CET W/32 Application Builder Users Guide

CET W/32 Application Builder Installation Guide

Published and printed in the United States of America.

First Edition	1994
Second Edition	June, 1995
Third Edition	July, 1995
Fourth Edition	December, 1995
Revision A	January, 1996
Fifth Edition	June, 1996
Sixth Edition	July, 1996

TABLE OF CONTENTS

CHAPTER 1

Installation Guide

Introduction

The CET W/32 Application Builder is a system of compilers, libraries, and utilities designed to produce 32-bit commercial applications for Microsoft Windows operating systems.

This chapter will provide you with the information you need to install the software.

Machine Requirements

The W/32 product is available as a Development System and a Runtime System. We recommend for

Development: A Windows 95 or NT system running on a 486 or Pentium, with at least 16 MB of memory. Approximately 4 MB of disk space is needed.

A Windows for Workgroups system may be used to develop 32-bit applications from the command line. Features such as the W/32 Integrated Development Environment are not supported due to problems in the Win32s product.

Runtime: A Windows for Workgroups, Windows 95 or Windows NT system running on a PC powered by a 486 or above, with at least 8 MB of memory. Note that you may also use Windows 3.1, but this platform is not recommended for heavy use on a large network.

Software Distribution

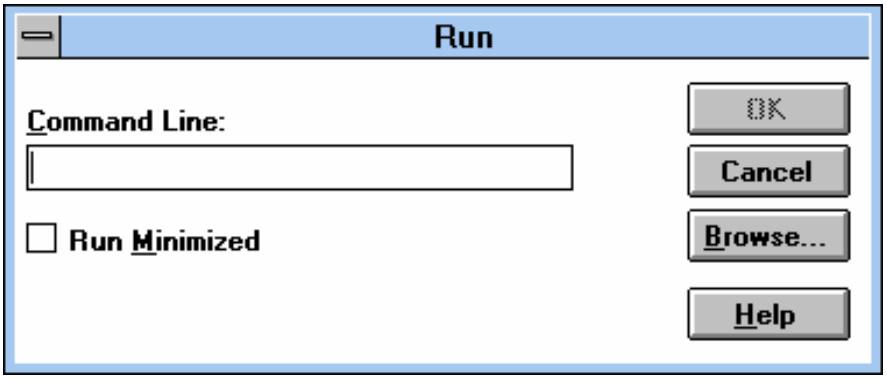
The W/32 Application Builder Development System is distributed on a set of diskettes. To simplify the installation at your customer's site, the Runtime System is provided on a separate diskette.

The Microsoft Win32s product is also included for use on any system running Windows.

Installing the Win32s Product (MS Windows Only)

The Win32s product provides you with the support system which permits any 32-bit application to execute under Windows for Workgroups or Windows 3.1. Therefore, if you are using either of these 16-bit operating systems to develop or run W/32 programs, you must have installed a copy of Win32s. It is not required for Windows NT or Windows 95.

This software **must** be installed first. To install Win32s, insert diskette 1 into drive A or B and select the Run command from the File menu in the Program Manager to open up a window like the one below:



Depending on which drive is being used, enter either "A:SETUP" or "B:SETUP" into the Command Line field and press the OK button:

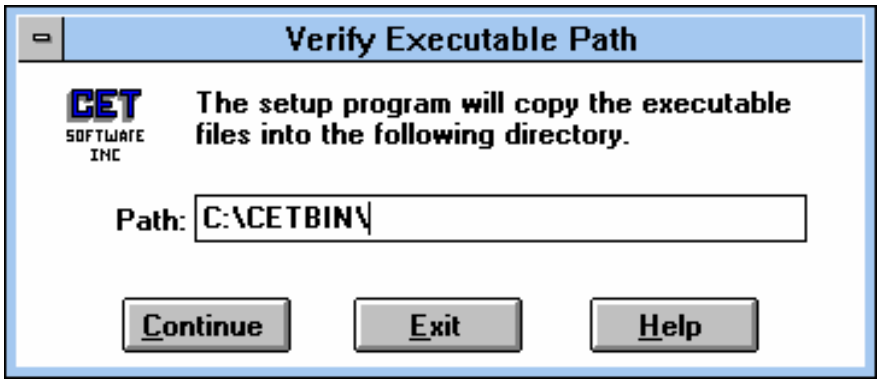
The SETUP command will install the Win32s components. At the appropriate time, you will be asked to insert the second diskette into the drive.

At the end of the Win32s installation, you will have the option of installing the Freecell game. We recommend that you install and launch the game to verify that the Win32s software has been installed correctly. (If Freecell does not run, the W/32 programs will not run either.)

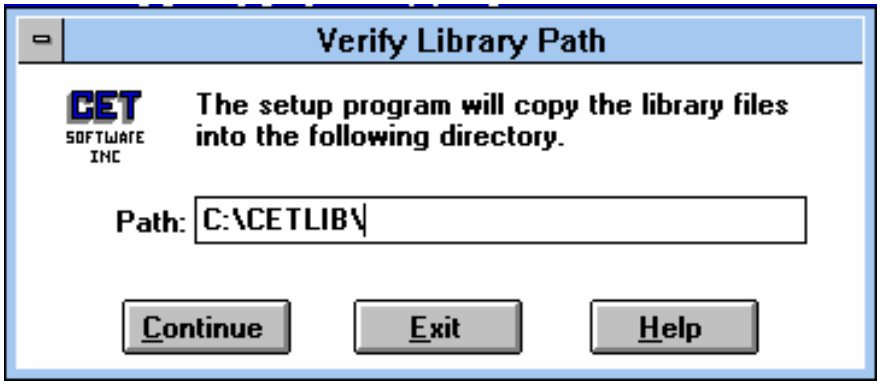
Installing the W/32 Application Builder

To install the W/32 Application Builder, insert diskette 1 into either drive A or B and select the Run command from the File menu in the Program Manager. Depending on which drive you are using, enter "A:SETUP" or "B:SETUP" into the Command Line field and press OK.

The SETUP program will ask you to indicate the directory to use to store the W/32 executables (except for **brtm.exe** which is covered below). We recommend that you use the default directory, **c:\cetbin**.



Next, the program will ask you to indicate which directory should be used to store the W/32 library components. We recommend that you use the default directory, **c:\cetlib**.



During the installation process, SETUP will search the Windows System directory for some Dynamic Link Libraries (DLLs) that are used by the W/32

Application Builder. These files will be installed if they do not exist. If an older version of a DLL is found, it will be saved with a **.bak** extension before the newer file is loaded. (If you attempt to compile with older versions of these DLLs, an error will occur.) When SETUP finds a file with the same or newer version, it will not be overwritten.

