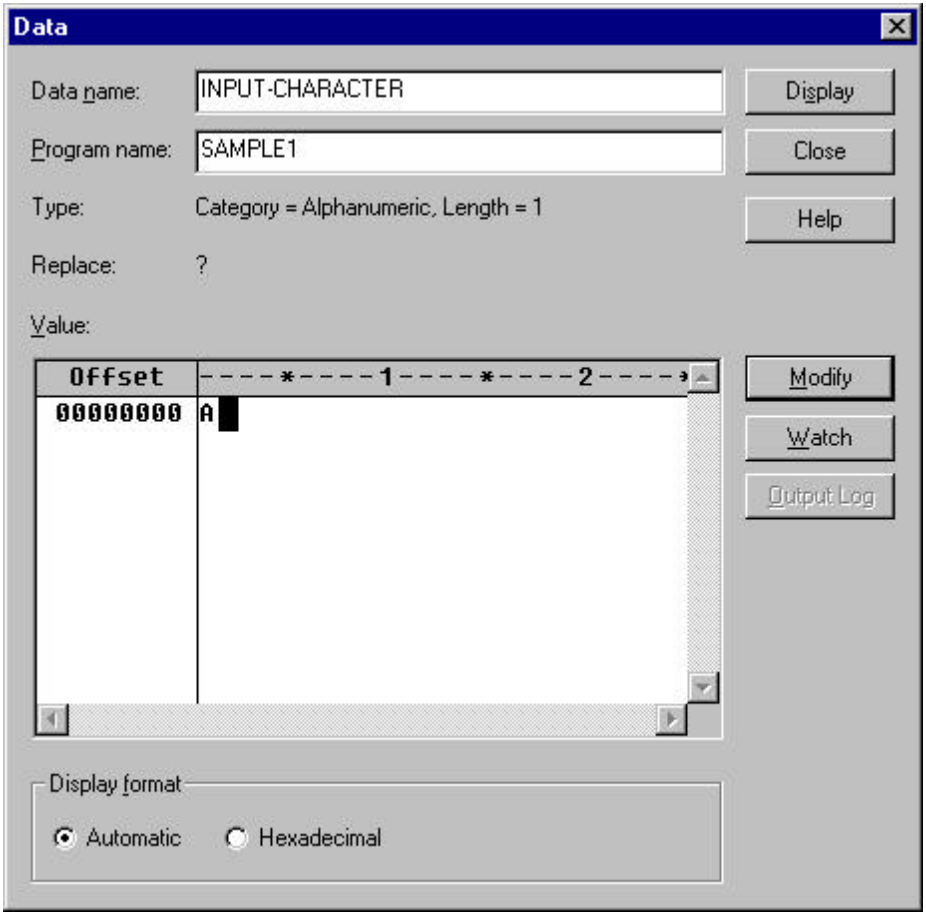


Figure 75. The Data dialog box

Change the value of the data in the data value area and click on the Modify button. Click on the Close button to exit the Data dialog box.

7. Specify a breakpoint.

   Move the cursor to Line 62 and click on Breakpoint in the Debug menu. The Breakpoint dialog box is displayed. Click on the Set button to set the breakpoint, then click on the Close button to exit the Breakpoint dialog box and return to the main window.

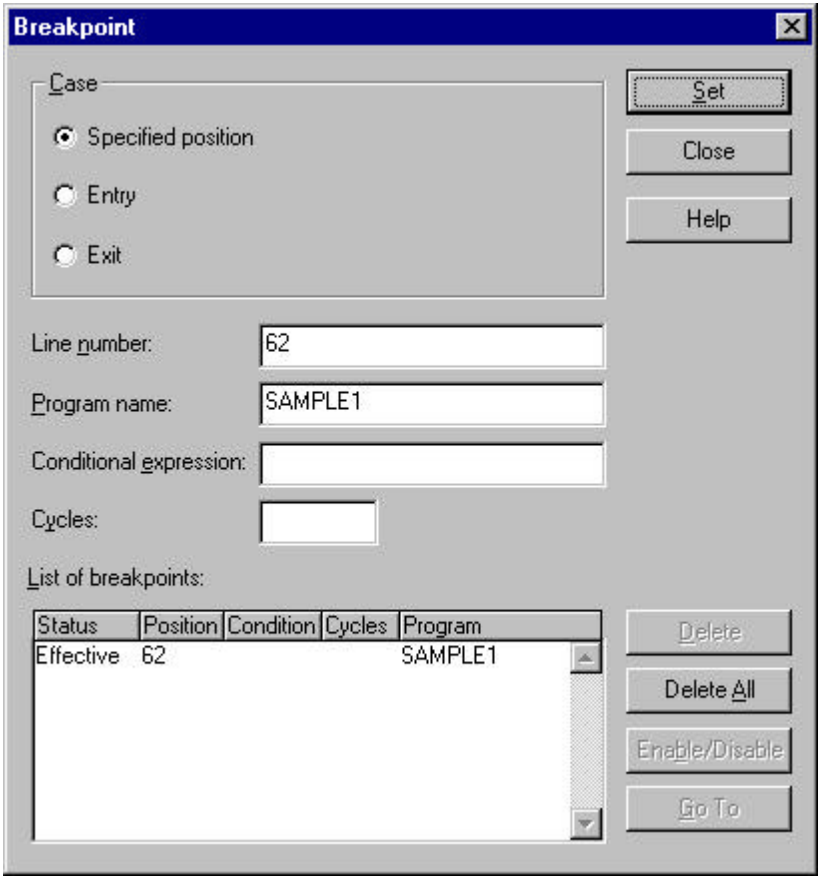


Figure 76. The Breakpoint dialog box

Alternatively, you can click on the right mouse button and select Set Breakpoint from the shortcut menu. A breakpoint is set but the Breakpoint window is not displayed.

Note that the statement changes color when the breakpoint has been applied.

To delete a breakpoint, go back to the Breakpoint dialog box and highlight the breakpoint you want to delete. Click on the Delete button, then click on the Close button to exit the Breakpoint dialog box. **Note**: You can delete more than one breakpoint at a time by highlighting multiple items and clicking on the Delete button.

8. Execute the program up to the breakpoint.

   Select Go in the Continue menu to execute the program to the breakpoint. Alternatively, you can press the F5 key. The program stops at Line 62.

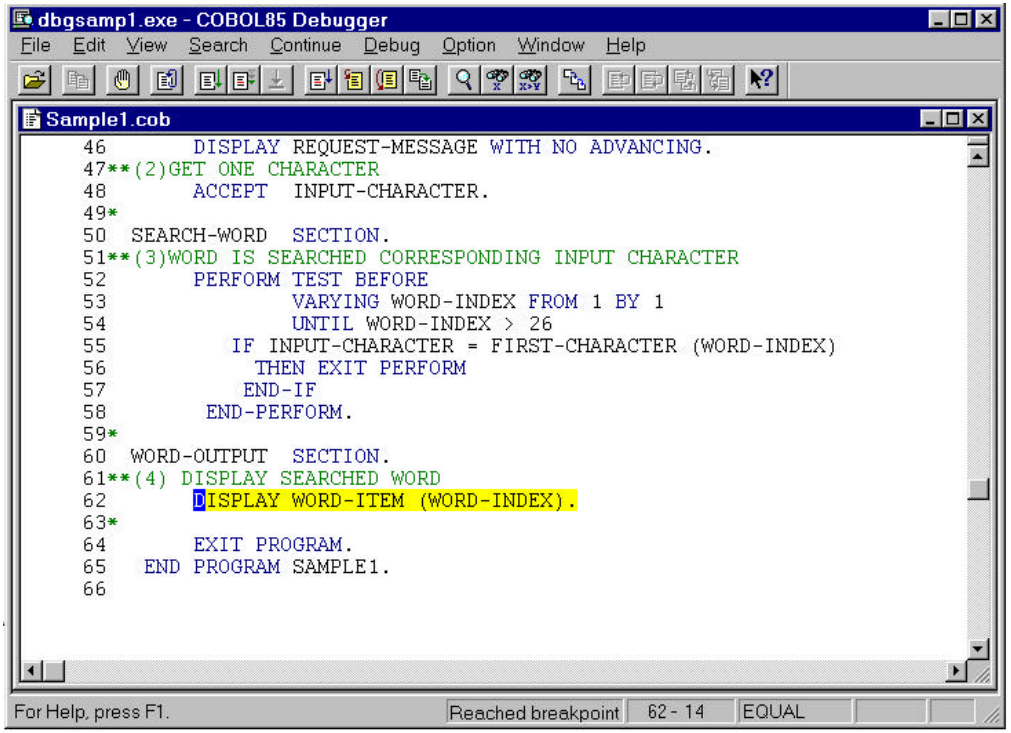


Figure 77. Executing the program up to a breakpoint

Select Go or press the F5 key again to complete the debugging session.

7. To exit the debugger, select Exit from the File menu.

Sample 2: Operating Line Sequential Files and Indexed Files

Sample 2 shows a program that reads a data file (line sequential file) created with the editor, then creates a master file (indexed file). For details on how to use line sequential files and indexed files, refer to the “COBOL85 User’s Guide”.

Function

Reads a data file (line sequential file) storing product codes, product names, and unit prices, and creates an indexed file with the product code as a primary record key and the product name as an alternate record key.

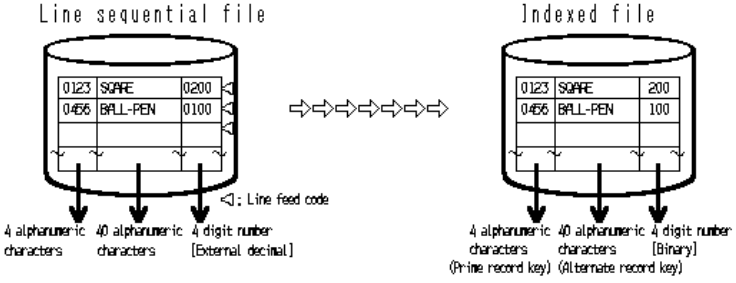


Figure 78. Creating an indexed file from a line sequential file

COBOL Statements Used

CLOSE statement, EXIT statement, GO TO statement, MOVE statement, OPEN statement, READ statement, and WRITE statements are used.

Program Compilation

The following figures show the setup items of the WINCOB window and the Compiler Options dialog box if C:\FSC is specified as the Fujitsu COBOL installation directory:

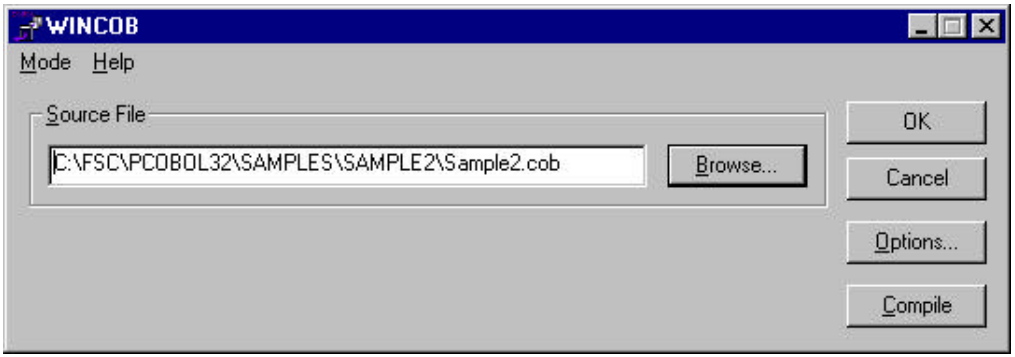


Figure 79. The WINCOB window

Specify the main program to the compiler option MAIN. Refer to Chapter 3, “A Quick Tour” for details on specifying compiler options.

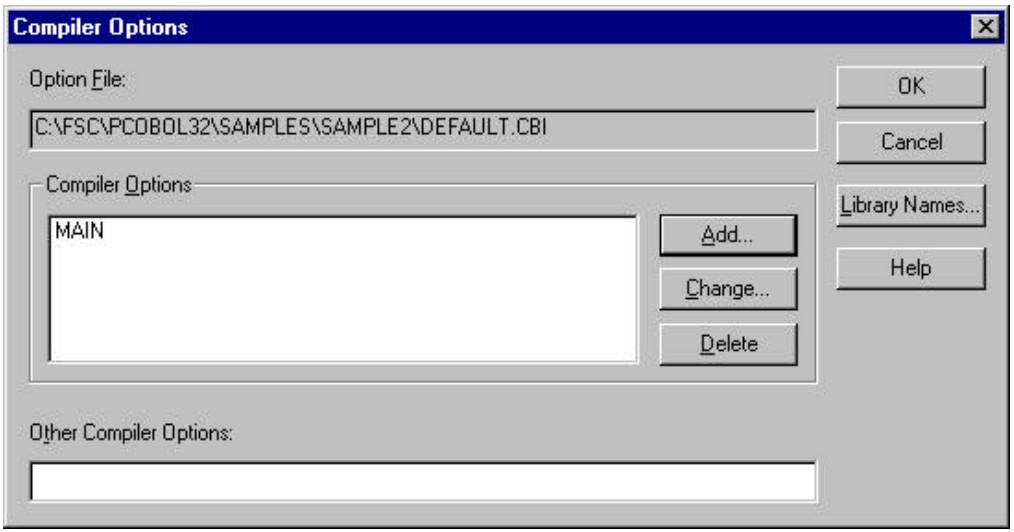


Figure 80. The Compiler Options dialog box with compiler option MAIN specified

Program Linkage

The following figure shows the setup items of the WINLINK [Linking Files] window for linking the object program (SAMPLE2.OBJ) created through compilation.

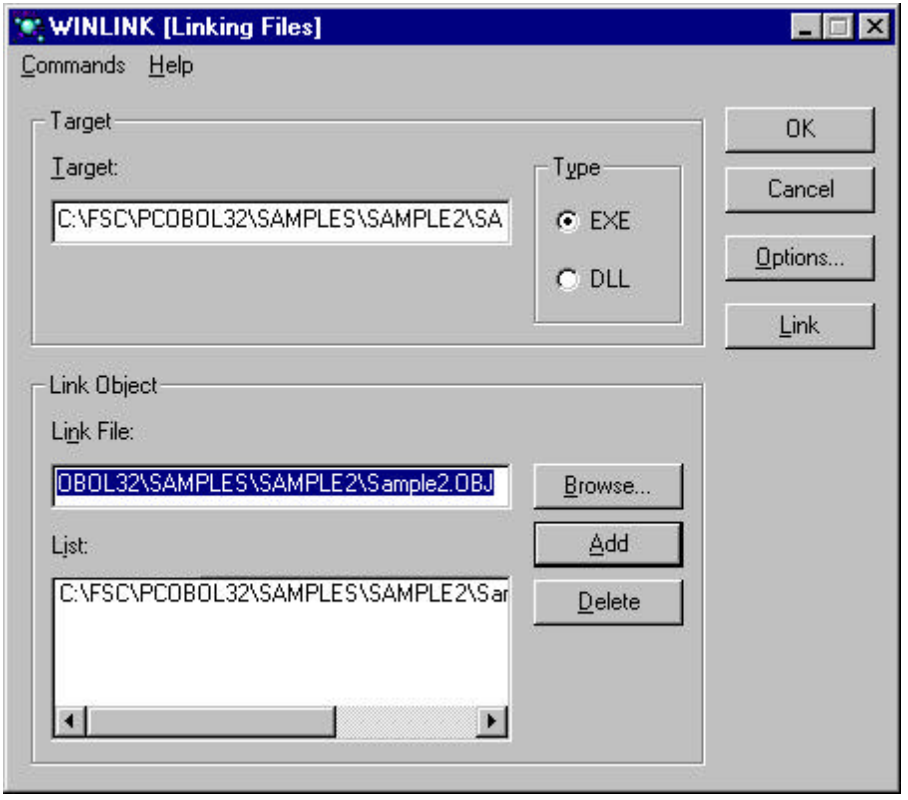


Figure 81. The WINLINK [Linking Files] window

A target name is displayed when an object file is added. Refer to Chapter 3, “A Quick Tour” for details on using WINLINK.

Program Execution

The following figures show the setup items of the WINEXEC window and the Run-time Environment Setup window for executing the executable program (SAMPLE2.EXE) created through linkage.

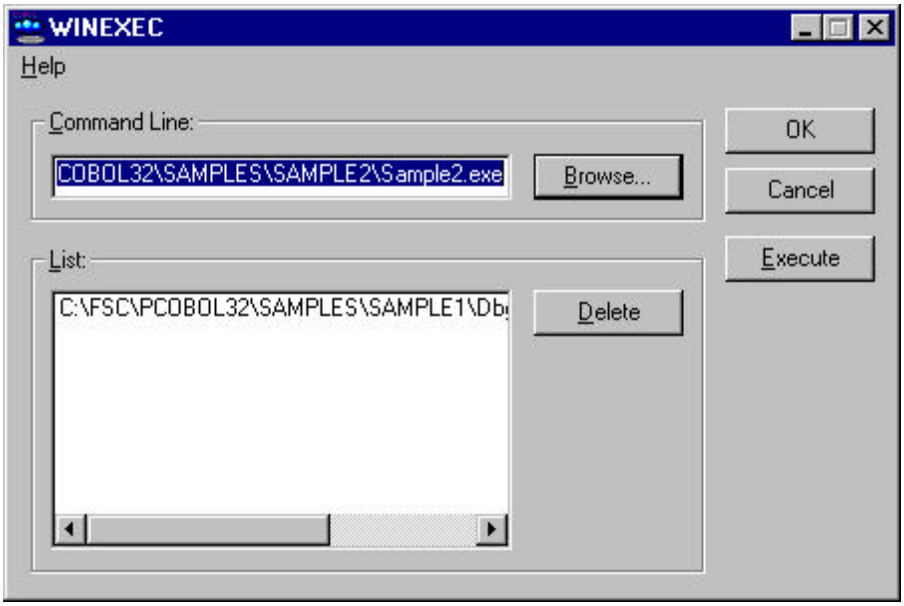


Figure 82. The WINEXEC window

The List box lists execution history information. The information is unrelated to execution of the program in Sample 2. Refer to Chapter 3, “A Quick Tour” for details on using WINEXEC.

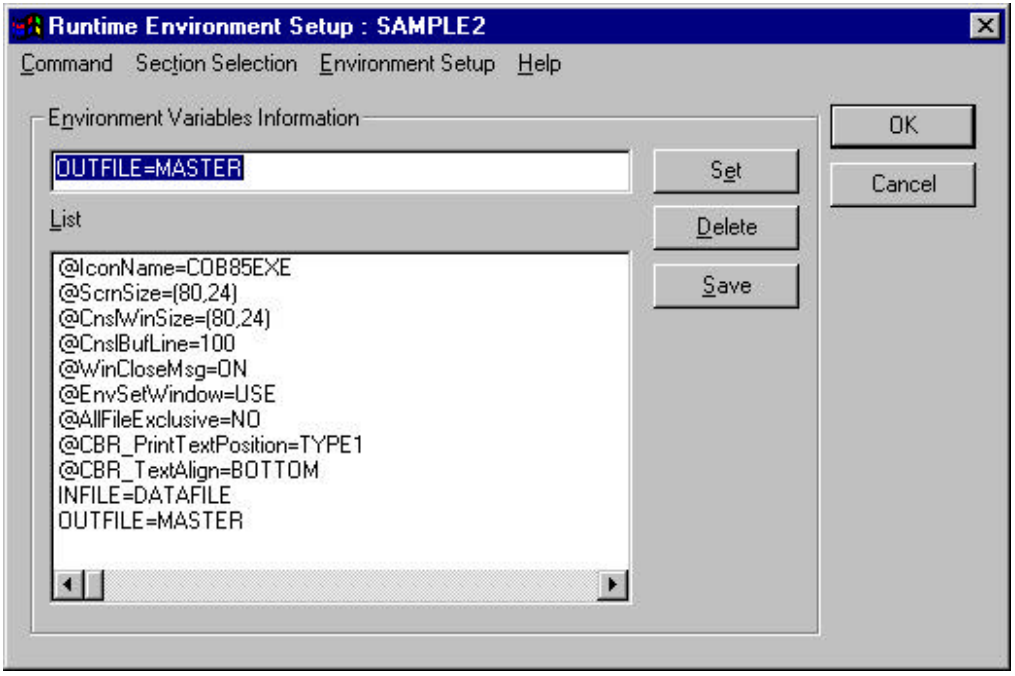


Figure 83. The Run-time Environment Setup window

Note: The file-identifiers INFILE and OUTFILE are file-reference-identifiers specified in the ASSIGN clause of the COBOL source program. The file-reference-identifiers are used to associate COBOL programs with actual media (files). Refer to the "COBOL85 User's Guide" for details on how to set file identifiers.

Execution Result

The termination message is not displayed in the window.

After completion of execution, an indexed file (MASTER) with a product code as a key is created in the directory of the program in Sample 2. Use Windows Explorer or File Manager to check whether the indexed file was created.

Sample 3: Screen Input-Output Operation Using the Presentation File Function

Sample 3 is available in the Fujitsu COBOL Enterprise Edition.

Sample 4: Screen Input-Output Operation Using the Screen Handling Function

Sample 4 shows a program that inputs or outputs data to the screen using the screen handling function. Refer to the "COBOL85 User's Guide" for details on how to use the screen handling function.

Function

If an employee's number and name are written to the screen, the program creates an indexed file with the employee's number as a primary record key and the name as an alternate record key.

COBOL Statements Used

ACCEPT statement, CLOSE statement, DISPLAY statement, EXIT statement, GO TO statement, IF statement, MOVE statement, OPEN statement, and WRITE statements are used.

Program Compilation

The following figures show the setup items of the WINCOB window and the Compiler Options dialog box if C:\FSC is specified as the COBOL85 installation directory.

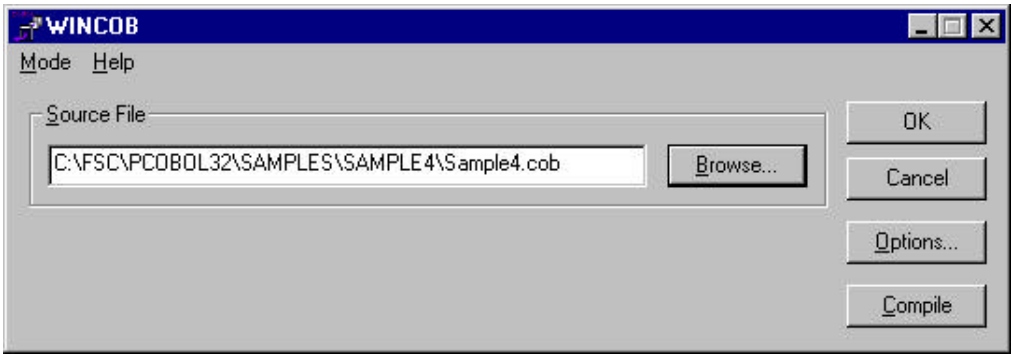


Figure 84. The WINCOB window

Specify the main program to the compiler option MAIN. Refer to Chapter 3, “A Quick Tour” for details on specifying compiler options.

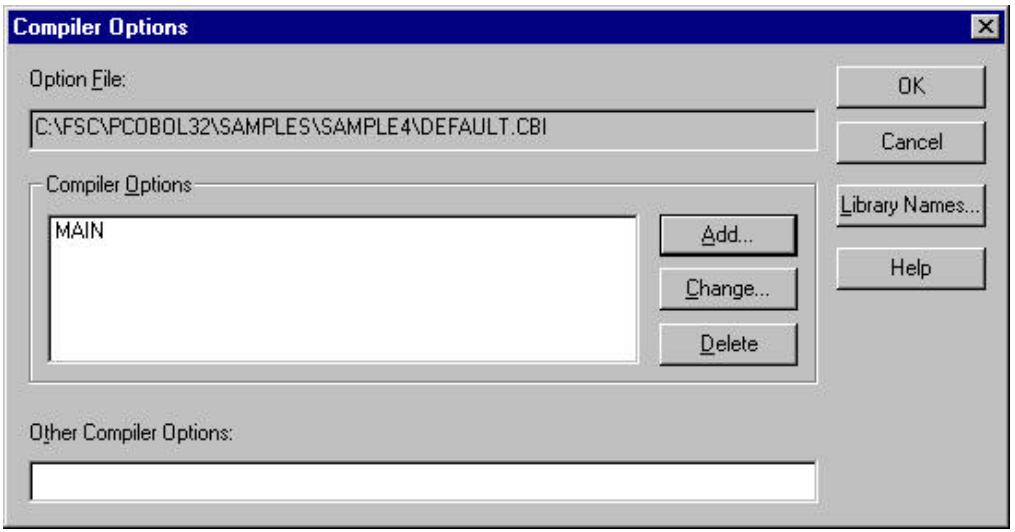


Figure 85. The Compiler Options dialog box with compiler option MAIN specified

Program Linkage

The following figure shows the setup items of the WINLINK [Linking Files] window for linking the object program (SAMPLE4.OBJ) created through compilation.

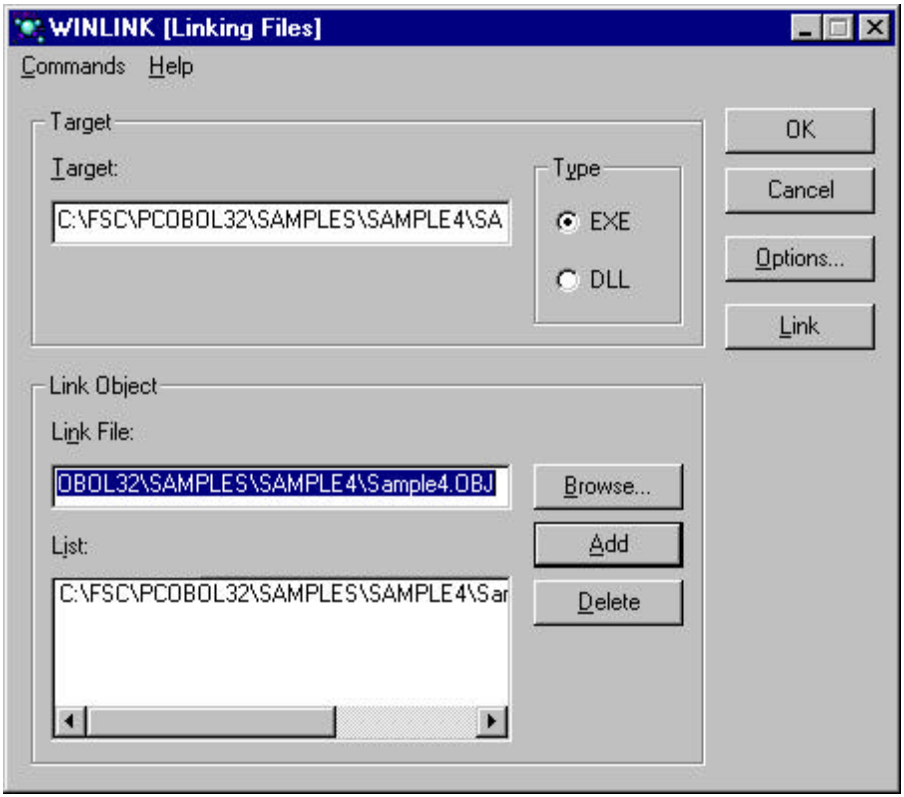


Figure 86. The WINLINK [Linking Files] window

A target name is displayed when an object file is added. Refer to Chapter 3, “A Quick Tour” for details on using WINLINK.

Program Execution

The following figures show the setup items of the WINEXEC window and the Run-time Environment Setup window for executing the executable program (SAMPLE4.EXE) created through linkage.

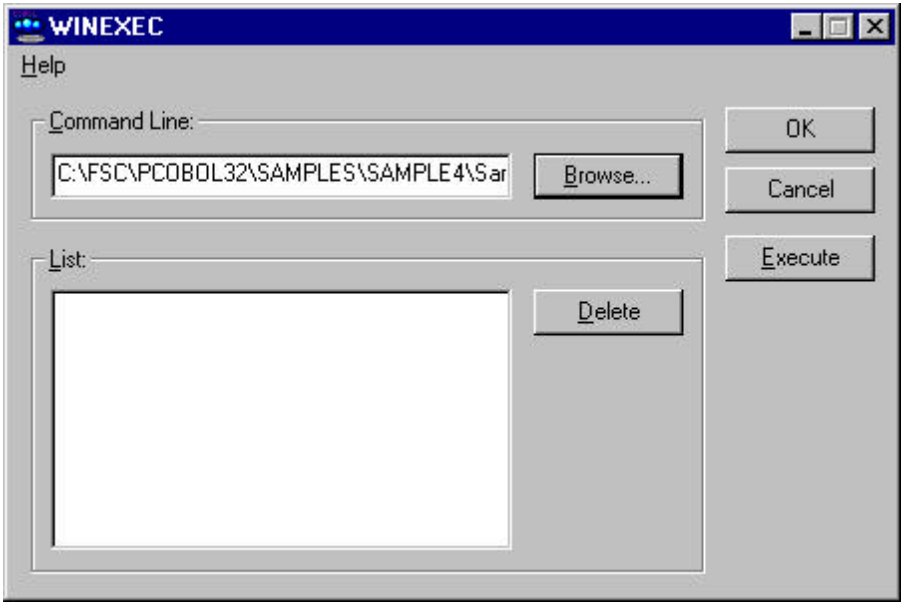


Figure 87. The WINEXEC window

Refer to Chapter 3, “A Quick Tour” for details on using WINEXEC.

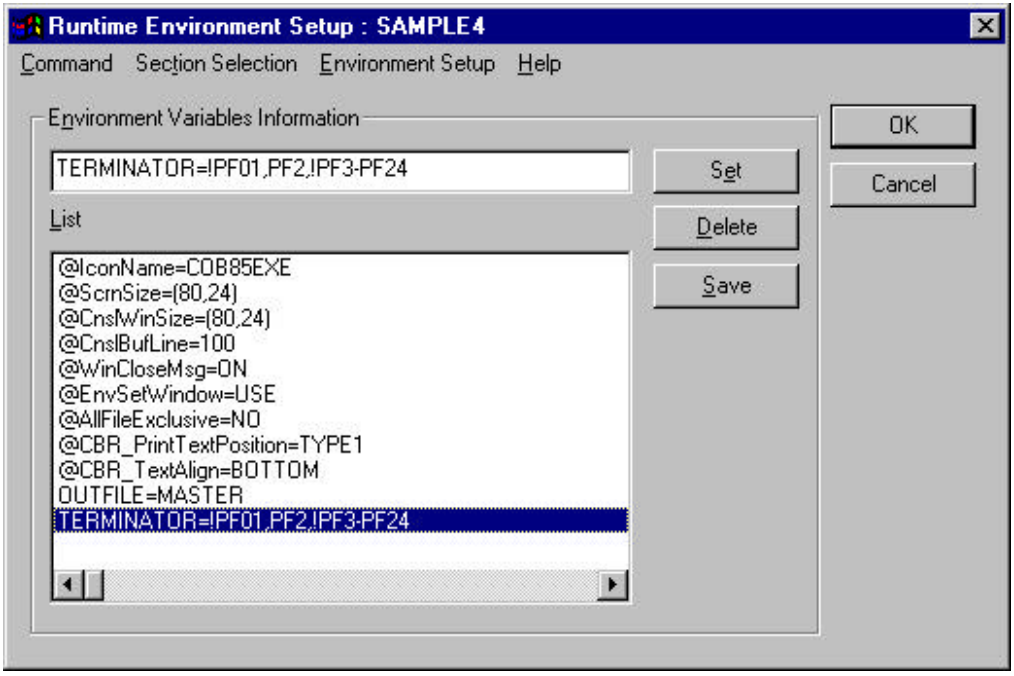


Figure 88. The Run-time Environment Setup window

Add the following environment variables to the list:

OUTFILE=MASTER

TERMINATOR=!PF01,PF2,!PF3-PF24

To add an environment variable to the list, do the following:

1. Type the environment variable to be added in the Environment Variables Information edit box.
2. Click on the Set button to add the environment variable to the list.

Execution Procedure

Click on the OK button of the Run-time Environment Setup window. The screen for entering an employee's number and name is displayed.

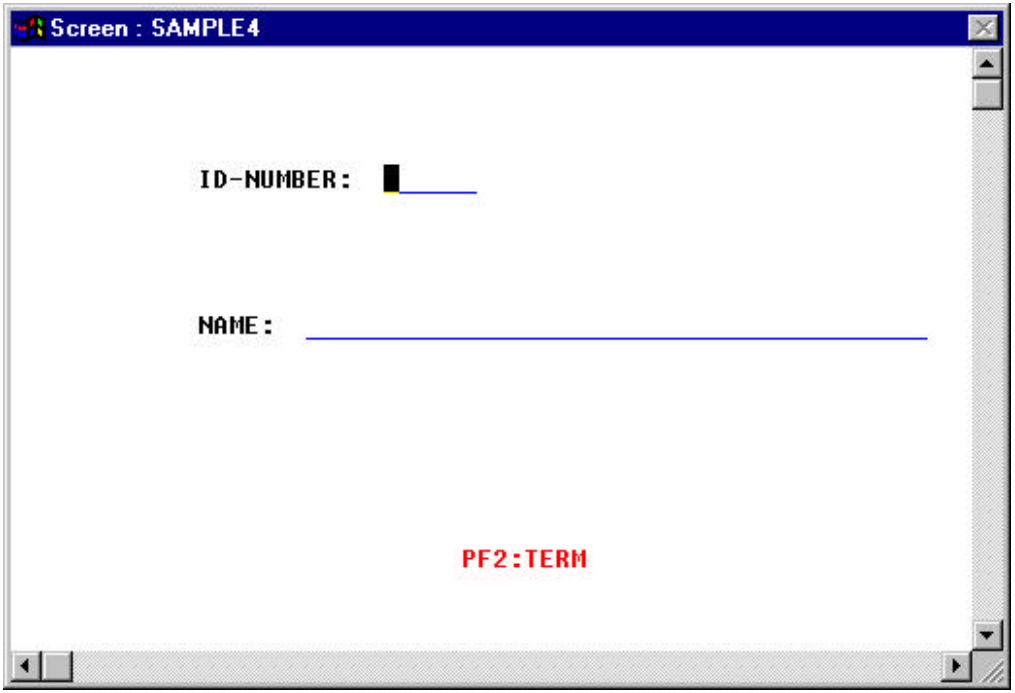


Figure 89. Sample 4 employee number and name input screen

Input the data:

Enter an employee's number (6 digit number) and name (up to 40 alphanumeric characters). Employee numbers must be entered in ascending order.