Currently, the following files are installed in the Windows System directory (normally **\windows\system** or **\windows\system32** for NT). Additional files may be installed with newer versions of W/32.

MFC30.DLL	backup: MFC30.BAK
MFCD30.DLL	backup: MFCD30.BAK
MSVCRT20.DLL	backup: MSVCRT20.BAK
CTL3D32.DLL	backup: CTL3D32.BAK
BRTM.EXE	Runtime module used by W/32

This procedure has been implemented in case you have installed other software that uses the same DLLs. Although your existing programs should continue to function correctly with the newer DLLs, remove the files that were replaced and rename the backup copies if you encounter any problems. (Please contact your CET distributor if you have a problem.)

Environment Variables

At the end of the installation, the program will indicate how to modify your **autoexec.bat** file to ensure that your W/32 Development System operates correctly.

During compilation, the W/32 Application Builder relies upon the PATH, LIB and INCLUDE environment variables to determine where to find the system components. These variables must be set to the directories specified during the installation process. For example, if you have used the default locations as recommended, update the PATH and set the LIB and INCLUDE variables in your **autoexec.bat** file as follows. The INCLUDE directory must be set to <*drive*>:\<*lib-path*>\INC, where <*lib-path*> is the same as the LIB directory.

```
SET PATH=C:\CETBIN

SET LIB=C:\CETLIB

SET INCLUDE=C:\CETLIB\INC
```

If you have installed a copy of CET BASIC for DOS and Networks on this system, make sure that you do not set the directory that contains the MASM files

used by the DOS product before **\cetbin** in the PATH. Otherwise, you will get an assembler error when attempting to compile a W/32 program.

Refer to the section on *Compiler and Runtime Environment Variables* in the *W/32 Compiler* chapter for information on how to set the other variables you need to develop and/or run your W/32 application.

Checking the Installation under Windows

To determine whether the W/32 software has been installed correctly, insert the CET KeyPlug into the parallel port, open an MS DOS window and enter the following command. The compiler should report the serial number in use.

```
obwin -V
```

Installing the W/32 KeyPlug Drivers for Windows NT

If you will be compiling and/or executing programs under Windows NT, you must install the device drivers to access the CET KeyPlug. (No special drivers are required under the other Windows platforms.)

The KeyPlug drivers are included with the W/32 Application Builder. Two drivers are required for Windows NT: a Rainbow driver for compiling and a Sentinel driver for running your application programs.

To install the drivers, log in as an *administrator* and perform the following:

Rainbow KeyPlug Activation

1. From the Program Manager, select the Main program group and then double-click on the Control Panel icon.
2. From the Control Panel window, double-click on the Drivers icon, and select Add.
3. When the line "Unlisted or Updated Driver" is highlighted in the Add window, select OK.
4. In the Install Driver window, type in the path to the KeyPlug drivers and press OK. This path will be **c:\cetbin\winnt** if you have used the default directory for the executables during installation. Otherwise, replace **\cetbin** with the path you have used.
5. Select the "Rainbow Port Driver" and press OK.
6. Select the Setup Actual button from the NT VDM Mapping window.

7. Select Edit from the Configure Actual Port window. Enter the address of the port you are going to use for the key into the Bus Address field, and select OK. Usually, the address for port1 (LPT1) is 378 and 278 for port2 (LPT2).
8. At this point, you should be back in the NT VDM Mapping window where you can select the down arrow in the Actual Port column associated with the VDM port address entered in step #7.
9. Select the port you are using from the Actual Port list and click the box in the Active Column for this row. After a check, select OK.
10. When the System Setting Change window appears, select the Don't Restart Now button so you can add another driver.

Sentinel KeyPlug Activation

11. From the Program Manager, select the Main program group and double-click on the Control Panel icon.
12. From the Control Panel window, double-click on the Drivers icon and select Add.
13. At this point, the line "Unlisted or Updated Driver" should be highlighted in the Add window. Select OK.
14. In the Install Driver window, type in the path to the KeyPlug drivers and select the OK button. This will be **c:\cetbin\winnt** if you have accepted the default directory for the executables during installation. Otherwise, replace **cetbin** with the path you have used.
15. Select the "Sentinel for i386 Systems" and select OK.
16. A Sentinel Driver window should now appear. Select OK.
17. You will now see a System Setting Change window. Select Restart Now and the drivers will be installed.

Checking the Installation under Windows NT

Insert the CET KeyPlug into the parallel port. To determine if your W/32 software is installed correctly, open an MS DOS window and enter:

```
obwin -V
```

The compiler should immediately identify itself and report the serial number.

The CET Bulletin Board System

The CET Bulletin Board System (BBS) is an on-line bulletin board used to store current releases, release notes, and technical memoranda. You may browse through the BBS and download any software you need.

You may also leave messages or problem reports for the CET support staff. Programs illustrating a problem may be uploaded to the SUPPORT library. If you submit a problem report via the BBS, please fax your CET distributor a Bug Report Form with all of the pertinent information regarding the problem so that we can respond as quickly as possible.

The CET BBS is available 24 hours a day, 7 days a week, except between 3:00 and 3:30 am (Mountain Time) when the system is down for maintenance.

CET Bulletin Board Systems are located in several places around the world. Please contact your local CET distributor for the number to use. The CET BBS in Boulder, Colorado, may be reached at:

(303) 530-9358

Logging onto the BBS

Use a communications package such as PROCOMM PLUS which provides common download protocols, such as ZMODEM, Kermit and Y-MODEM. We suggest using a 14.4K baud modem set with 1 stop bit, 8 data bits, and no parity. Do not use compression as this will tend to slow the download process. CET files are normally compressed before they are uploaded.

Once you have entered the CET BBS, you must supply an account ID and password. If you do not have an account, enter "new". You will be asked to supply some information about you and your company, in addition to the ID and password you wish to use.

After you have finished the login procedure, you can traverse the system. Menus and on-line help simplify the process.

Downloading Files

CET products are easily downloaded from the BBS. Check the following libraries for the current W/32 App Builder software and the other products you may need:

Library	Products
WIN32	W/32 Compiler and runtime software.
SHWARE	DOS shareware such as **pkzip\pkunzip** and **tar.**

The WIN32 library contains technical notes and sample programs in addition to the latest software release. If you select the options to "Find Files" by "Date" in the "Current library", the most recently added files will appear first on the list.

Most of the software in these libraries will have a **.zip** extension. These files are in a compressed format so that they may be downloaded quickly.

After downloading a compressed file, you will have to unzip it. If you have downloaded a software upgrade, follow the installation instructions in the release note. Otherwise, change to the directory where you want to store the software and entering the following command replacing *filename.*zip with the name of the specific **.zip** file you want to decompress.