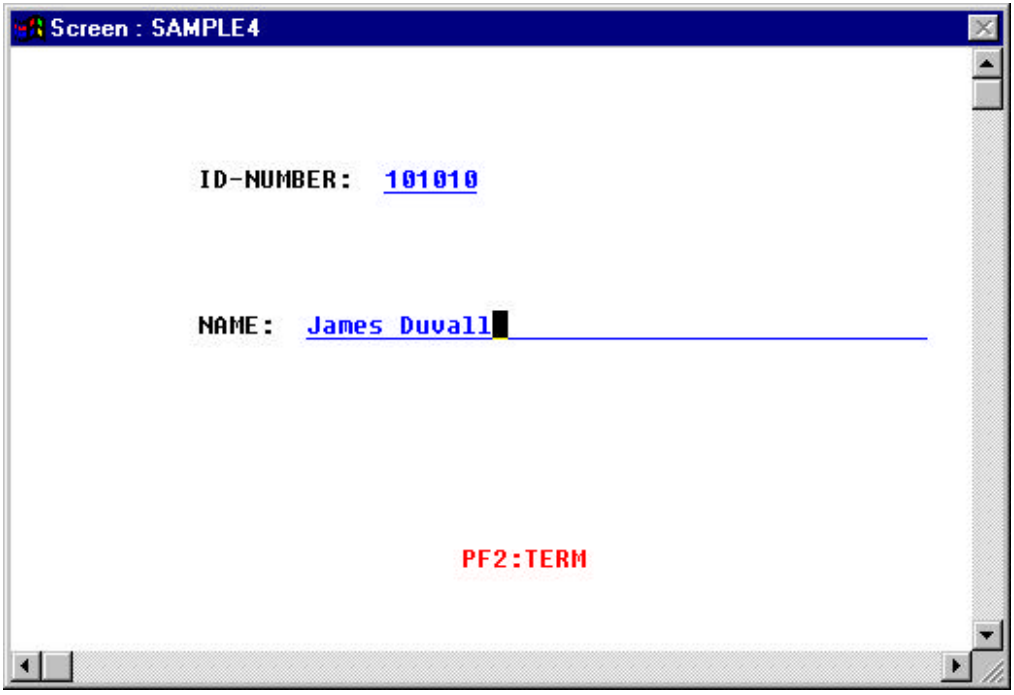


Figure 90. Registering data to the master file

To register data, press the Enter key. The input data is registered in the master file, and the screen is cleared for input of subsequent data.

To quit processing, press the F2 key.

Execution Result

Because the employee number becomes a main record key and the employee name becomes a record sub key, you should use Windows Explorer to confirm that the index file (MASTER) was created in the Sample 4 directory. **Note**: You can edit the MASTER file using the Edit command in the COBOL File Utility (COBFUT32).

Sample 5: Calling Between COBOL Programs

Sample 5 shows a program that calls a subprogram from the main program. Sample 5 was created using the free format.

Function

Converts the contents of the master file (indexed file created in Sample 2) to characters that can be printed, stores the characters in the work file (*.TMP), then prints them.

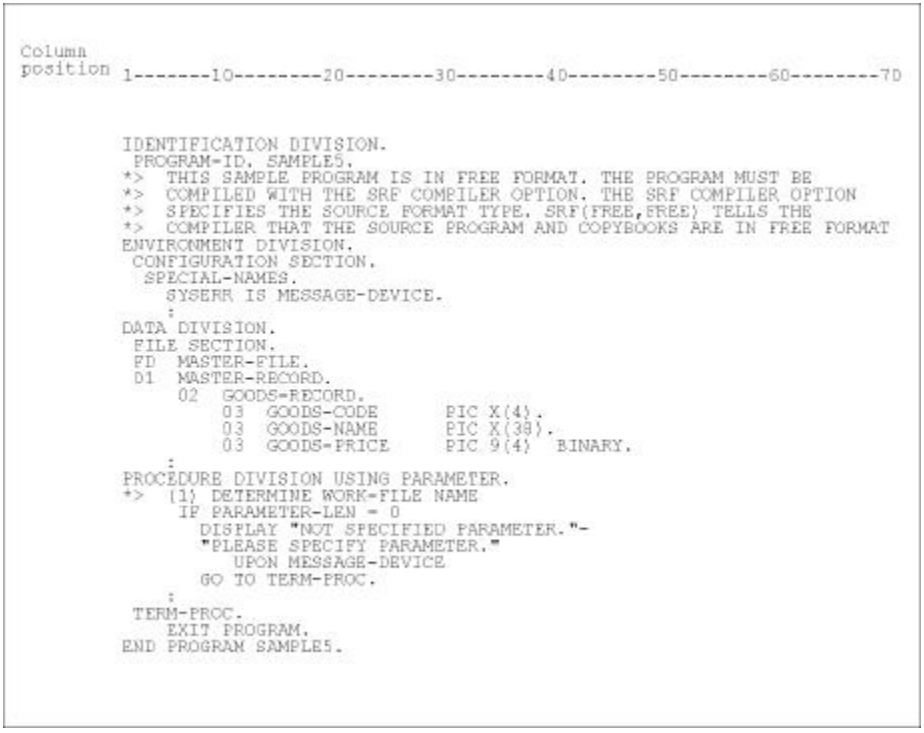
The master file stores product codes, product names, and unit prices. The work file name must be specified in the parameter at program execution.

COBOL Statements Used

CALL statement, DISPLAY statement, EXIT statement, GO TO statement, MOVE statement, OPEN statement, READ statement, and WRITE statements are used.

COBOL Source Program Using the Free Format

The following is an example of the free format used in a COBOL source program.



```
Column
position 1-------10--------20--------30--------40--------50--------60--------7D


         IDENTIFICATION DIVISION.
          PROGRAM-ID. SAMPLE5.
         *>  THIS SAMPLE PROGRAM IS IN FREE FORMAT. THE PROGRAM MUST BE
         *>  COMPILED WITH THE SRF COMPILER OPTION. THE SRF COMPILER OPTION
         *>  SPECIFIES THE SOURCE FORMAT TYPE. SRF(FREE,FREE) TELLS THE
         *>  COMPILER THAT THE SOURCE PROGRAM AND COPYBOOKS ARE IN FREE FORMAT
         ENVIRONMENT DIVISION.
          CONFIGURATION SECTION.
           SPECIAL-NAMES.
             SYSERR IS MESSAGE-DEVICE.
             :
         DATA DIVISION.
          FILE SECTION.
          FD  MASTER-FILE.
          01  MASTER-RECORD.
              02  GOODS-RECORD.
                  03  GOODS-CODE      PIC X(4).
                  03  GOODS-NAME      PIC X(30).
                  03  GOODS-PRICE     PIC 9(4)  BINARY.
             :
         PROCEDURE DIVISION USING PARAMETER.
         *>  (1) DETERMINE WORK-FILE NAME
              IF PARAMETER-LEN = 0
                DISPLAY "NOT SPECIFIED PARAMETER."-
                "PLEASE SPECIFY PARAMETER."
                   UPON MESSAGE-DEVICE
                GO TO TERM-PROC.
             :
          TERM-PROC.
             EXIT PROGRAM.
         END PROGRAM SAMPLE5.
```

Figure 91. A COBOL source program using the free format

Note: In the above figure, ellipses are used to denote sections of source code that have been omitted in this example.

In the free format, COBOL statements can be written in any arbitrary character position on the line. Lines beginning with "*>" will be treated as comments. Additionally, you can divide character constants, Japanese constants, etc. into two or more lines by using the "*>".

Note: You must specify the SRF compiler option in order to use the free format. The SRF compiler option has two parameters; the first specifies the format for the source program and the second specifies the format of the copy books. All copy books must have the same format type. The available types are FIX, for fixed format source, VAR, for variable format source, and FREE, for free format source.

File Interdependence

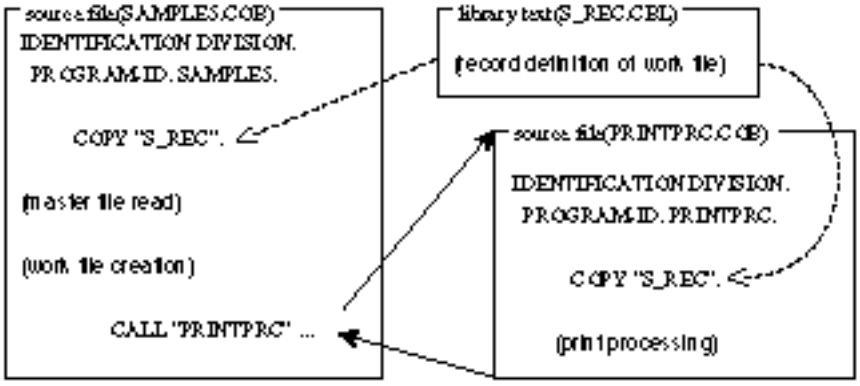


Figure 92. Interdependence between SAMPLE5.COB and PRINTPRC.COB

Prerequisite to Program Execution

The master file created in Sample 2 is used. Therefore, execute the program in Sample 2 before executing Sample 5.

Project File Contents

Sample 5 supports creation of a new project file (SAMPLE5.PRJ). The project file is based on the assumption that C:\FSC is specified as the COBOL85 installation directory.

1. Create the project file (SAMPLE5.PRJ).

   Select Open from the Project menu of the P-STAFF window, then type the project file name (SAMPLE5.PRJ) in the Open Project window.

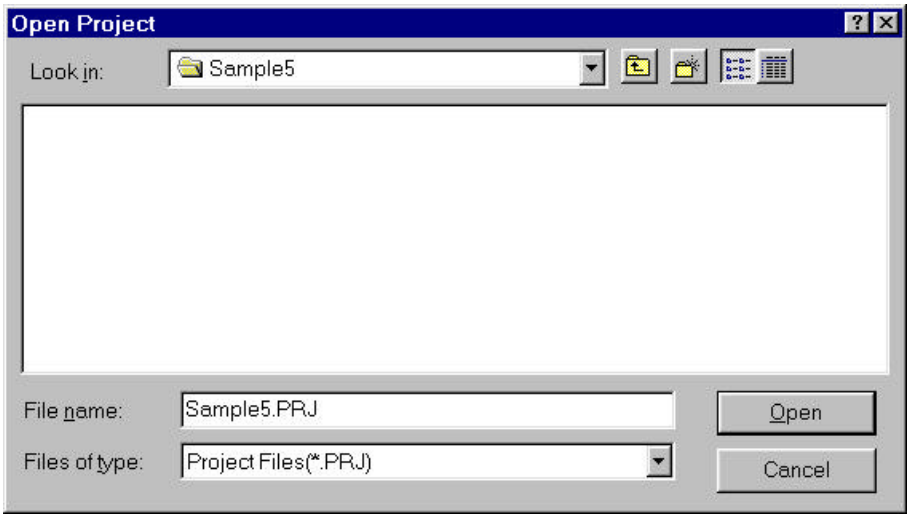


Figure 93. The Open Project window

Click on the Open button. The following window is displayed.

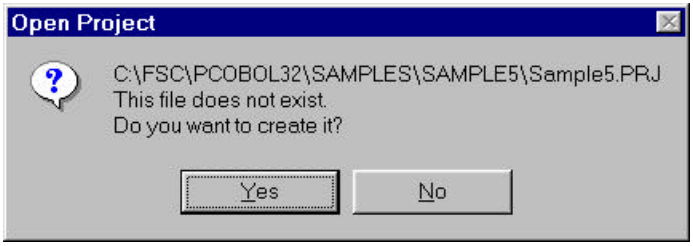


Figure 94. The Open Project creation window

Click on the Yes button to create the SAMPLE5.PRJ file.

The Sample5.prj window (inactive) and the Target Files dialog box (active) are displayed.

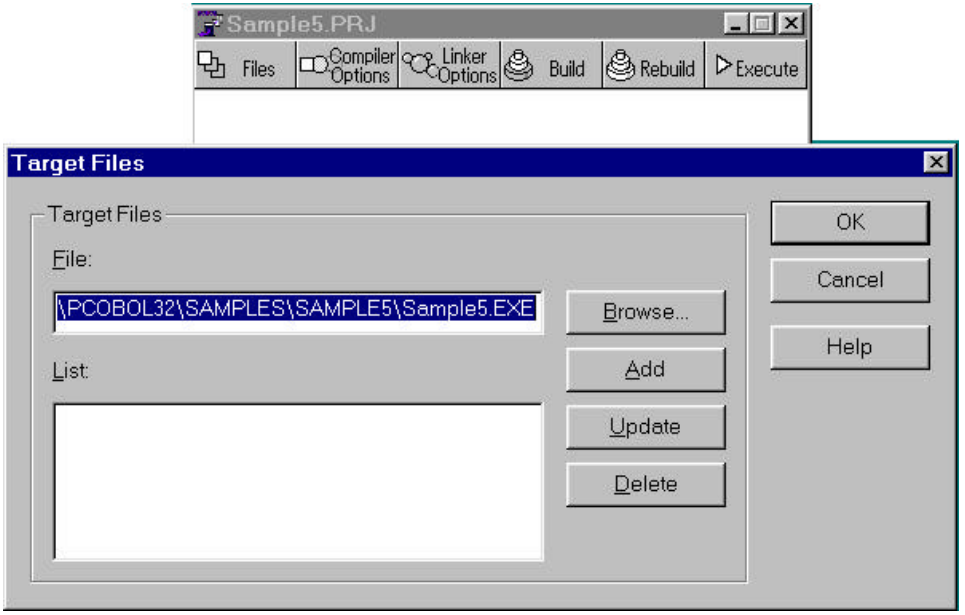


Figure 95. The Target Files dialog box and the Sample 5 project window

2. Register a target file.

   By default, the executable program name appears in the File edit box. Click on the Add button to specify the executable program name (SAMPLE5.EXE) of the main program to be created.

You can type over the executable program name to specify the DLL name (PRINTPRC.DLL) of the subprogram to be created. Then click on the Add button to add the DLL name to the list.

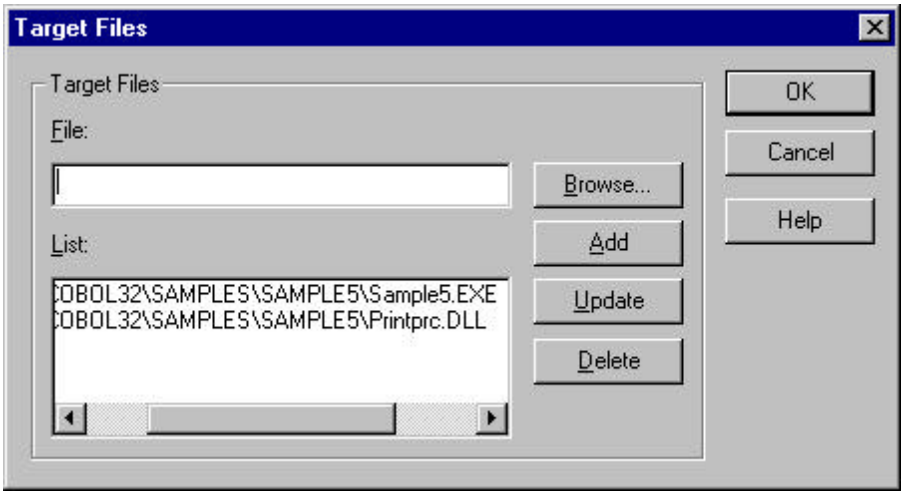


Figure 96. Registering target files in the Target Files dialog box

3. Register dependent files.

   Click on the OK button of the Target Files dialog box. The Dependent Files dialog box opens.

By default, the SAMPLE5.EXE file appears in the Targets edit box in the Dependent Files window. Click on the Browse button to locate the source file name (SAMPLE5.COB) of the main program. Highlight SAMPLE5.COB in the Browse Files window and click on the Open button. Then click on the Add button to add SAMPLE5.COB to the list. Highlight SAMPLE5.COB in the list and click on the Main Program button. The SAMPLE5.COB icon is displayed in red.

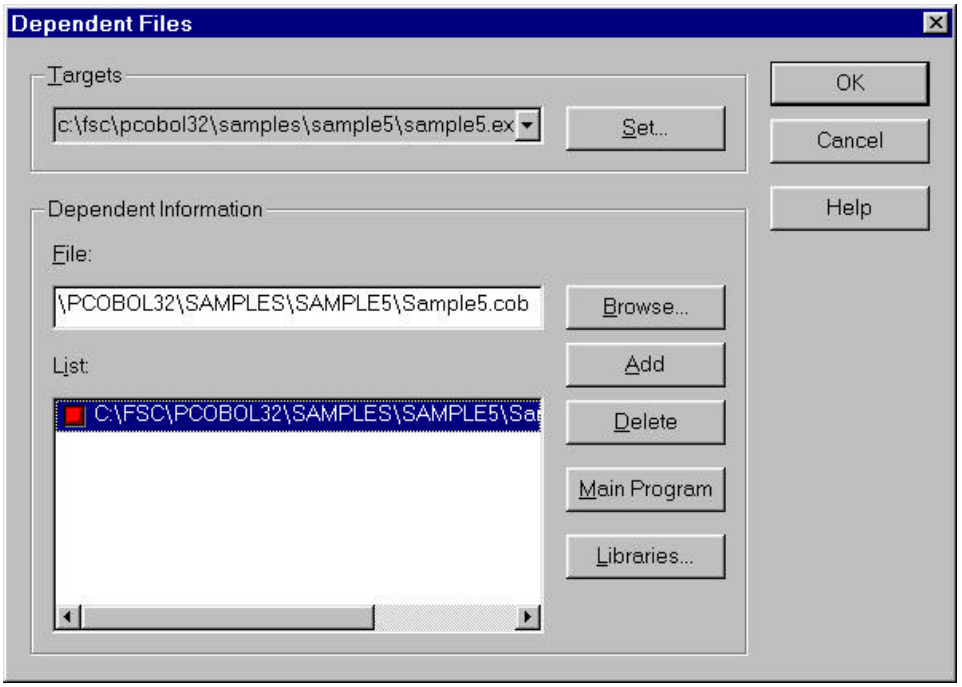


Figure 97. Specifying the main program in the Dependent Files dialog box

4. Register a library file.

   Highlight SAMPLE5.COB in the list and click on the Libraries button. The Library Files dialog box is displayed.

Click on the Browse button to locate the library file (S_REC.CBL) and then click on the Add button to add it to the list.

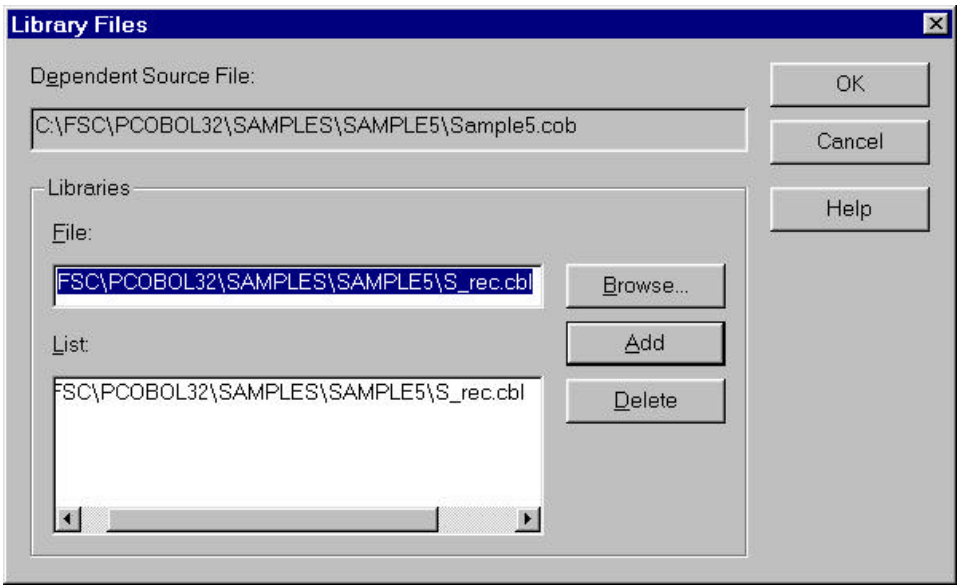


Figure 98. Registering library files in the Library Files dialog box

Click on the OK button to return to the Dependent Files dialog box.

5. Specify the import library name (PRINTPRC.LIB). This file will be created by P-STAFF when the PRINTPRC.DLL file is created.

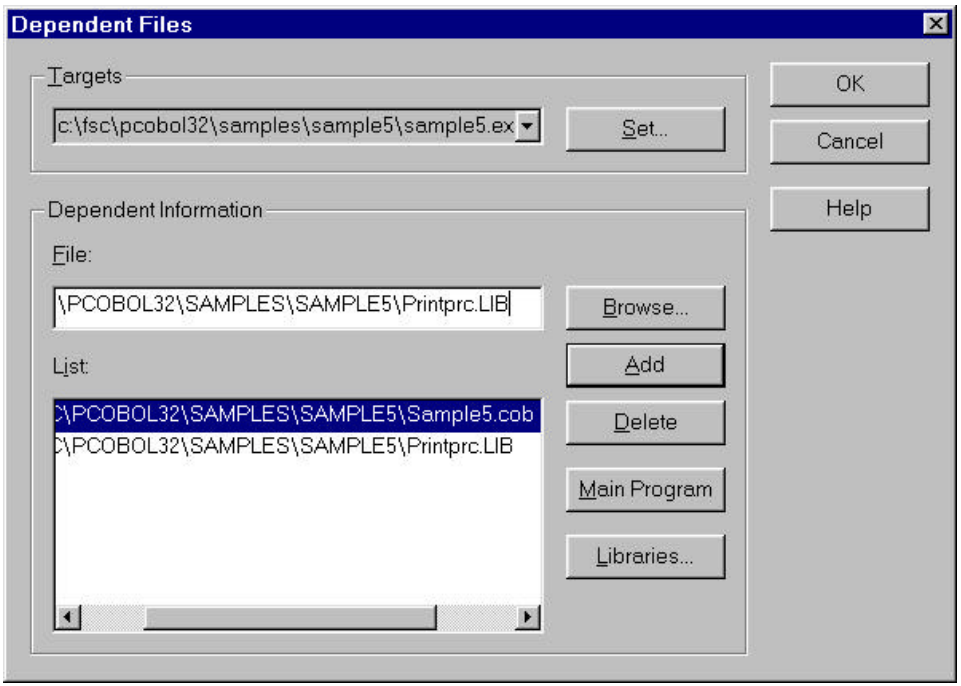


Figure 99. Specify the import library name in the Dependent Files dialog box

6. Change the target to PRINTPRC.DLL.

   Specify the source file name (PRINTPRC.COB) of the subprogram in the same manner as described above. **Note**: DLL's do not have a main program therefore PRINTPRC.COB cannot be specified as a main program.

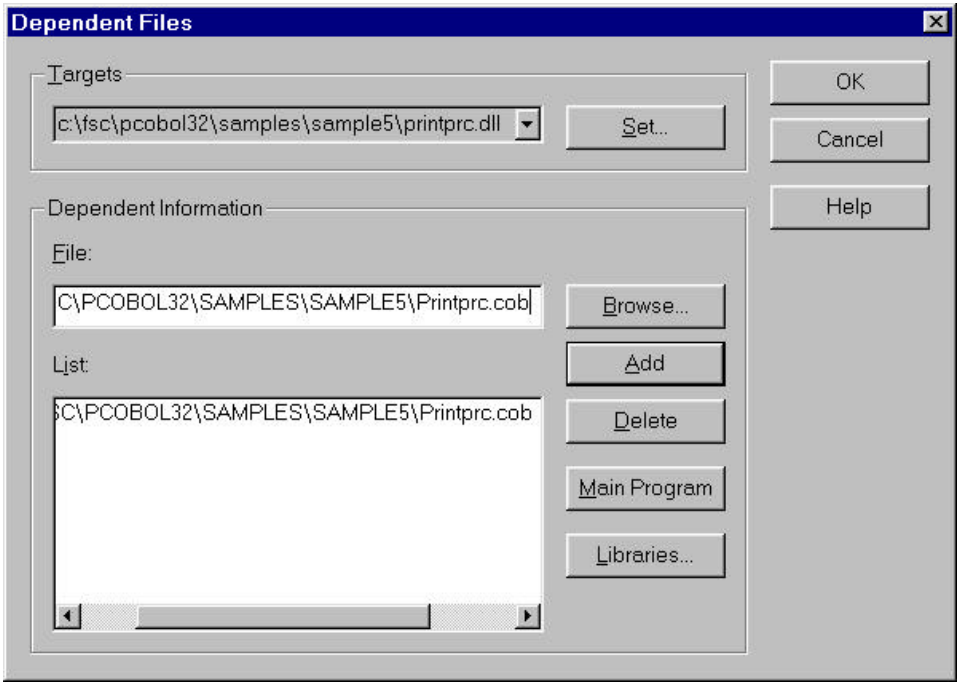


Figure 100. Specifying dependent files in the Dependent Files dialog box

7. Register a library file.

   Highlight PRINTPRC.COB in the list and click on the Libraries button. The Library Files dialog box is displayed.

Click on the Browse button to locate the library file (S_REC.CBL) and then click on the Add button to add it to the list.

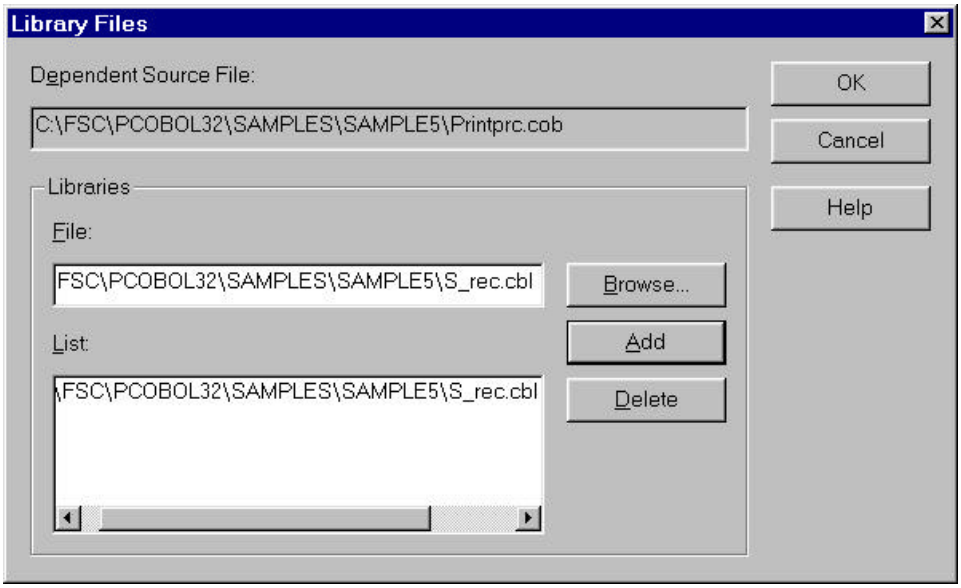


Figure 101. Registering library files in the Library Files dialog box

Click on the OK button to return to the Dependent Files dialog box.

Click on the OK button at the completion of file registration.

The programs (SAMPLE5.COB, PRINTPRC.COB) fetch the library text (S_REC.CBL) with the COPY statement. Registering library files in this manner allows P-STAFF to automatically recompile programs when library files have been modified.

The registered file names are displayed in the project window.

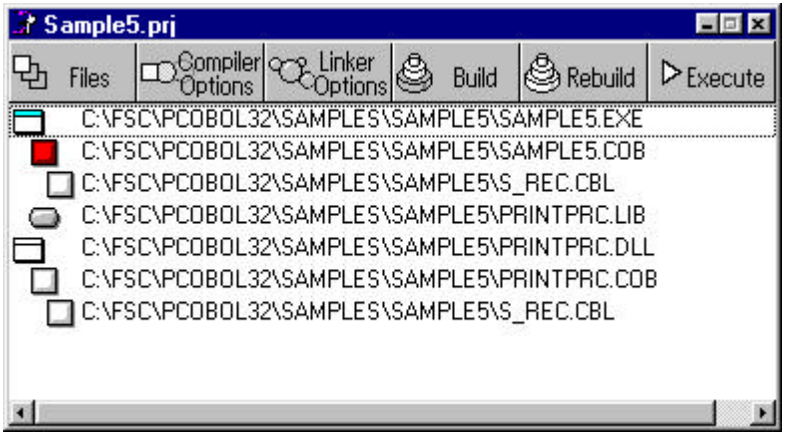


Figure 102. The project window

8. Specify compiler options.

   Click on the Compiler Options button. The Compiler Options dialog box opens.

Click on the Add button and highlight the LIB option, then click on the Add button. Alternatively, you can double click on the LIB option to select it. Click on the Browse button to locate the directory where the library texts are stored. You can click on any file in the directory to specify the directory of the library file (S_REC.CBL). Click on the OK button to close the Browse Files window and then click the OK button in the Compile Options sub-window.

Click on the Add button again and highlight the SRF option. Click on the Add button and then specify the free format (FREE, FREE) in the Compile Options sub-window. Click on the OK button to return to the main Compile Options window.

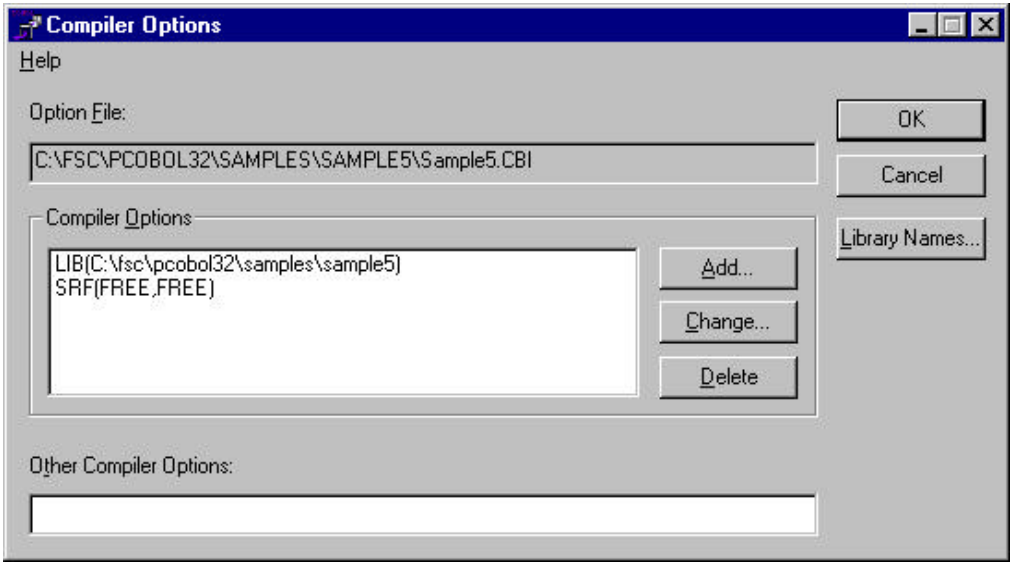


Figure 103. The Compiler Options dialog box with LIB and SRF compiler options specified

Note: If all library files are registered in the Dependent Files dialog box it is not necessary to specify the LIB compiler option.

9. Specify a linker option.

   If you need to specify linker options, click on the Linker Options button in the project window. The Project-Linker Options dialog box opens. For this sample, data specification is not required. Close this dialog box.

Specification of information required for the project is now completed.

Building a Project

Build (or rebuild) a project to create an executable program. To build (or rebuild) a program, click on the Build (or Rebuild) button of the project window.

The following files are created at completion of program build or rebuild:

- PRINTPRC.OBJ
- SAMPLE5.OBJ
- PRINTPRC.LIB
- PRINTPRC.DLL
- SAMPLE5.EXE

Program Execution

To execute a program, click on the Execute button in the project window. The Run-time Environment Setup window opens when the program execution starts.

Specify the following setup items:

1. Specify a work file name for @MGPRM (executing parameter). The work file name can contain up to 8 alphanumeric characters. A file with the extension "TMP" (added to the specified work file name) is created.
2. For the file-identifier INFILE, specify the path name of the master file (MASTER) created in Sample 2. You need to have run Sample 2 correctly in order for this sample to run.

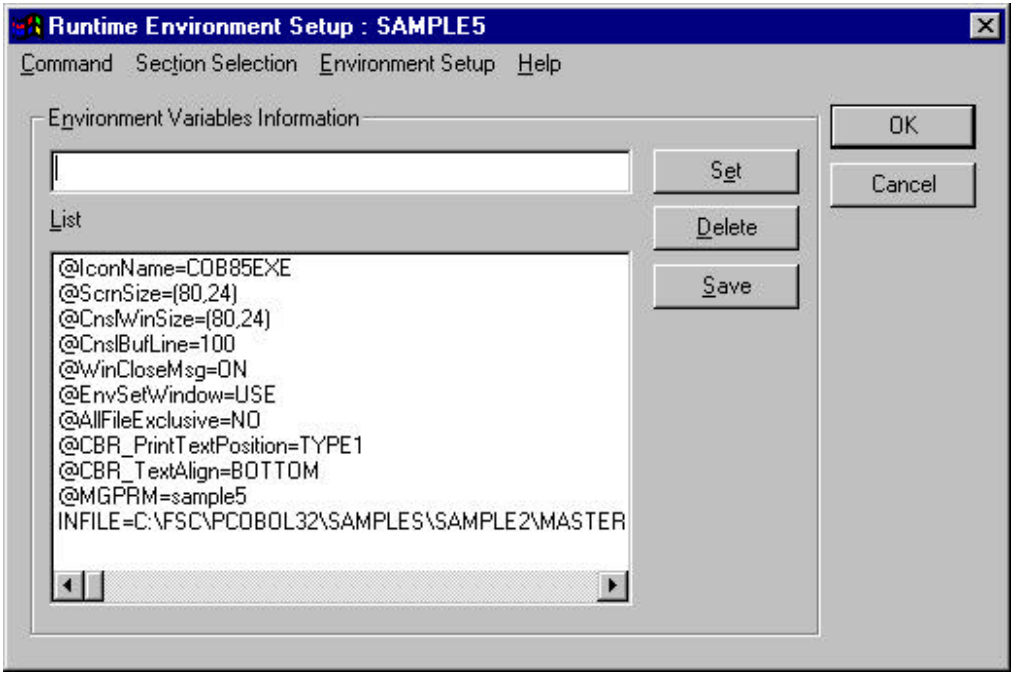


Figure 104. The Run-time Environment Setup window

Note: Click on the Save button to save the additions to the initialization file.

Execution Procedure

Click on the OK button in the Run-time Environment Setup window to start program execution. The message "GENERATE WORK-FILE=sample5.TMP" is displayed.

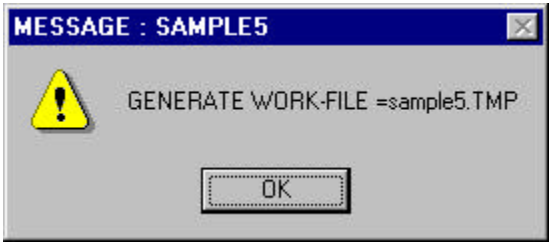


Figure 105. A message box

Confirm the contents, then click on the OK button to close the message box.

Execution Result

The master file contents are written to the system default printer at the completion of program execution. The system default printer is as follows:

Windows 95

“Set as Default” is checked for the printer.

Windows NT

Printer is set as default using the Print Manager.

For Windows 3.1:

Printer is set as default using the Print Manager.

Sample 6: Receiving a Command Line Argument

Sample 6 shows a program that receives an argument specified at execution of a COBOL program, using the command line argument handling function. Refer to the "COBOL85 User's Guide" for details on how to use the command line argument handling function.

Sample 6 also calls an internal program.

Function

Obtains the number of days from the start date to end date. The start and end dates are specified as command arguments in the following format:

```
command-name  start-date  end-date
```

START-DATE END-DATE

Specify a year, month, and day between January 1, 1900 and December 31, 1999 in the YYMMDD format. For YY, specify the last two digits of the year.

COBOL Statements Used

ACCEPT statement, CALL statement, COMPUTE statement, COPY statement, DISPLAY statement, DIVIDE statement, EXIT statement, GO TO statement, IF statement, MOVE statement, and PERFORM statement are used.

Program Compilation

The following figures show the setup items in the WINCOB window and the Compiler Options dialog box if C:\FSC is specified as the COBOL85 installation directory.