PKUNZIP -o *filename.*zip

The -o flag will automatically replace any existing files with the same name. Omit the flag and PKUNZIP will prompt you before replacing each of the files.

If you do not have a copy of PKZIP, download **pk204g.exe** from the SHWARE library. Move the file to a directory in your PATH (e.g. **cetbin**) and execute **pk204** to *explode* the software into its individual components. After that, you will be able to use PKZIP to compress your own files.

Please contact your CET distributor if you have questions about using the CET Bulletin Board System or any of the files that have been uploaded.

CHAPTER 2

An Overview

Introduction

The W/32 Application Builder is designed to produce 32-bit commercial applications for the Microsoft Windows operating systems. W/32 differs from most MS Windows development packages in that it provides the tools to methodically convert existing third-generation programs into modern Windows applications with a graphical user interface.

The Base Language

Microsoft and other companies who market development tools have rediscovered BASIC as the best *base* language to use to express routines, macros and programs. The BASIC language has grown in power and sophistication, while retaining its ease of use.

The W/32 Application Builder also uses BASIC as its base language. Designed first for use under XENIX, UNIX, DOS and network operating systems, CET BASIC is a rich language with excellent facilities with which to develop commercial applications.

CET BASIC is easy to learn, and easy to grow with. Even statements that perform very complex operations are easy to write.

CET BASIC includes sophisticated control structures; a large number of data types; all useful forms of file types and operations, including built-in ISAM file processing; hundreds of functions for string handling, data conversion, screen control, output formatting; error trapping and recovery.

Unlike many interpretive systems, CET BASIC provides high performance during execution. The W/32 compiler generates very efficient 32-bit code which will run natively under Windows, Windows NT and Windows 95.

CET BASIC is commercially oriented, and includes such features as:

- Sophisticated sequential, direct and indexed file operations with built-in file and record locking, lock detection and shared file control.
- Decimal data types and arithmetic with floating point support.
- Full matrix arithmetic and file I/O.
- COBOL-like formatting for screen display and printed reports.
- Program chaining to produce structured, easy-to-maintain applications.
- Complete error detection and trapping.
- Easy access to operating system commands and externally compiled subroutines in C, BASIC and assembler.
- Full C preprocessor support.
- Multi-platform support under DOS, networks, Windows and UNIX.

CET BASIC is used internationally. Complete control over number and date formatting is provided. Error messages are stored in external files so that they may be easily translated.

The W/32 Windows Framework

The W/32 compiler takes conventional CET BASIC applications and links them into a 32-bit Windows framework. That means your existing CET BASIC programs may be compiled and executed normally without modification, and then improved methodically to include other Windows features.

By default, a W/32 application begins by opening a text window, such as:

The text window provides a familiar text-oriented screen to interact with existing CET programs. The default menu commands allow you to select specific fonts, display the default toolbar and status bar, and look at a Help-About window.

The W/32 Application Builder provides you with a powerful tool for improving an application's "look-and-feel" without having to modify your source code. By default, all W/32 applications are linked with a Windows Framework which controls the display font, the number and types of menu items, the status bar display and the tool bar. The *framework* may be modified (e.g. to display a custom menu) and the new feature will be automatically used in every program you compile.

A special compiler flag may be used to turn your existing input fields created by LINPUT, LINPUT USING and GET statements into Windows edit controls (white boxes with black frames) to give your application more of a Windows look.

Refer to the W/32 Compiler chapter for information on using these special edit controls. The Windows Framework is covered in its own chapter.

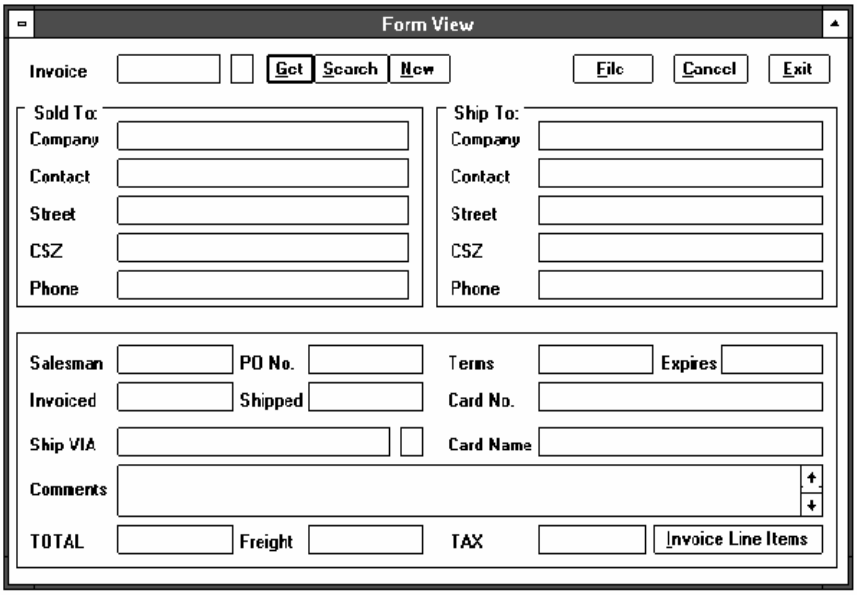
The W/32 Integrated Development Environment

The W/32 Application Builder provides you with an integrated graphical development environment with all the functionality you need right at your fingertips. Without leaving the program, you can create

- √ Projects to build, test and maintain the executables in your application.
- √ Source files using a special text editor designed for writing large programs. The W/32 Debugger is available to assist you in finding any errors that may occur during program execution.
- √ GUI input forms and dialogs like those found in other Windows products.

Because of the variety of features that are available, they will be covered in three separate chapters; the *W/32 Integrated Development Environment*, the *W/32 Debugger* and the *W/32 Dialog Editor*.

The following screen is the sample form used to illustrate some of the features covered in the *W/32 Dialog Editor* chapter:



Both form views and dialog boxes consist of a group of controls that either display text or get input from the user. Each type of control has a set of properties that define its behavior.

A form view is a GUI input screen that will be used to replace the main, text-based application window. Although the window will automatically be full-screen or set to the size defined with the Dialog Editor, it may be moved and resized by the program or the user.

Any time the form view is active, you have full access to any of the menu bar commands and/or toolbar buttons. A form may be minimized or maximized.

A dialog box is similar to a form view except that the dialog must be closed (usually by clicking OK or Cancel) before the user can continue working with the rest of the application.

You can create a wide variety of dialogs. A simple dialog box is a message box that displays some text and waits for the operator to select a button. A complex dialog box could display a number of input fields and await the entry of data from the user by means of the mouse or the keyboard.