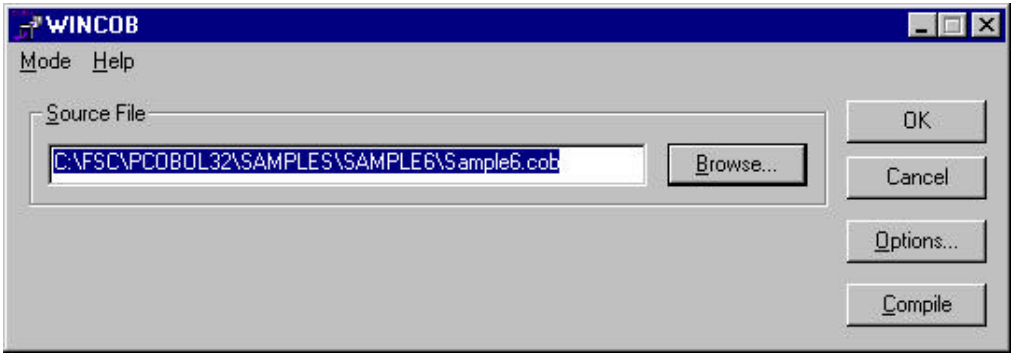


Figure 106. The WINCOB window

Specify the main program to the compiler option MAIN. Refer to Chapter 3, “A Quick Tour” for details on specifying compiler options.

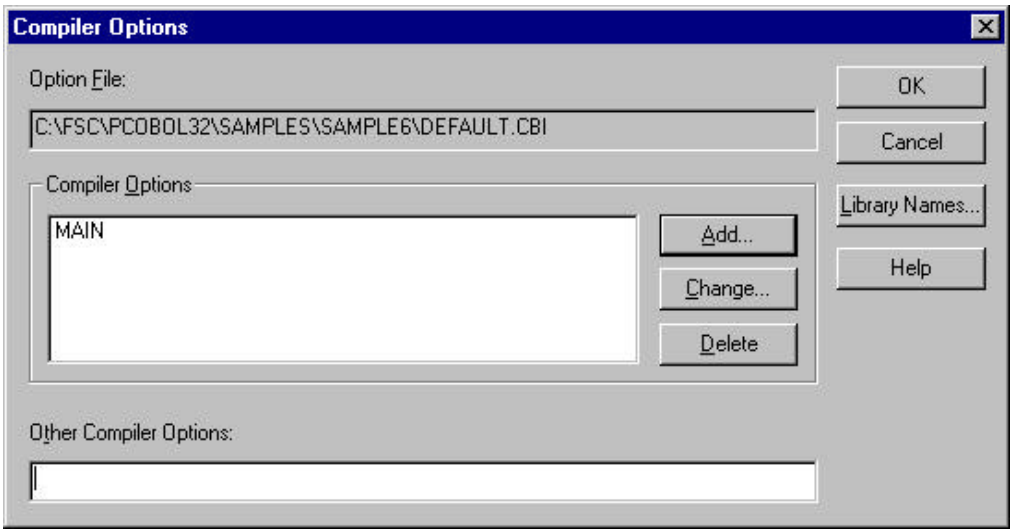


Figure 107. The Compiler Options dialog box with compiler option MAIN specified

Program Linkage

The following figure shows the setup items of the WINLINK [Linking Files] window for linking the object program (SAMPLE6.OBJ) created through compilation.

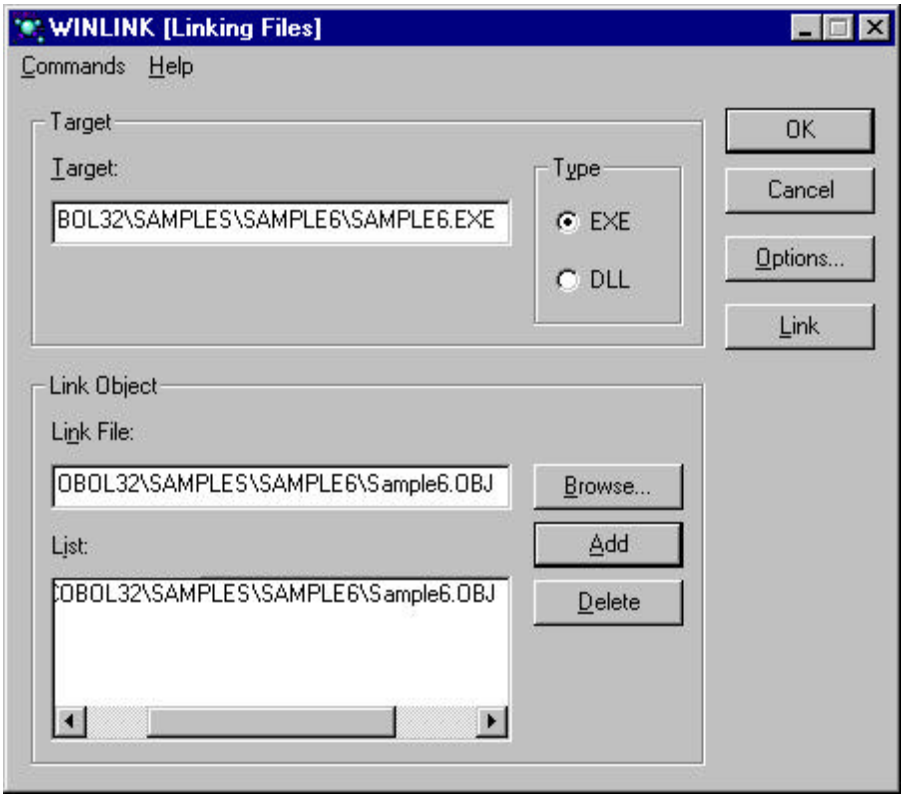


Figure 108. The WINLINK [Linking Files] window

A target name is displayed when an object file is added. Refer to Chapter 3, “A Quick Tour” for details on using WINLINK.

Program Execution

The following figure shows the setup items of the WINEXEC window for executing the executable program (SAMPLE6.EXE) created through linkage.

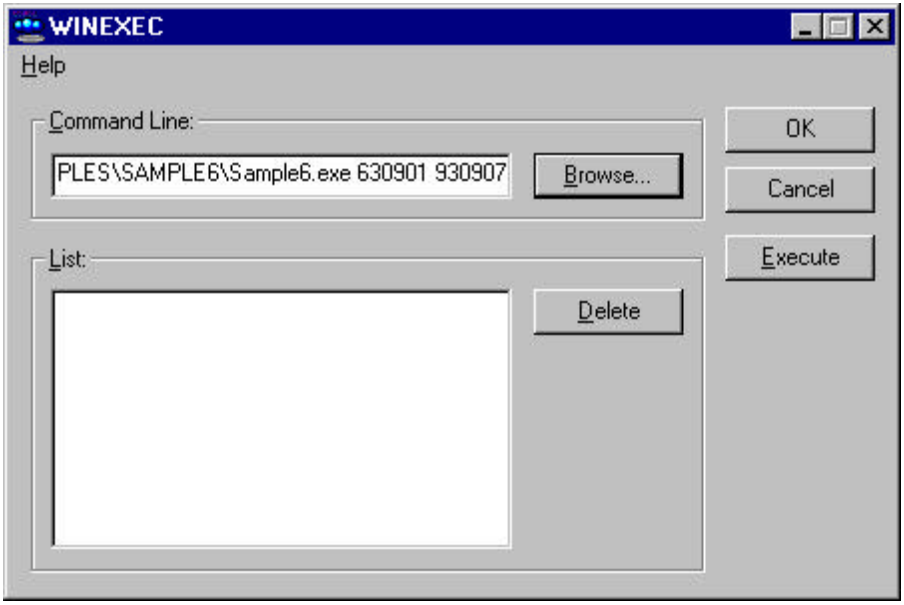


Figure 109. The WINEXEC window

Enter the start date and end date following the file name.

Sample 6 does not need specification of run-time environment information. Click on the OK button when the Run-time Environment Setup window is displayed.

Refer to Chapter 3, “A Quick Tour” for details on using WINEXEC.

Execution Result

The number of days from April 6, 1971 to April 6, 1996 is displayed.

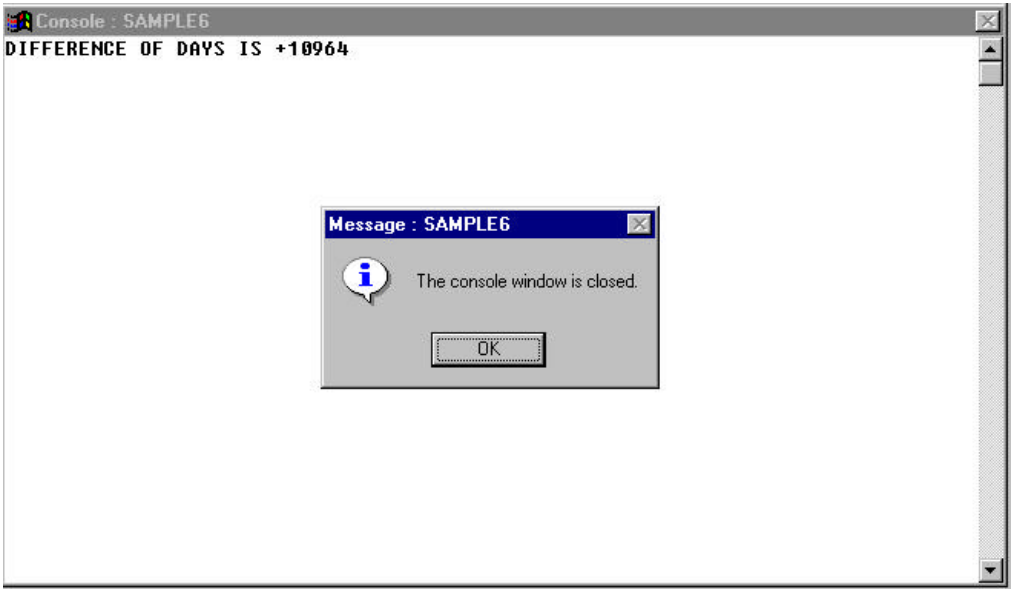


Figure110. Execution results displayed in the console window

Sample 7: Environment Variable Handling

Sample 7 shows a program that changes the value of an environment variable during COBOL program execution, using the environment variable handling function. Refer to the "COBOL85 User's Guide" for details on how to use the environment variables handling function.

Function

Divides the master file (indexed file created in Sample 2), storing product codes, product names, and unit prices, into two master files according to product codes. Table 4 shows the division method and the names of the two new master files:

Table 4. Division of the master files

Product Code	File Name
Code beginning with 0	master-file-name.A
Code beginning with a non-zero value	master-file-name.B

COBOL Statements Used

ACCEPT statement, CLOSE statement, DISPLAY statement, EXIT statement, GO TO statement, IF statement, OPEN statement, READ statement, STRING statement, and WRITE statements are used.

Prerequisite to Program Execution

The master file created in Sample 2 is used. Therefore, execute the program in Sample 2 before executing Sample 7.

Program Compilation

The following figures show the setup items of the WINCOB window and the Compiler Options dialog box if C:\FSC is specified as the COBOL85 installation directory.

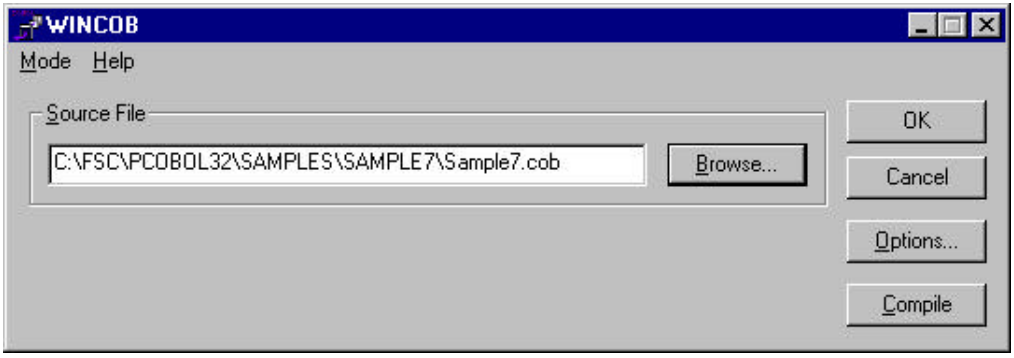


Figure 111. The WINCOB window

Specify the main program to the compiler option MAIN. Refer to Chapter 3, “A Quick Tour” for details on specifying compiler options.

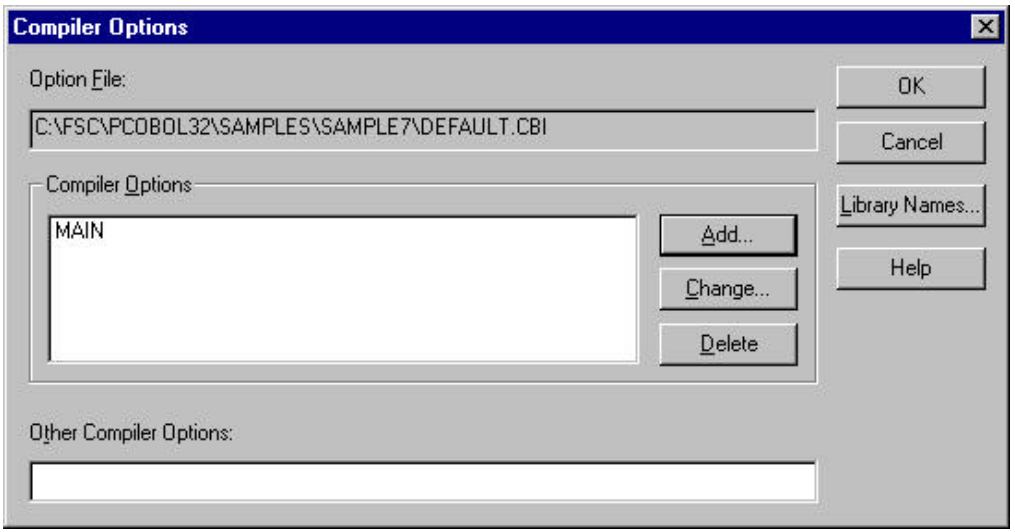


Figure 112. The Compiler Options dialog box with compiler option MAIN specified

Program Linkage

The following figure shows the setup items of the WINLINK [Linking Files] window for linking the object program (SAMPLE7.OBJ) created through compilation.

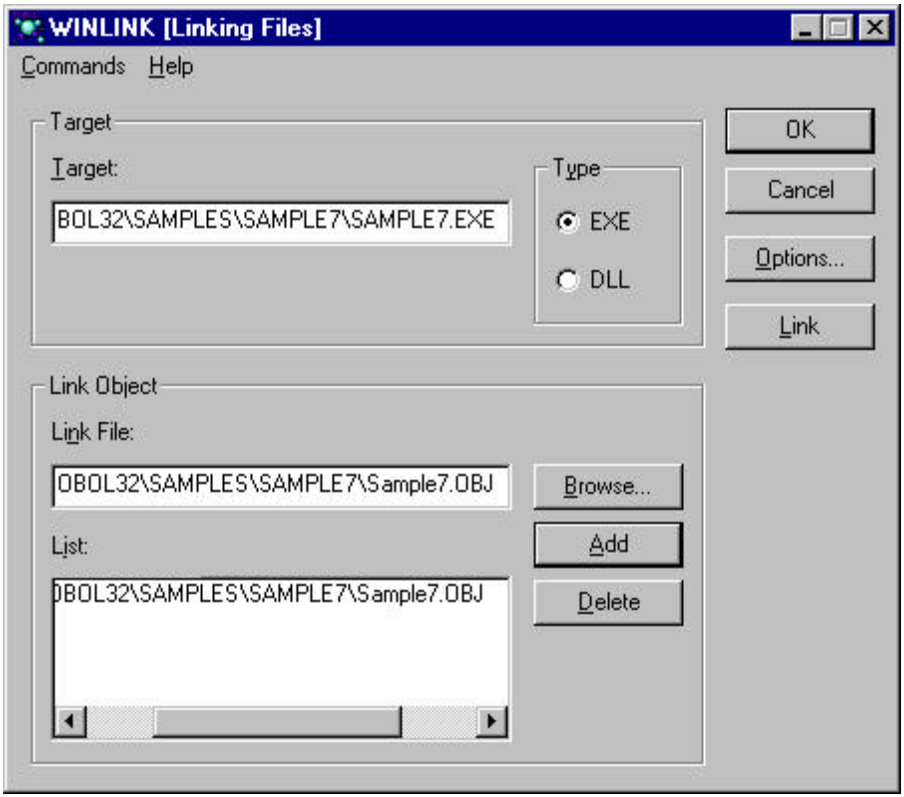


Figure 113. The WINLINK [Linking Files] window

A target name is displayed when an object file is added. Refer to Chapter 3, “A Quick Tour” for details on using WINLINK.

Program Execution

The following figures show the setup items of the WINEXEC window and Run-time Environment Setup window for executing the executable program (SAMPLE7.EXE) created through linkage.

Note: Execute the program in Sample 2 beforehand.

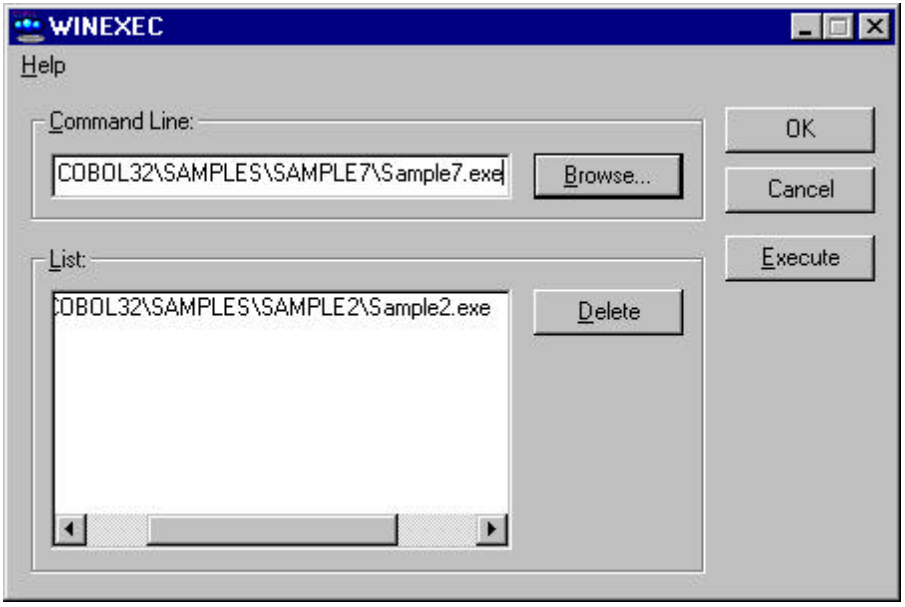


Figure 114. The WINEXEC window

Refer to Chapter 3, “A Quick Tour” for details on using WINEXEC.