Basically, dialog boxes may be used anytime you want to get information to or from the operator without affecting what is currently on the screen or in other active dialog boxes. In other operating system environments, you may have

opened a window (using the Phase One Window System), displayed some text, performed an operation such as LINPUT and then closed the window when you were ready to continue. Under Windows, this would typically be done using a dialog box.

Whether you design a form view or a dialog box, the W/32 Dialog Editor will generate a BASIC program that may be customized to meet your specific needs.

BASIC Subroutines

The W/32 Application Builder provides you with special statements that may be used to link program modules that call BASIC subroutines into a single executable. For example, a subroutine header has the following format:

$SUB *module-name*

$SUB defines a subroutine that can be called by the *main* BASIC program or by another BASIC subroutine.

Two methods are provided so that data can be shared between BASIC subroutines. They are:

1. Any variable referenced within the same source module or **.b** file refers to the same variable. In other words, all variable names are global between BASIC subroutines that reside in the same source file.
2. Any variables that are used in separate source modules are distinct unless they are defined by the $GLOBAL statement. This is true even for variables with the same name.

This model for *name space* management allows you to write and compile separate BASIC subroutines (to be linked into one executable), without worrying about a possible collision between the names of variables in the separate modules.

CET BASIC recognizes other statements which include $GLOBAL to define global data, $CLEAR to clear all variables and dimensioned arrays and $EXIT to return from a BASIC subroutine.

You will find that being able to use BASIC subroutines is a powerful programming feature. Please refer to a separate chapter in this manual for details.

Mouse Support

The mouse is the most important data entry device for Windows applications. Since it provides an intuitive way for a user to make selections and invoke actions, mouse support is automatically built into any form or dialog created with the W/32 Dialog Editor.

Although mouse input is normally difficult to integrate into existing keyboard-oriented applications, W/32 provides you with a powerful method for handling asynchronous mouse events. The W/32 mouse handler detects events and responds immediately by calling specific BASIC subroutines such as:

- cetLBtnDn
- cetLBtnUp
- cetLBtnDClk
- cetRBtnDn
- cetRBtnUp
- cetRBtnDClk
- cetMouseMove

These subroutines are called when the mouse buttons are depressed or released, double-clicked and/or when mouse movement is detected. Your program can take any action desired upon detecting a mouse event or set global variables for use by other routines at a later time.

Even existing applications which use INPUT, LINPUT or GET to obtain user input can be improved with mouse detection. By calling the W/32 function called cInputQ, a program can preload the keyboard input queue with responses. Using this method, a mouse event can cause keystrokes to be forced into the input queue, which a LINPUT or INPUT statement may acquire later.

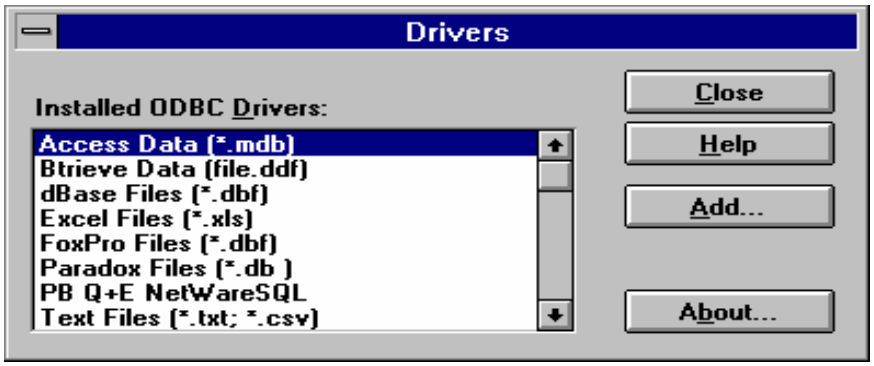
Image Display

W/32 provides routines that may be used to display, position and resize image files (in a **.bmp** or **.dib** format) during the execution of a W/32 application. This feature permits you to display photos, artwork and other images in a window without leaving your program.

ODBC Support

The Open Database Connectivity (ODBC) facility is a standard created by Microsoft to provide transparent access by applications to a wide range of database and file management systems.

ODBC drivers exist for MS Windows file managers (e.g., Dbase, Btrieve), database managers (e.g., FoxPro, Paradox and MS Access) and SQL servers such as Watcom SQL, and Gupta SQL servers. Full relational databases such as the MS SQL Server, Oracle, Informix and Sybase also offer ODBC drivers for desktop connectivity.



To use ODBC with a W/32 program, you must have the correct ODBC driver installed on your system and a data source configured according to the documentation for the driver. Then, a W/32 program can allow the user to specify a data source or open a data source directly via the OPEN statement.

The standard CET BASIC READ statement may be used to select the set of rows desired from the opened table. The names of the columns (fields) in the table are returned as the data fields, which may then be used to construct further queries via READ. The READNEXT statement is used to access each row that has been read. WRITE and MAT WRITE statements may be used to update the table.

W/32 Libraries

The CET Commercial Development Libraries are provided to augment the power and flexibility of CET BASIC. These libraries contain a wide variety of functions which may be linked into your program to perform operations which would be difficult or inefficient to code using BASIC alone.

The POS Window Library

CET Software has implemented the Phase One Window System to provide compatibility with the CET products for DOS and UNIX-based systems. If you are porting existing POS windows programs, we suggest that you replace your

text-based windows with dialog boxes designed using the W/32 Dialog Editor as the first step in creating your MS Windows product.

The CET Bar Code Library

The CET Bar Code Library provides the subroutines necessary to generate and print bar codes from a CET BASIC program. This is accomplished by using a simple CALL statement to the Bbarcode function.

With the CET Bar Code Library, you can generate any of the following types of bar codes:

- UPC-A and E
- 2 & 5 digit supplementals
- Code 39 and Extended 39
- Code 128 & UCC-128
- Code 93 and Extended 93
- EAN/JAN 8 and 13
- Interleaved 2 of 5
- Codabar
- MSI Plessey Standard
- Zip+4 Postnet

The Bbarcode function returns a string that will produce the correct bar code when directed to one of the following printers:

- Postscript
- IBM Proprinter 9 pin
- HP Laserjet at 300 dpi
- Toshiba 24 pin
- IBM Proprinter 24 pin
- HP Deskjet 300 dpi
- HP Laserjet 150 dpi Compat Mode
- Epson 9 pin
- HP Laserjet at 150 dpi
- Epson 24 pin
- Okidata Microline Standard
- HP Deskjet 150 dpi
- HP Paintjet 180 dpi
- HP Laserjet 300 dpi Compat Mode

The CET File Access Library

CET BASIC may be linked to C language routines by following the procedure documented in the *CET C Interface Guide*.