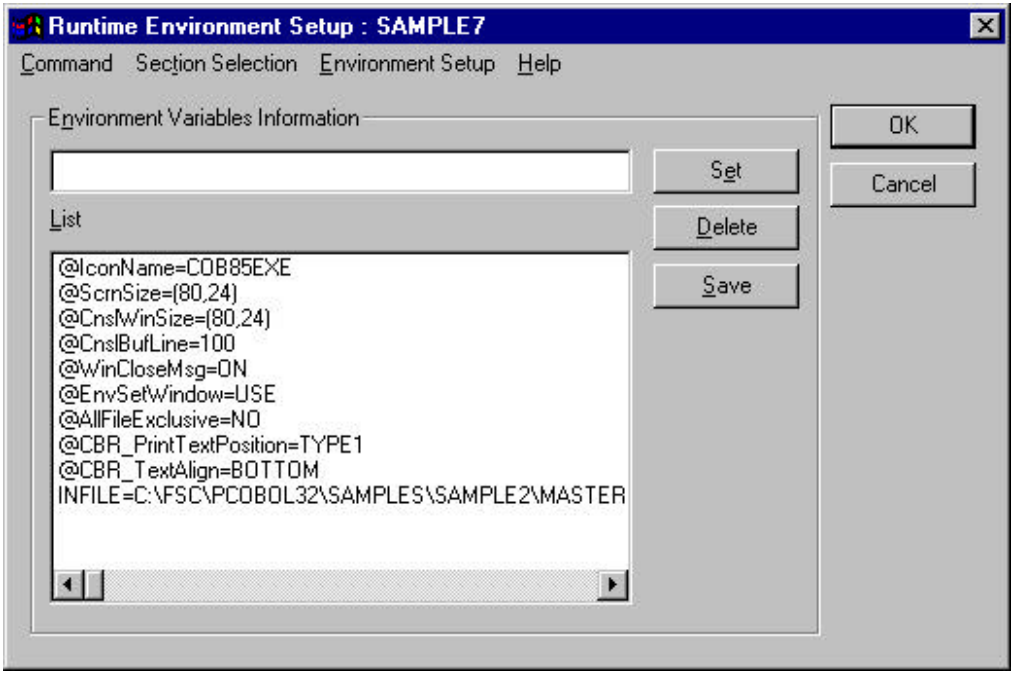


Figure 115. The Run-time Environment Setup window

For the file-identifier INFILE, specify the path name of the master file (MASTER) created in Sample 2.

Execution Result

The following two files are created in the directory of the master file created in Sample 2:

- MASTER.A: Stores the data of products whose codes begin with 0.
- MASTER.B: Stores the data of products whose codes begin with a non-zero value.

The contents of the newly created master files (MASTER.A and MASTER.B) can be checked with the program in Sample 5, in the same manner as for the master file created in Sample 2.

Sample 8: Program Using a Print File

Sample 8 shows a program that outputs data (input from the console window) to a printer using a print file. Refer to the "COBOL85 User's Guide" for details on the print file.

Function

Inputs data of up to 40 alphanumeric characters from the console window, and outputs the data to the printer.

COBOL Statements Used

ACCEPT statement, CLOSE statement, EXIT statement, GO TO statement, IF statement, OPEN statement, and WRITE statements are used.

Program Compilation

The following diagram shows the setup items of the WINCOB window and the Compiler Options dialog box if C:\FSC is specified as the COBOL85 installation directory:

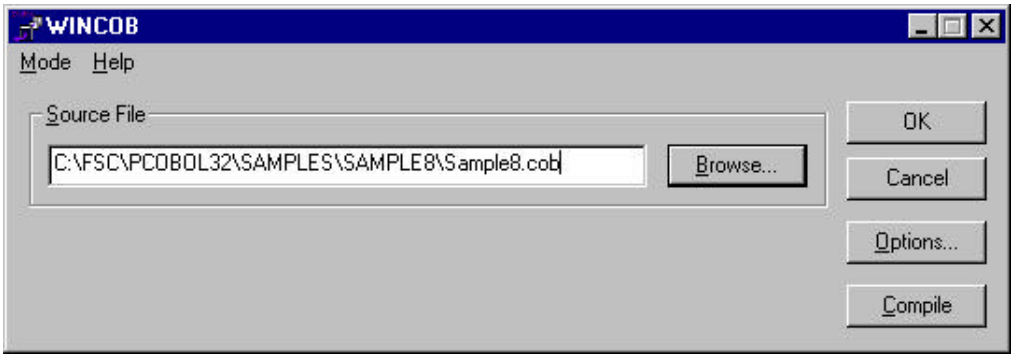


Figure 116. The WINCOB window

Specify the main program to the compiler option MAIN. Refer to Chapter 3, “A Quick Tour” for details on specifying compiler options.

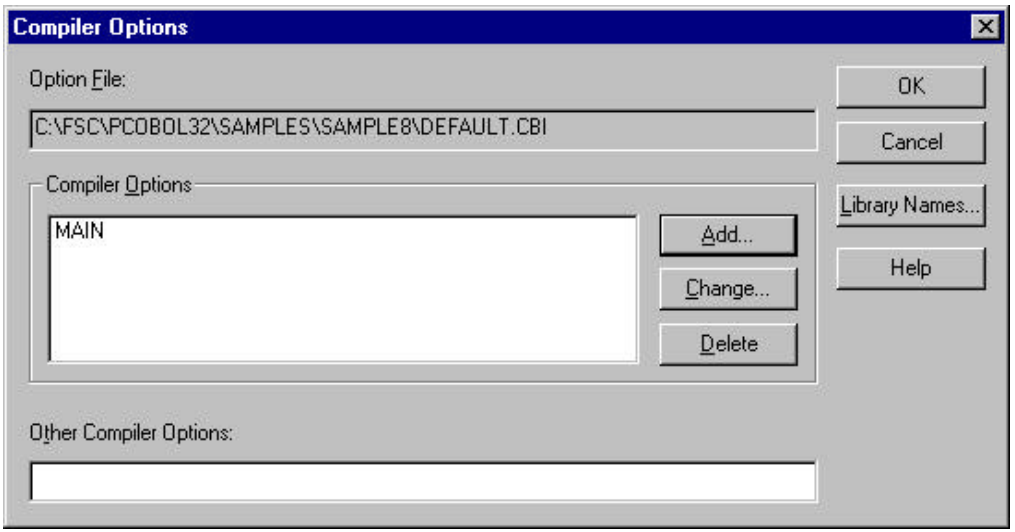


Figure 117. The Compiler Options dialog box with compiler option MAIN specified

Program Linkage

The following figure shows the setup items of the WINLINK [Linking Files] window for linking the object program (SAMPLE8.OBJ) created through compilation.

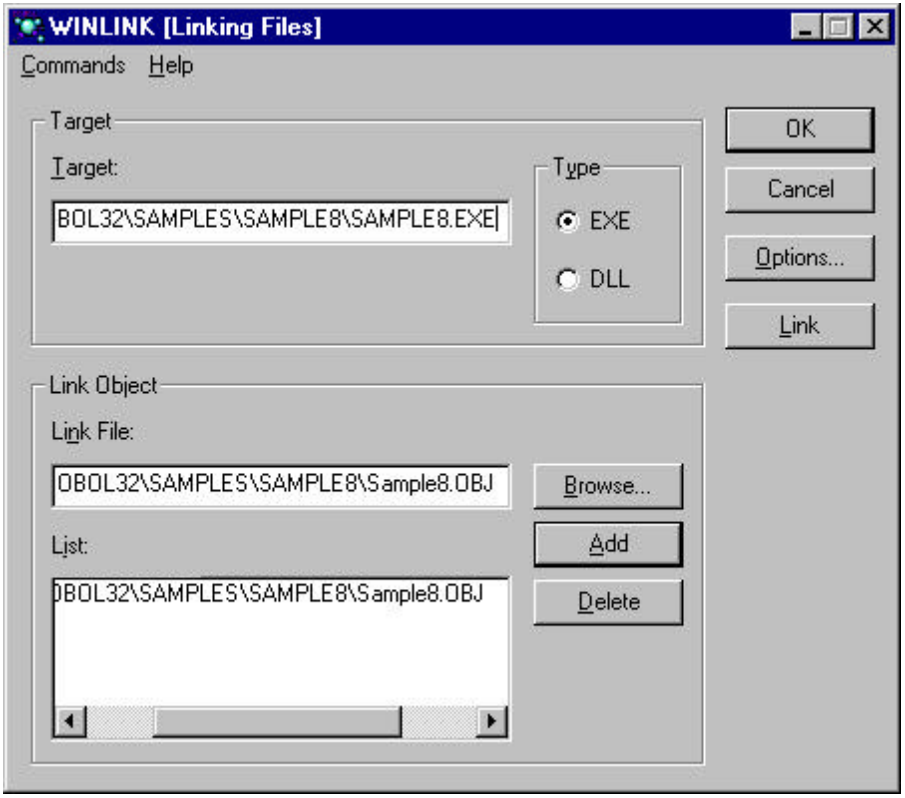


Figure 118. The WINLINK [Linking Files] window

A target name is displayed when an object file is added. Refer to Chapter 3, “A Quick Tour” for details on using WINLINK.

Program Execution

The following figure shows the setup items of the WINEXEC window for executing the executable program (SAMPLE8.EXE) created through linkage.

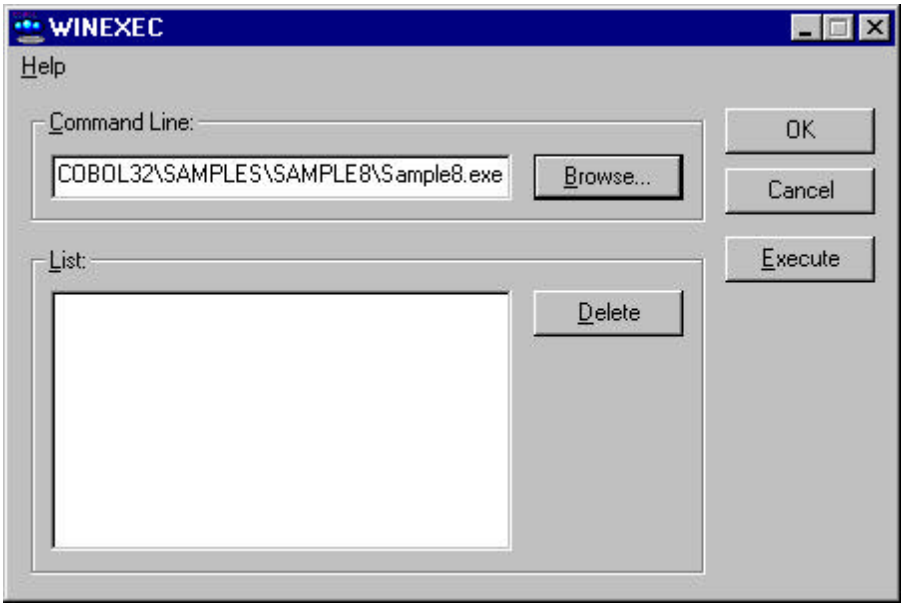


Figure 119. The WINEXEC window

Sample 8 does not require specification of run-time environment information. Refer to Chapter 3, “A Quick Tour” for details on using WINEXEC.

Execution Procedure

Click on the OK button when the Run-time Environment Setup window is displayed. The console window is then displayed.

Enter the data to be printed to the console window. Data of up to 40 characters can be entered at a time.

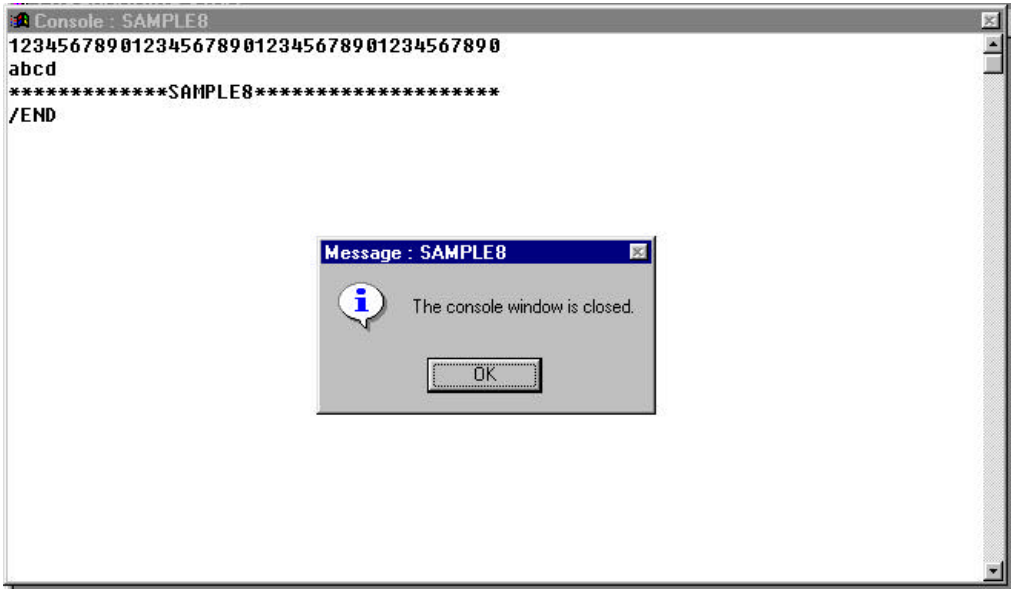


Figure 120. Entering data to be printed to the console window

To terminate the program, press the RETURN key, type "/END" and press the RETURN key. Click on the OK button to close the message window.

Execution Result

The input data is written to the printer at termination of the program.

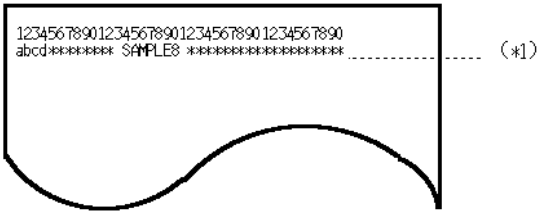


Figure 121. Input data is written to the printer

*1 The second input to the console window is less than 40 characters. The second and third inputs are executed together by the second ACCEPT statement in the program.

Sample 9: Inter-application Communication Function

Sample 9 shows how to use the simplified inter-application communication function to write or read messages to or from logical destinations. For details on how to use the simplified inter-application communication function for message transfer, refer to the “COBOL85 User’s Guide.”

Function

Messages are transferred between SAMPLE9 and COMMU.

SAMPLE9 establishes a connection to server "SERVER1", writes messages to logical destination "MYLD1", and reads messages from logical destination "MYLD2". If no message comes from logical destination "MYLD2", SAMPLE9 waits for the message arrival for up to 60 seconds. After reading messages from logical destination "MYLD2", SAMPLE9 reads messages from logical destination "MYLD1" in order of priority.

COMMU establishes a connection to server "SERVER1", reads messages from logical destination "MYLD1", then writes messages to logical destinations "MYLD1" and “MYLD2”.

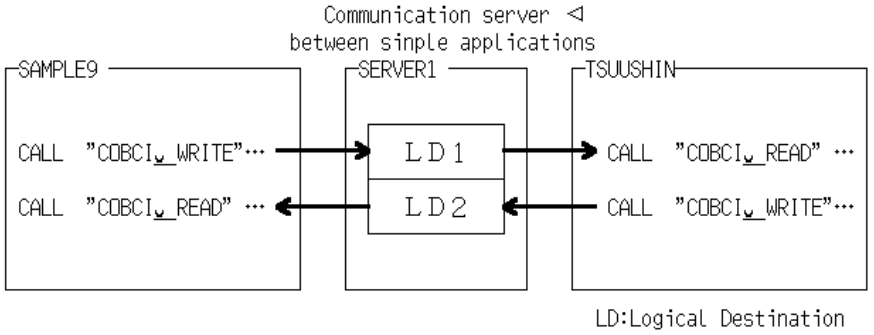


Figure 122. Reading and writing messages between logical destinations

COBOL Statements Used

CALL, DISPLAY, IF, and MOVE statements are used.

Before Executing the Program

Refer to Chapter 14. “Communication Functions” in the “COBOL85 User’s Guide” for more details on the following tasks:

- Activate the server for simplified inter-application communication.
- Create logical destinations "LD1" and "LD2" on the server for simplified inter-application communication.
- Change the information on the opponent machine name of the logical destination definition file by using the logical destination definition file making utility. Specify the host name of the server as the opponent machine name and click on the Add button.

Program Compilation

The following figures show the setup items of the WINCOB window and the Compiler Options dialog box if C:\FSC is specified as the COBOL85 installation directory.