Since CET BASIC data files do not use the native access methods of the underlying operating system, it is difficult to write external subroutines which perform file operations. The CET File Access Library provides you with functions which are equivalent to the CET BASIC OPEN, CLOSE, READ, WRITE, and UNLOCK statements .

Using these library functions will ensure that file accesses performed by an external C subroutine and the calling CET BASIC program are synchronized.

All record and file locking is enforced by the library functions, and the external subroutine can detect locked and end-of-file conditions. The functions use the following syntax:

```
fclose(int channel)
int fwrite(int channel, char *buffer, int len)
long ftell(int channel)
void fseek(int channel, long offset, int base)
int fread(int channel, char *buffer, int len)
void finfo(char *filename, struct sfinfo *fi )
char *fgets(char *buf, int max, int channel)
int readp(int channel, char *key, char *buffer)
int readn(int channel, char *key, char *buffer)
int readk(int channel, char *key, char *buffer)
void unlock(int channel)
```

The CET File Access Library consists of at least two separate libraries: one library is designed for linking the functions to C main routines, and the other is for linking to CET BASIC programs. The Microsoft C compiler is required to use these features.

Future Releases

CET Software has planned the development of additional libraries to support the needs of commercial applications developers. If you find that you need a feature that is not currently provided, please contact your CET distributor to find out about its future availability.

CHAPTER 3

The W/32 Compiler

Introduction

The W/32 Application Builder is a system of compilers, libraries, and utilities that produces 32-bit Windows applications. This chapter will discuss how the compiler and runtime system may be affected by setting a variety of compiler flags and environment variables.

All of the compiler features covered in this chapter may be used from the command line. If you are developing under Window 95 or NT, we recommend that you use the W/32 Integrated Development Environment to create, debug and maintain your applications. (Refer to a separate chapter for details.)

The W/32 Components

The W/32 development system includes the OBWIN Compilation Manager, the compilers (**bb16.exe** and **bb32.exe**), a runtime system (**brtm.exe**), and the Windows Framework or resource files (**cetwin.obj** and **cetwin.res**).

The product is shipped with a number of libraries. Some libraries provide support for the CET BASIC language and built-in functions. Others provide additional functionality such as bar code printing and ODBC. The W/32 compiler will recognize demands for any of the standard libraries automatically, and ensure that a single executable is linked correctly.

The Windows Framework is a powerful component of the development system which allows you to implement Windows features in your application without modifying your source code. (Please refer to the *Windows Framework* chapter for details on using the resource files and the chapter on the *W/32 Window Functions* for more information on the specific features that are available for your use.)

The W/32 Application Builder comes with two compilers. While both produce 32-bit executables, one compiler is a 16-bit program designed for development systems running Windows for Workgroups and the other is a 32-bit compiler for use under Windows NT and Windows 95. A compiler flag must be used to direct OBWIN to use the 16-bit version.

If you have been using any of the CET UNIX-based compilers, you are familiar with the advantages of a 32-bit compiler. On the other hand, if you have been using an MS-DOS compiler, you may have found that you did not have enough data or text (code) space for some of your larger programs.

The W/32 32-bit compiler supports 4GB of flat address space so that the total amount of text and data space can be as large as 4GB. Technically, the size of an individual array is unlimited and could be dimensioned for 200,000 elements.

The only limitation (besides the maximum of 4GB) is the combined amount of real and virtual memory (the SWAP file). For instance, suppose you have 16MB of memory, and are trying to run a program that requires 20MB. In that case, the unused programs and the least recently used portions of your program will be swapped to disk. While this will affect performance, it means that you can run programs that are larger than your main memory.

Creating W/32 Programs

The W/32 Compiler accepts source programs which are stored in an ASCII format since text files are the easiest to port to and from other operating systems. Refer to the *THEOS Compatibility Guide* for specific information if you are porting BASIC source files from that environment.

To import an existing CET BASIC program, simply copy the program source files to the desired directory. If your source code is on an MS DOS system with Windows support, you do not need to move the file before recompiling.

New W/32 programs may be created using any text editor, but we recommend that you use the W/32 Integrated Development Environment which has been designed for that purpose. For specific programming information, refer to the *CET BASIC Language Reference Manual.*

For now, it is only important to know that all CET BASIC programs must have a **.b** extension. The statement lines do not have to be numbered.

If you wish to practice using the features in this chapter, we suggest that you use any text editor and create a program called **tst.b** that contains the line PRINT “Hi”.

Compiling W/32 Programs

The OBWIN Compilation Manager is a multi-purpose program designed to invoke the compiler (**bb32** by default), link the program to the required libraries and create a 32-bit Windows executable with the extension **.exe**. A flag is available to invoke the 16-bit compiler under Windows for Workgroups.

OBWIN operates similarly to cc, the UNIX C language compiler. This syntax is supported to provide compatibility between the CET BASIC products for Windows, DOS and UNIX.

The syntax for OBWIN is shown below. Note that the square brackets indicate an optional item. The names of the program source files and/or object modules are *italicized* to indicate that they are variables to be entered by the operator. Input the names in any order and in any case mode. W/32 does not distinguish between upper or lower-case file names.

obwin [–flags] [*pgm*] [*B-pgm*] [*C-pgms*] [*A-pgms*] [*O-pgms*]

–flags

is any valid combination of flags or switches used to control the operation of the compiler. Entering OBWIN without any parameters will list the currently supported flags and their function to the screen.

The following example will use the 16-bit compiler to create an executable with the name **newpgm** and the default extension of **.exe**. Note that the name of the executable program must immediately follow the -o flag.

```
obwin -16 -o newpgm newpgm.b
```

pgm

is the name to be used for the executable. If no name is specified, OBWIN will use the default name **cetwin.exe**. For example, the following command will produce **cetwin.exe** using the default 32-bit compiler.

```
obwin newpgm.b
```

B-pgm

is the name of the BASIC source file to be compiled. This must be an ASCII formatted file with a **.b** extension.

C-pgms

is the name of one or more C language programs to be compiled. To use this feature, the program must have an extension of **.c**. The Microsoft C compiler version 8.0 must also be available on the system.

Note that W/32 comes with a variety of C language functions. These functions are covered in a separate chapter in this manual and in the *CET BASIC Library Manual.*

A-pgms

is the name of one or more assembler programs with an extension of **.asm**. These programs must be in a Microsoft MASM format. (Refer to the -S compiler flag description.)

O-pgms

is the name of one or more object modules with an extension of **.obj**.

The Compiler Flags

Compiler flags may be used to control the operation of OBWIN. Enter the command without any parameters to list the flags and environment variables that are supported. Each of the compiler flags are described in this section.

It is important to note that the compiler flags must be entered in exactly the same case mode as shown in this manual.

16-Bit Flag (-16)

The -16 flag should be used under Windows for Workgroups to direct OBWIN to use the 16-bit compiler. When omitted, the 32-bit compiler (**bb32.exe**) will be used, by default. For example:

```
obwin -16 -o newpgm newpgm.b
```

32-Bit Flag (-32)

The -32 flag may be used under Windows NT and Windows 95. This is the default behavior so the flag may be omitted.

Assembler Output Flag (-S)

The -S flag may be used to inform the compiler to terminate after the assembly language program is generated. For example, to compile **newpgm.b** and terminate after saving **newpgm.asm** in the current directory, enter:

```
obwin -S newpgm.b
```

Assembler programs will be output in a Microsoft MASM format.

Define Symbol Flag (-D*symbol*)

When your BASIC program contains C preprocessor statements such as #ifdef, this flag may be used to define a value for the preprocessor symbol. (This feature is covered separately in the *CET BASIC Language Reference Manual.*)

Error Line Selection Flag (-#)

The -# flag may be used to specify how to report runtime errors. When used, ERL will display the actual line number in the source file that caused the error to occur. Otherwise, the default is to display the number of the last numbered line that was executed successfully.

This flag is particularly useful when a BASIC program contains statements both with and without line numbers. For example, the following code will terminate with ERL = 120 if the -# flag is not used. Otherwise, ERL will equal 2.

```
100     X = 0 \ GOTO 120
BADLINE: PRINT 1 / X
120     GOTO BADLINE
```

Floating Edit Control Flag (-Zi)

This is a 32-bit compiler feature which may be used to make a W/32 text-based program look more like a Windows application with little or no modification to the BASIC source code. Although this feature in unavailable when developing under Windows for Workgroups, the compiled programs will run under that environment.

The sample text-based program documented in the *Appendix* illustrates how this feature may be used. To compile a program you would enter:

```
obwin -Zi -o client client.b
```

When you run the program, a white box with a black frame will appear each time a LINPUT, LINPUT USING or a GET statement is executed. The box will disappear (leaving the data on the screen) as the operator presses the ENTER key to move to the next statement that gets some input. (The CLIENT demo displays a white box without a frame for the inactive fields.)

This box is actually a W/32 edit control which means that the mouse can be used to position the cursor and select characters to be deleted. Pressing TAB will automatically select any existing characters. In fact, the edit control will operate like in other Windows applications except that the HOME and END keys will not reposition the cursor (so the code that uses their INP values will still work).

The width of the box is determined by the USING mask. If your program uses LINPUT statements, without USING, a box 16 characters wide will be displayed, by default. In this case, the horizontal scroll property is set on. (This property is set off for LINPUT USING edit controls.)

The cLinputOpts function is available to override the default, 16-character length. When used, it will affect all subsequent LINPUT statements up to another call to cLinputOpts. It will have no effect across a CHAIN or LINK.

Since this function may only be used with the floating edit control feature, it is covered here instead of in the *W/32 Window Functions* chapter. The syntax of the CALL statement is:

```
call cLinputOpts( len%, use3D%, all% )
```

Where:

len%

is the *new* default length for non-LINPUT USING statements. (This value is ignored with LINPUT USING.

use3D%

is set to 1 to make the edit control 3-dimensional. Note that these controls are taller than a line of text, and will overlay text that may be directly above or below the current line. (Controls are 2-dimensional, by default.)

all%

is set to 1 to select all text for editing if you are using a literal mask. That means that the text will be highlighted in blue as soon as the control appears.

We also recommend setting the default screen colors to black on gray in the **w32app.w32** file:

```
B_DFLTFGC=0
B_DFLTBGC=7
```

Include Path Flag (-I*path*)

This flag may be used to specify the directory that contains the files referenced by the #include preprocessor statement. The name following the -I flag will be passed to the W/32 Compiler and the resident C compiler.

For example, if **newpgm.b** contains the statement #include "getfld", you could use the -I flag to direct the compiler to search for **getfld** in **\include**:

```
obwin -I\include -o newpgm newpgm.b
```

Note that the Microsoft C compiler is not required when the #include statement is the only preprocessor directive that is used. (Refer to the *CET BASIC Language Reference Manual* for information on using the C preprocessor.)

Library Flag (-l*name*)

Normally, the linker program searches the current working directory for a runtime library before checking the directory specified by the LIB variable (**c:\cetlib** by default). The library flag (a lower case "L") may be used to pass the name of an additional library that contains the required compiled or assembled functions to the linker.

For example, the W/32 utility called CETLNK32 may be used to create a library of object files (produced with the Dialog Editor) named **myobj.lib**. If you CALL these subroutines from another BASIC program, you would direct the compiler to search this library by entering:

```
obwin -o newpgm newpgm.b -lmyobj
```

Line Number Flag (-Ln)

The -Ln flag may be used to suppress the default line number check. Otherwise, an error will be detected if the line numbers of the BASIC statements are out of order or invalid.

Object Output Flag (-c)

The -c flag may be used to compile source programs and terminate after generating the object module. If this flag is omitted, the intermediate object module will be erased after the program is linked. For example, the following command will compile a BASIC subroutine created with the W/32 Dialog Editor and terminate after producing the file **dlgcet.obj**:

```
obwin -c dlgcet.b
```

The next command will compile the C program **cpgm.c** and the BASIC program **bpgm.b**, then terminate after saving **cpgm.obj** and **bpgm.obj** in the current working directory:

```
obwin -c cpgm.c bpgm.b
```

If C language programs are to be compiled, OBWIN will invoke the Microsoft C compiler. Version 8.0 (or higher) is required to use this feature.

Output File Flag (-o *executable*)

The -o flag must be used to create an executable program with a specific name. For example, the following command will compile the BASIC program **newpgm.b** and produce the executable **newpgm.exe**.

```
obwin -o newpgm newpgm.b
```

If the output flag is omitted, OBWIN will assign the name of **cetwin.exe**, by default. The following will compile **newpgm.b** into **cetwin.exe**.

```
obwin newpgm.b
```

Preprocessor and Preprocessor Only Flags (-P and -PO)

If the Microsoft C Compiler is in a directory set by the PATH environment variable, the full range of C preprocessor commands may be used including #define and macro substitution, and conditional compilation with #if.