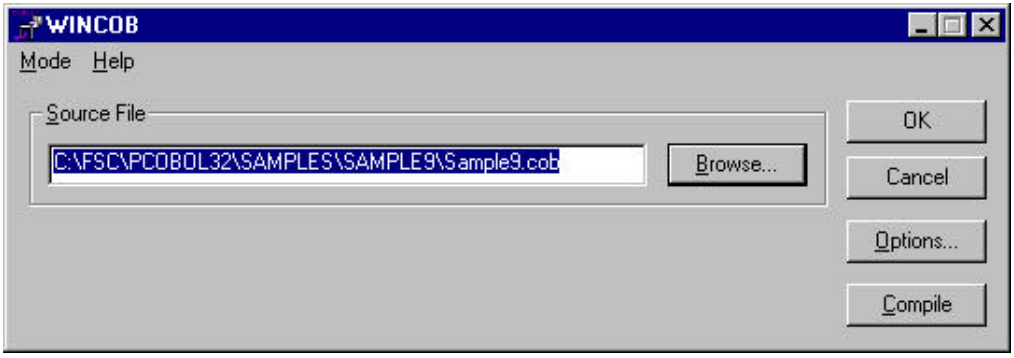


Figure 123. The WINCOB window

Specify the main program to the compiler option MAIN. Refer to Chapter 3, “A Quick Tour” for details on specifying compiler options.

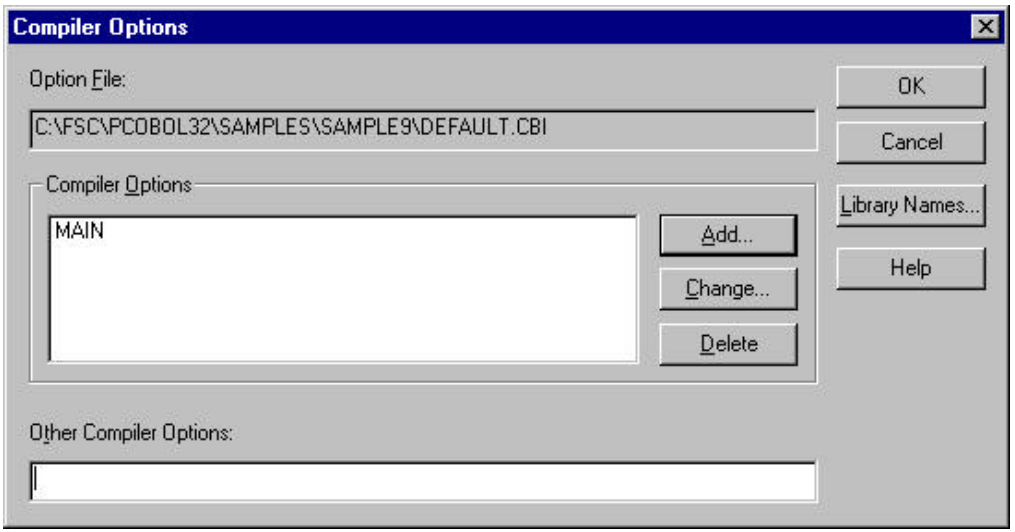


Figure 124. The Compiler Options dialog box with compiler option MAIN specified

Set up COMMU.COB in the same manner.

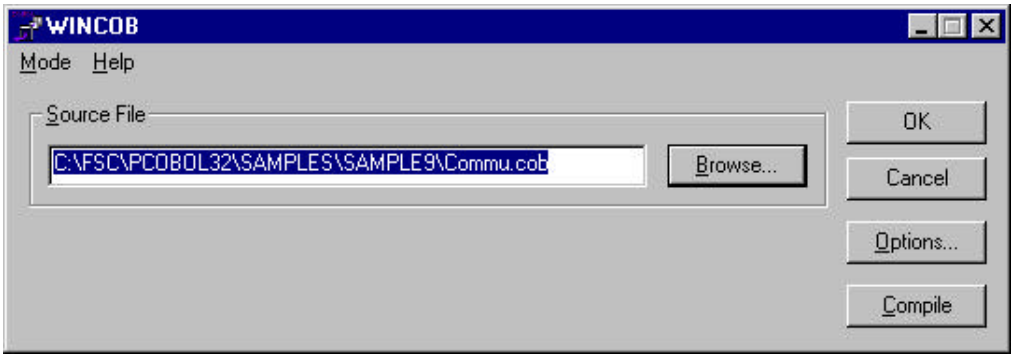


Figure 125. The WINCOB window

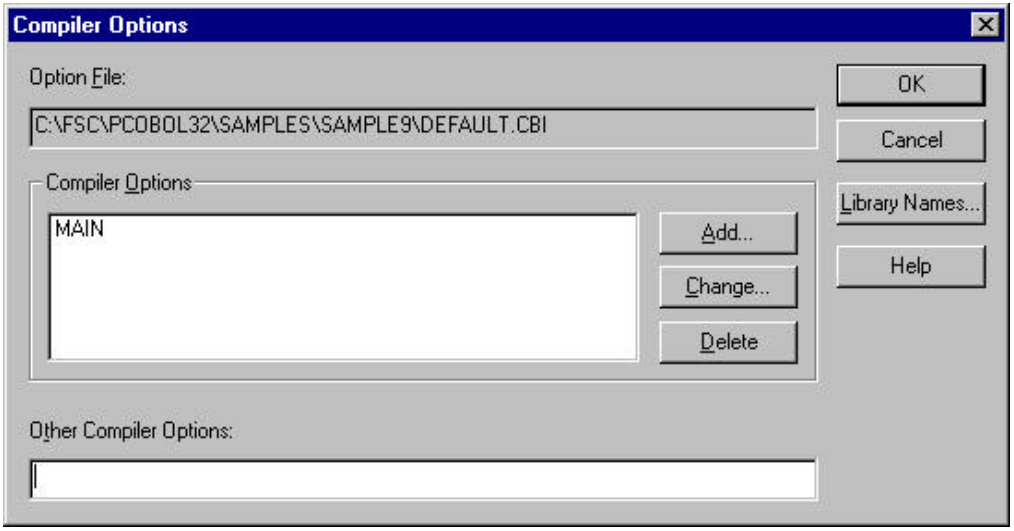


Figure 126. The Compiler Options dialog box with compiler option MAIN specified

Program Linkage

The following figure shows the setup items of the WINLINK [Linking Files] window for linking the object program (SAMPLE9.OBJ) created through compilation.

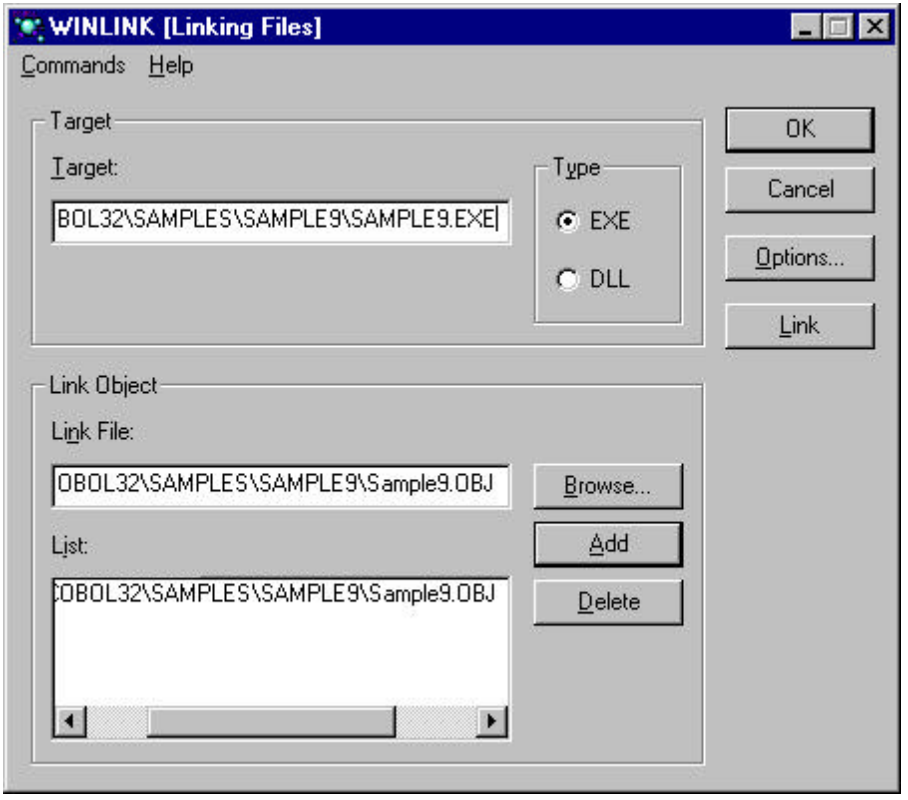


Figure 127. The WINLINK [Linking Files] window

Refer to Chapter 3, “A Quick Tour” for details on using WINLINK.

Set up COMMU.OBJ in the same manner.

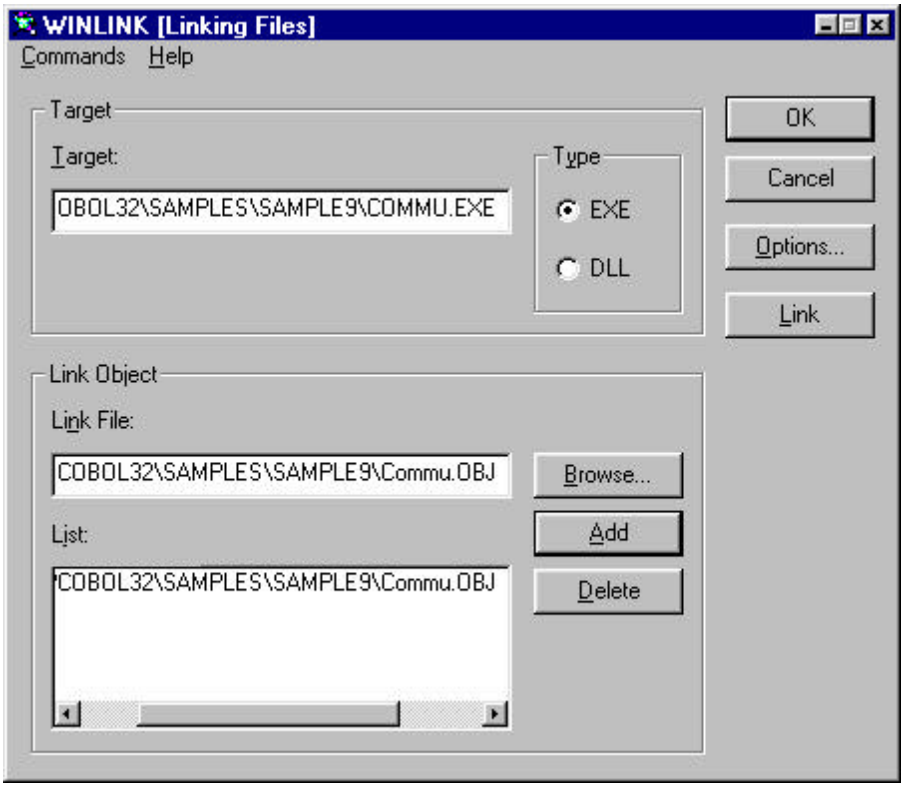


Figure 128. The WINLINK [Linking Files] window

Program Execution

The following figures show the setup items of the WINEXEC window and Run-time Environment Setup window for executing the executable program (SAMPLE9.EXE) created through linkage.

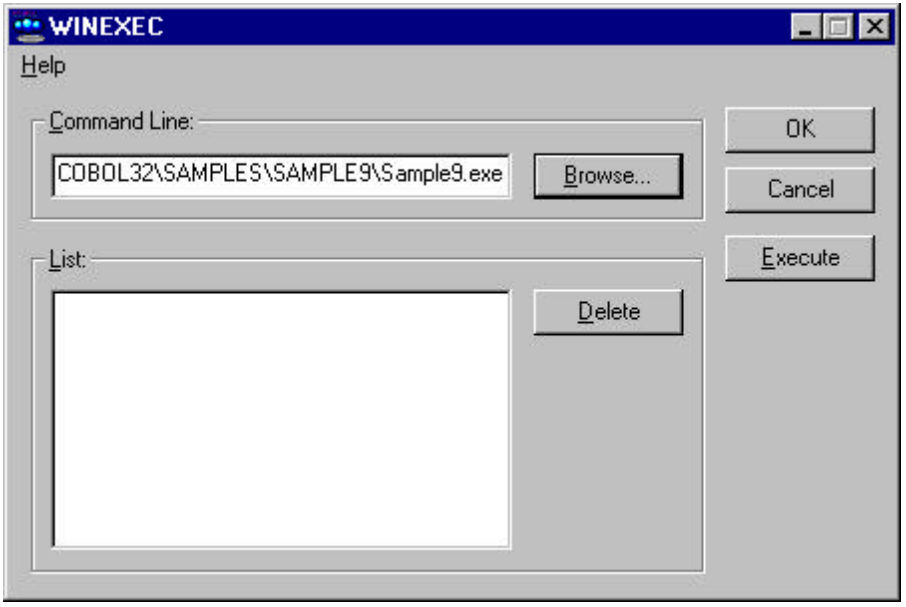


Figure 129. The WINEXEC window

Refer to Chapter 3, “A Quick Tour” for details on using WINEXEC.

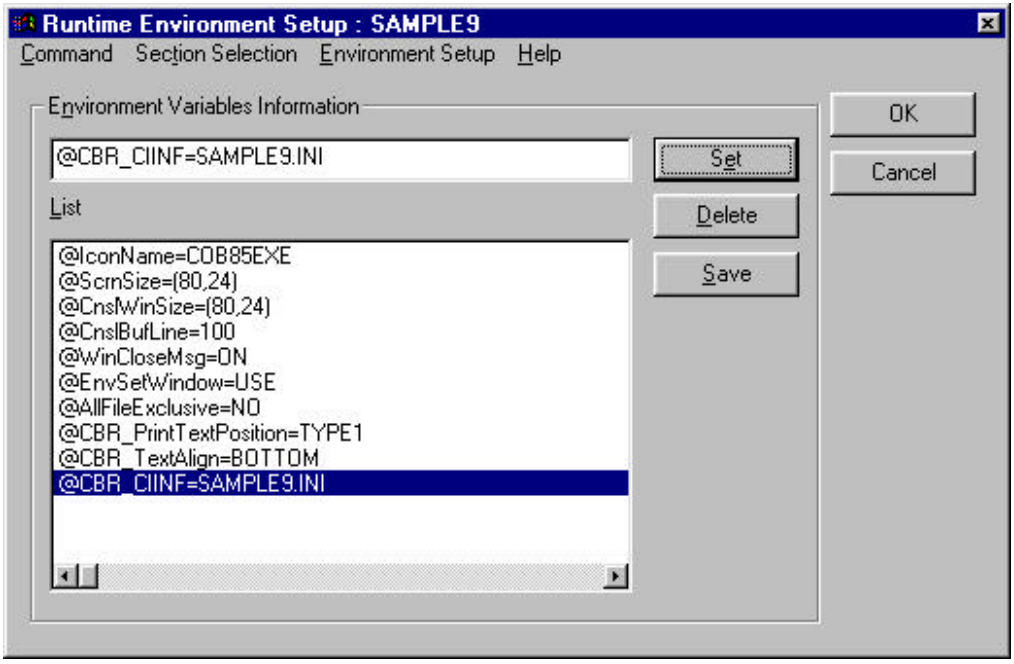


Figure 130. The Run-time Environment Setup window

Assign the path name of the logical destination definition file to "@CBR_CIINF".

Set up COMMU.EXE in the same manner.

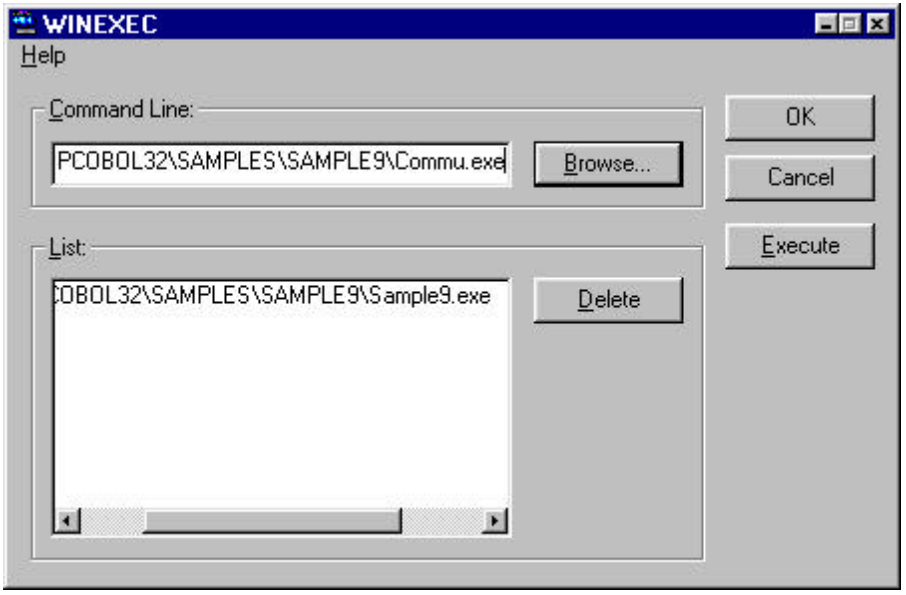


Figure 131. The WINEXEC window

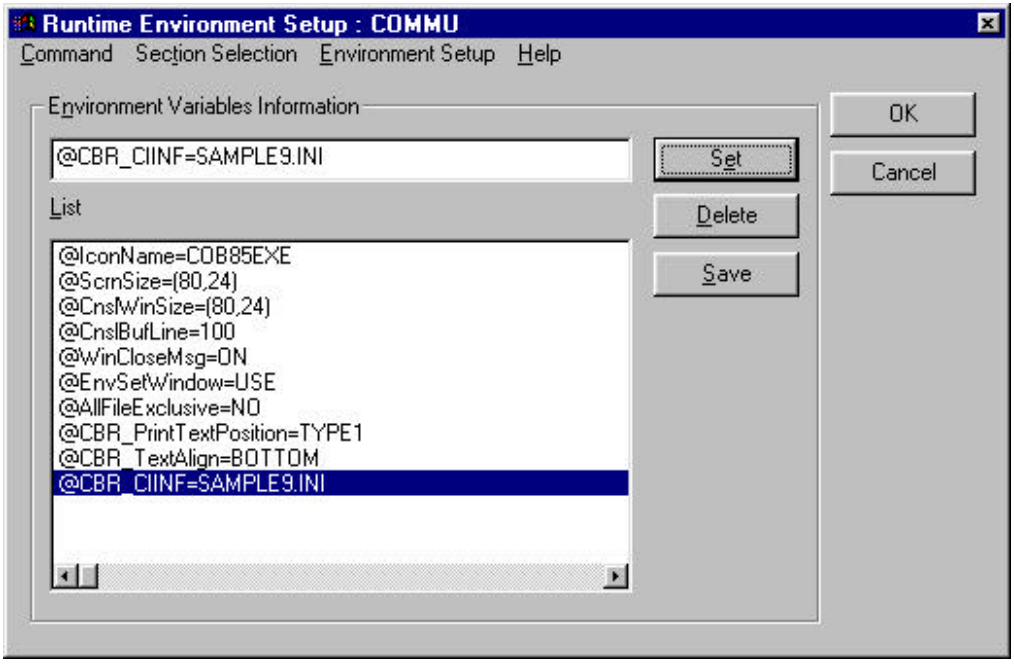


Figure 132. The Run-time Environment Setup window

Execution Procedure

Click on the OK button in the SAMPLE9 Execution Environment Information Setup window.