The C preprocessor may be invoked with the -P flag. If the -PO flag is specified, only preprocessing is performed and the result is stored in a file with the same name as the source file and the extension **.i.**

This feature also permits variables to be defined and assigned values on the command line as in the following:

```
obwin -P -Dabc -Ddef=xyz newpgm.b
```

This command would be equivalent to entering the following lines at the top of the source file **newpgm.b**:

```
#define abc
#define def xyz
```

The C preprocessor is a special feature that is covered in a separate chapter in the *CET BASIC Language Reference Manual.*

Report Flag (-R*step*)

The report flag is available for debugging purposes when you need to know the exact command lines used in each of the steps invoked by OBWIN. The letter following the flag indicates which step to report:

-RB	BASIC compilation	-RC	C compilation
-RA	assembly	-RL	link

Resource Flag (-r *name*)

The resource flag is used to compile a modified resource file with the specified ***name*.rc** into a file with the same name and the extension **.res.** Because the -r flag may only be used to compile a resource file, the .**rc** extension is assumed and should not be specified on the command line. For example:

```
obwin -r myuser
```

Refer to the *Windows Framework* chapter for information on how to use the features in the resource file.

Trace Code (-T[*trace-sub*])

The -T flag directs OBWIN to include line number tracing code when compiling a program that contains an ON ERROR statement. For example:

```
obwin -T -o myprog myprog.b
```

The -T flag may also be used with the name of a C subroutine. In the following example, **mysub.c** is linked into the BASIC program **myprog.exe** so that it is automatically called after each BASIC statement.

```
obwin -Tmysub -o myprog myprog.b mysub.c
```

Note that this feature is useful if you are compiling programs from the command line. The W/32 Integrated Development Environment comes with its own source level debugger which may be used to simplify this process (under Windows 95 and NT).

UX-BASIC Flag (-U)

This flag may be used to indicate that all data files used in the program are to be opened, read, and written in UX-BASIC compatibility mode. When this mode is specified, files written by W/32 BASIC programs may later be read by UX-BASIC programs and vice versa.

Using this flag is equivalent to entering the statement "OPTION MATIO 0" at the top of the program and opening each file with the option UX.

Version Number Flag (-V)

The -V flag may be used to display the W/32 compiler version and serial number on the screen. For example:

```
OBWIN -V
```

Windows Object Flag (-wo *name*)

This flag may be used to suppress the default which is to link to the W/32 file called **cetwin.obj** during compilation. If a copy of this file has been customized using the procedures in the *Windows Framework* chapter, then the file with the specified *name* will be used instead.

In the examples below, the first command will create an object file with the name **mywin.obj.** The second command will link the file (without the .**obj** extension) into the executable **test.exe.**

```
obwin -c mywin.b
obwin -o test test.b -wo mywin
```

Windows Resource Flag (-wr *name*)

The W/32 compiler will automatically link with the Windows Framework file **cetwin.res** to generate 32-bit programs with certain default Windows features. The source code for the features that may be modified is found in the resource file called **cetuser.rc**.

If a copy of the **cetuser.rc** file has been customized using the procedures in the *Windows Framework* chapter, then this flag will indicate that the file with the specified *name* should be linked into the executable.

In the following example, the first command will compile a new resource file called **myuser.res.** The second one will link the resource file into the executable **tst.exe**. Note that the extensions **.rc** and **.res** are assumed, and should be omitted from the command line.

```
obwin -r myuser
obwin -wr myuser -o tst tst.b
```

Environment Variables Used by the Compiler

Environment variables may be used to pass additional information to the W/32 compiler. These variables must be set in the **autoexec.bat** file (along with LIB and INCLUDE) in order for them to be recognized during the compilation process. Use the following syntax to set an environment variable:

set *variable = value*

B_CETBIN

Set to the path to search for the CET W/32 system components. For example, if you installed the executables in the default directory you should enter:

```
B_CETBIN = C:\CETBIN
```

B_CETINC

Set to the path for the CET W/32 include directory. For example, if you installed the library files in the default directory you should enter:

```
B_CETINC = C:\CETLIB\INC
```

B_CETLIB

Set to the path for the CET W/32 include directory. For example, if you installed the library files in the default directory you should enter:

```
B_CETINC = C:\CETLIB
```

B_CMDFLAGS

Set to the compiler flags to be enabled for each compilation. For example, if you use the (optional) Phase One Toolkit library, you could enter the library flags to indicate where the compiler can find the routines.

```
SET B_CMDFLAGS = “-lPostk”
```

The following example illustrates how you can set B_CMDFLAGS so that OBWIN will always use the Phase One Toolkit library and the customized Windows Framework files (located in the current working directory). Separate multiple flags with a space as in:

```
SET B_CMDFLAGS = “-lPostk” “-wr myres” “-wo myobj”
```

When this variable is used, the compiler will indicate that it is enabling the default flags and display their values.

B_THCONV

Set to 1 to convert string literals to THEOS values. This will ensure that converted codes for international characters (with ASCII values greater than 128) are output by the assembler.

Environment Variables Used by the Runtime System

A wide variety of environment variables may be used to modify the behavior of the runtime system. For example, if you have ported your application from the THEOS operating system and you want the W/32 programs to operate in the same way, set the following variable:

```
B_EMULATE=THEOS
```

Runtime variables such as B_EMULATE must be set in a special initialization file. When an application first starts, the Runtime System searches for a file that has the same name as the application and the extension **.w32** in the Windows directory (typically **windows**). If an application specific file is not found, then the default **w32app.w32** file is used instead. (Any chained or linked programs will always use the same initialization file that the *main* program was using.)

The initialization file **w32app.w32** is an editable, text file that contains records that set all the runtime environment variables to null or empty strings so that the default values will be used. Edit the file (or a copy) to set any of the variables to a specific value. Its contents will not be replaced when you install a software upgrade.

To list the available environment variables, execute OBWIN without any parameters. They have been grouped here by function for your convenience.

W/32 Runtime Variable

B_BRTMDIR

Set to the path for the W/32 Runtime System. By default, this program is installed in the **windows\system** or **winnt\system32** directory. If you have moved the Brtm program, you must set this variable to its path.

File and I/O-Related Environment Variables

Note that whenever CET BASIC expects to find a file name, any of the following formats will be considered valid:

1. File names may be in a DOS format. The complete path name may be specified as in **\pos\cp\data\custname** or relative to the current directory as in **.\custname**.

2. The CET W/32 product recognizes the slash character ('/'), and automatically converts any slashes in the name to back slashes ('\') to provide compatibility with the CET UNIX-based products.

   Note that anytime a file name begins with a dot character or a '/', '\' or a drive code (c:), CET BASIC assumes that you are using a file name *native* to the operating system and no *filetype* translation will be done. (See #3.)

3. File names may also be specified in a THEOS format with periods separating each part of the name. By default, BASIC will use the following *filetype* method to *translate* the name of the file. (See also B_LIBFNTYP.)