The message "*** EXECUTE COMMU.EXE ***" is displayed.

When SAMPLE9 is in wait mode, click on the OK button in the COMMU Execution Environment Information Setup window.

Execution Result

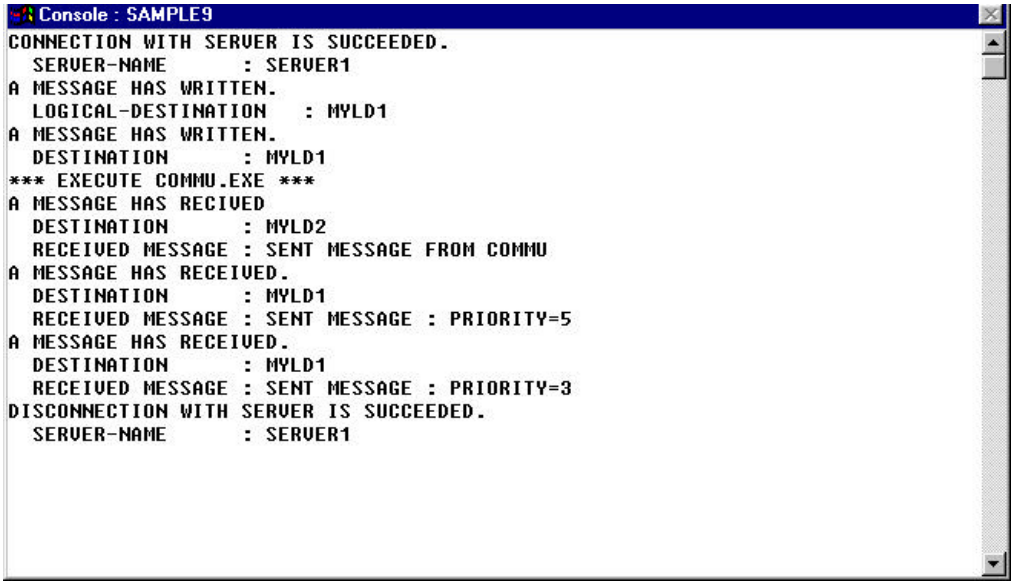


Figure 133. The SAMPLE9 console window

Check the messages displayed on the console to ensure the following:

In the SAMPLE9 console window:

1. Messages from COMMU have been read from logical destination "MYLD2".
2. Messages have been read from logical destination "MYLD1" in order of priority.

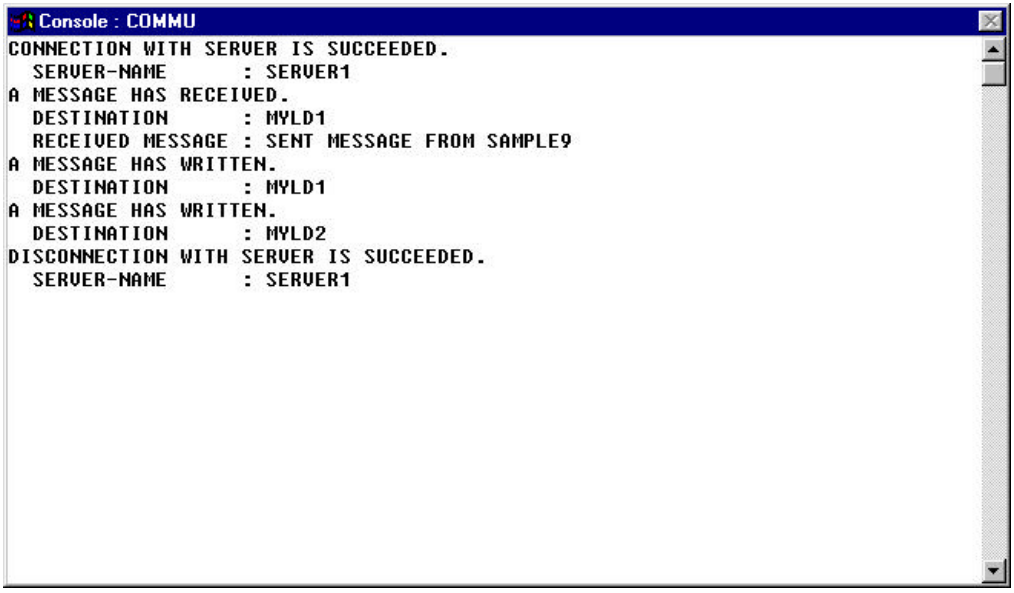


Figure 134. The COMMU console window

In the COMMU console window:

1. Messages from SAMPLE9 have been read from logical destination "MYLD1".
2. Messages have been written to logical destination "MYLD1" and logical destination "MYLD2".

Sample 10: Remote Database Access

Sample 10 extracts data from a database and assigns it to a host variable using the database (SQL) function.

The database is placed on the server and is accessed by the client via the ODBC driver. For more about database (SQL) using the ODBC driver, refer to the “COBOL85 User’s Guide.”

To run this sample program, the following products are required:

- Client
  - ODBC driver manager
  - ODBC driver
  - Products needed for the ODBC driver
- On server
  - Database
  - Products needed for accessing the database via ODBC

Function

The sample program accesses the database on the server and outputs all data stored in the database table "STOCK" to the client console. When all data has been referenced, the linkage to the database is disconnected.

COBOL Statements Used

DISPLAY, GO TO, and IF statements are used.

Embedded SQL statements (embedded exception declarations and CONNECT, DECLARE CURSOR, OPEN, FETCH, CLOSE, ROLLBACK, and DISCONNECT statements) are also used.

Before Executing the Program

Refer to Chapter 15, “Database (SQL) in the “COBOL85 User’s Guide” for more details on the following tasks:

- Build an environment which allows the database on the server to be accessed via the ODBC driver.
- Specify the default server and create a table named "STOCK" in the database on the server. This table must be in the following format.:

Table 5. STOCK table

GNO	GOODS	QOH	WHNO
SAMLLINT	CHARACTER 20 bytes	INTEGER	SMALLINT

Upper: Row name

Lower: Row attribute

You can store any data in the STOCK table. An example is given below.

GNO	GOODS	QOH	WHNO
110	TELEVISION	85	2
200	AIR CONDITIONER	4	1
380	SHAVER	870	3

- Create the ODBC information file with the ODBC information file set tool.

Program Compilation

The following figures show the setup items of the WINCOB window and the Compiler Options dialog box if C:\FSC is specified as the COBOL85 installation directory.

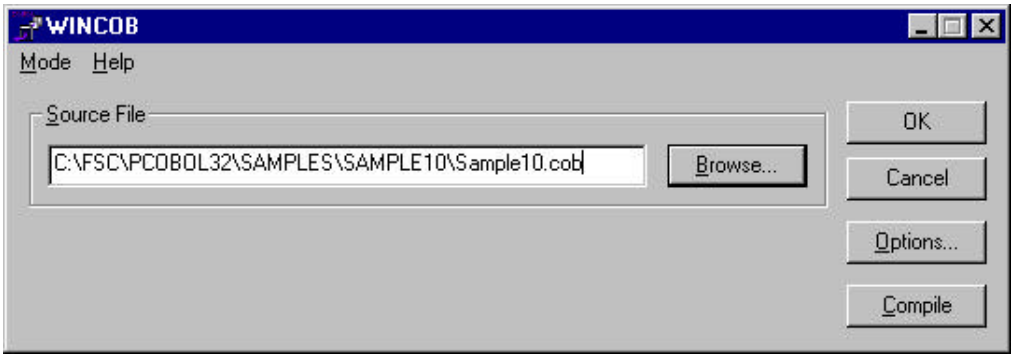


Figure 135. The WINCOB window

Specify the main program to the compiler option MAIN. Refer to Chapter 3, “A Quick Tour” for details on specifying compiler options.

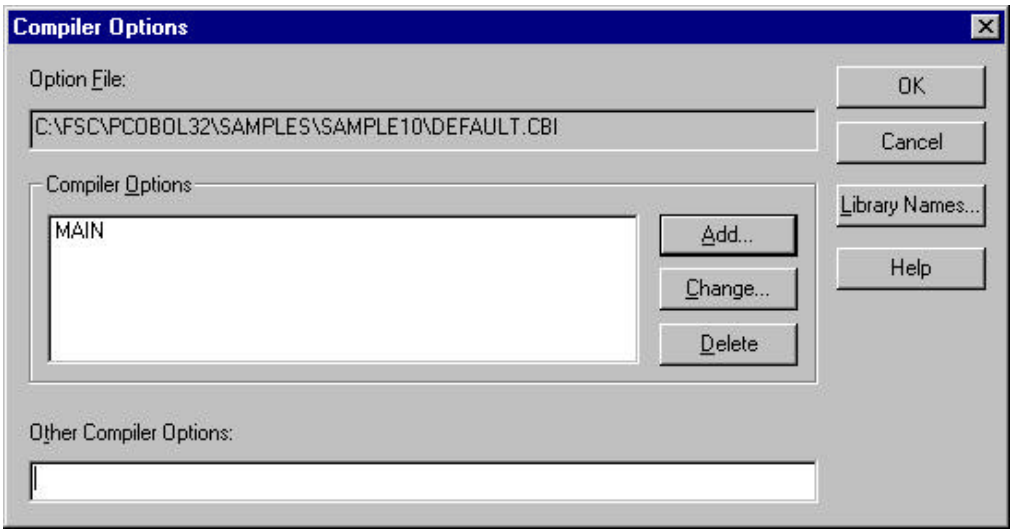


Figure 136. The Compiler Options dialog box with compiler option MAIN specified

Program Linkage

The following figure shows the setup items of the WINLINK [Linking Files] window for linking the object program (SAMPLE10.OBJ) created through compilation.

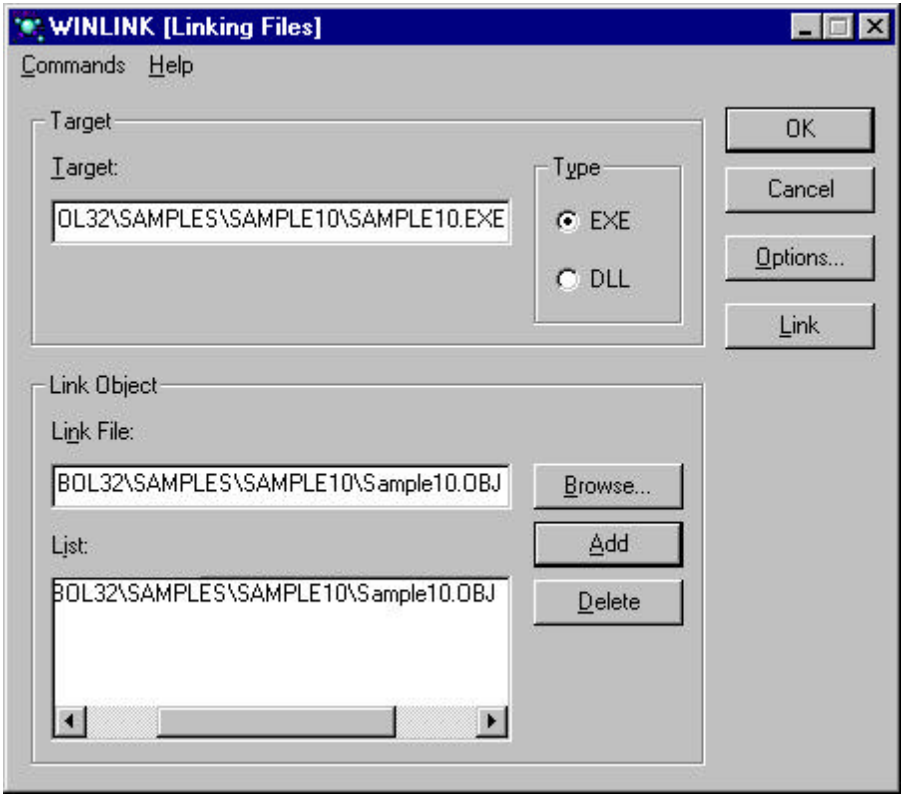


Figure 137. The WINLINK window

A target name is displayed when an object file is added. Refer to Chapter 3, “A Quick Tour” for details on using WINLINK.

Program Execution

The following figures show the setup items of the WINEXEC window and Run-time Environment Setup window for executing the executable program (SAMPLE10.EXE) created through linkage.

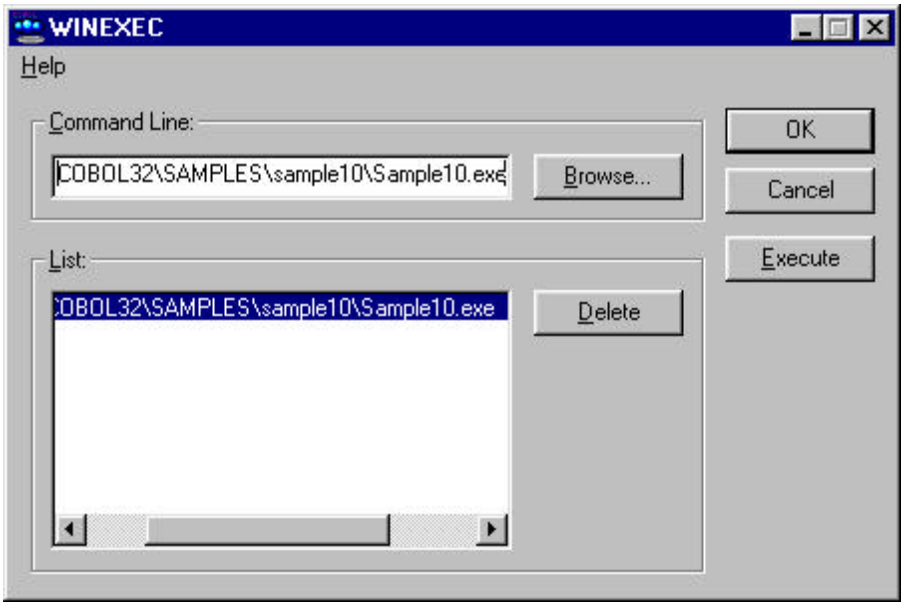


Figure 138. The WINEXEC window

Refer to Chapter 3, “A Quick Tour” for details on using WINEXEC.

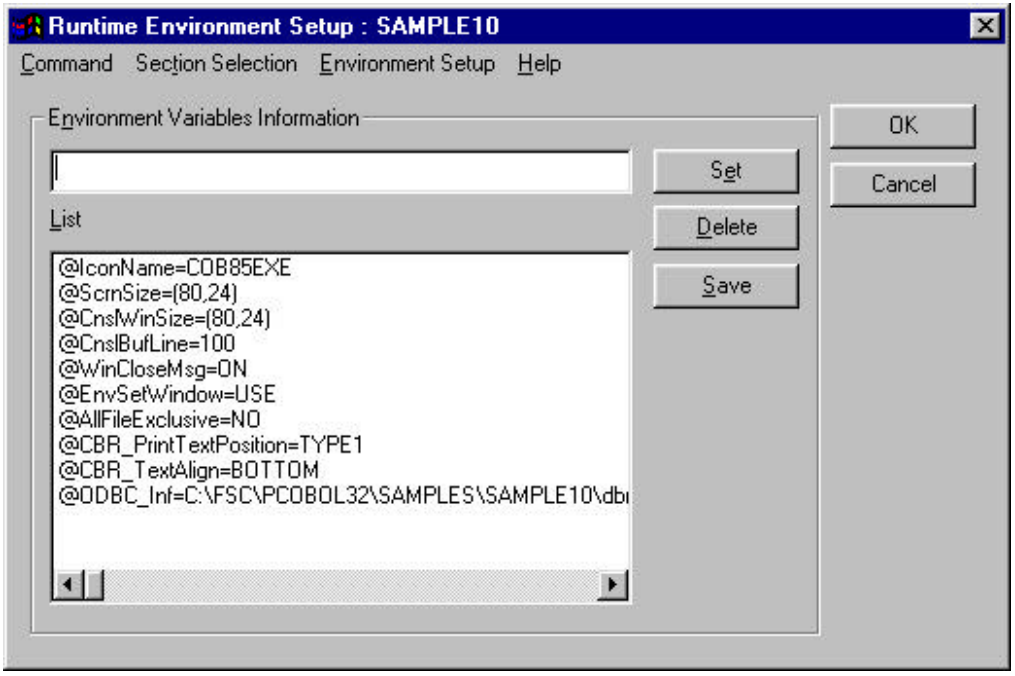


Figure 139. The Run-time Environment Setup window

Assign the ODBC information file name to "@ODBC_Inf".

Execution Procedure

Click the OK button in the Run-time Environment Information Setup window.

Execution Result

The data extracted from the table is displayed.

Index