   THEOS File Name DOS File Name

```
CUST              CUST
CUST.DATA         DATA\CUST
CP.DATA.CUST      CP\DATA\CUST
```

Unless the complete path name is specified, BASIC will search for the file in the current directory, and then along the path specified with the B_FPATH variable. If the file is still not found, a trappable error 30 is detected. (The B_FPATH setting will be searched to find all files excepts for those opened for OUTPUT SEQUENTIAL. CET BASIC always expects to find these files in the current working directory.)

B_CSHFCLOSE

Set to 1 to close all files before performing a CSH/CSI statement. This variable will override the default which is to not close any open file.

B_FPATH

Set to the list of search paths for file OPEN statements. Separate multiple path names with a semicolon. For example, if the file **inquiry** is opened, BASIC will first search the current directory. When the file is not found and the variable is set to the following, the program will attempt to open **d:\apf\inquiry**.

```
B_FPATH=D:\APF
```

Note that the B_FPATH variable will be searched to find all files except for those opened for OUTPUT SEQUENTIAL. In that case, the runtime system will always output the file in the current directory.

B_LIBFNTYP

Set this value to 1 so that a filename in the form **a.b.c** will be opened as **.\b\a\c** (in the current directory). If this variable is not set, then the BASIC system will attempt to open the file in the form **.\a\b\c**, by default.

B_LKPATH

Set to the list of paths to search for executables used in LINK, CHAIN, and RUN statements. Separate multiple path names with a semicolon.

Note that W/32 behaves like Windows when searching for an executable. The search sequence is as follows:

```
the directory where the calling program was executed
the current working directory
the \windows or \windows\system directory
the directories set in the PATH variable
```

the directories set in the B_LKPATH variable

B_OPTLOCK

Set to 1 so that the file pointer is not moved after executing a RESUME from within a routine designed to handle a (record) lock condition. The READNEXT and READPREV statements will attempt to read the desired record again.

If this variable is not set when a lock condition is encountered, BASIC will move on to the next or previous record after a RESUME statement.

Since this behavior is typical in the THEOS operating system, the variable will automatically be set with B_EMULATE=THEOS.

B_REOPENS

Set to the number of retries for OPEN statements that access an indexed file. The default is one try.

Variables Which Affect Printer Use and Control

For complete information on printing under Windows see the *Implementation Notes* in the Appendix.

B_FFPTR

Set to 1 to have a form feed issued upon closing the printer. The default is to not generate a form feed.

B_MAPPRT*n*

Set to the name of the file that contains the control sequences for PRINTER*n*. The following example indicates that the escape sequences for the CRT functions to use for PRINTER1 can be found in the file **hplaser.lpr:**

B_MAPPRT1=HPLASER.LPR

For information on using this variable refer to the *Formatted Output* chapter in the *CET BASIC Language Reference Manual.*

B_PRINTER*n*

Set to a specific printer when the Windows printing interface variable (B_WINPRINT) is in effect. A specific printer may be defined by setting:

B_PRINTER*n*=printer name

Under Windows NT and 95, this name must be the "Printer Name" assigned to the printer in Print Manager or the Printers folder, respectively. (This is not the "Driver" or the port which will be used.) Under Windows, you do not have the option to name a printer so use the name of the driver that was installed on the port (e.g. "HP LaserJet III").

Note that this is the only case, where the printer number 1 to 9 in the variable name is associated with the LPT port for that printer so that the B_MAPPRT*n* feature may still be used. If the B_PRINTER*n* variable is set to DEFAULT or DIALOG, there is no association made between the printer number and the LPT port under any Windows platform.

The variable may also be set to display a Print dialog so users may select a printer from the list specified in Print Manager.

B_PRINTER*n*=DIALOG

Since you will be printing ASCII text, there are some options in the Print dialog such as "Copies" and "Page Range" that are disabled. They will be implemented along with Windows (image) printing in a future release.

The default system printer specified in Print Manager is used if B_PRINTER*n* is not set or is set equal to:

B_*PRINTERn*=DEFAULT

Printing is described in detail in the *Implementation Notes* in the Appendix.

B_WINPRINT

Set to 1 to use the printer interface covered under B_PRINTER*n*. If the B_WINPRINT variable is not set, then the.**bat** type printing method will be used. Both methods are described in the *Implementation Notes.*

Variables Which Affect Screen Display

B_DFLTBGC

Set to the value of the default background color that you want used when a program is executed.

During the initialization process, the Windows Framework will check the setting for this variable in the **w32app.w32** file. If it is not set, a black application window will appear before any color used in your program is displayed.

If your application uses a background color, we recommend that you set B_DFLTBGC to the same color to avoid having the screen change. If you do not

currently use color, a Windows look may be achieved simply by setting the environment variables to black on white (gray).

To display the background in gray:

```
SET B_DFLTBGC =7
```

For details on using color, refer to the COLOR statement or the chapter on *Line Drawing and Color Features* in the *CET BASIC Language Reference Manual*.

B_DFLTFGC

Set to the value of the default foreground color to be used for the same reason we recommended setting B_DFLTBGC. For example, the following variable will set the foreground color to black.

```
SET B_DFLTFGC = 0
```

B_SCRNOLINES

Set to the number of rows to be displayed on the screen. The default is 25 unless B_ EMULATE is set. In that case, 24 rows will be displayed.

Variables that Provide Compatibility with THEOS

The following variables may be set so that CET BASIC programs will behave as they do under the THEOS operating system. Setting B_EMULATE=THEOS automatically activates the other variables in this section with the exception of B_?DRIVE, B_CSHFCLOSE and B_SEARCH. To turn off a default behavior, set the specific environment variable to null or zero. For example:

```
B_THLOCK=0
```

B_<*drive-code*>DRIVE

Set to affect the manner in which the THEOS drive designations used in an OPEN statement are interpreted. For example, if you enter

```
B_SDRIVE=C:\TREND
```

The following code may then be used to access **c:\trend\dms\data\dms01**:

```
lib$ = "DMS"
drive$ = "S"
open #1: LIB$&".DATA.DMS01:"&DRIVE$, update direct
```

Setting this variable will override the default which is to search for a file in the current directory, and then along the path specified by B_FPATH.

B_CSHFCLOSE

Set to 1 to close all files before performing a CSH/CSI statement. This variable will override the default which is to not close open files.

B_DBSLIU

Set to 1 to enable the destructive backspace for LINPUT USING. Normally, the backspace character is not destructive.

B_DOSTAB

Set to 1 so that LINPUT USING statements will return an INP value of 9 for the TAB character as it does under THEOS. Otherwise, the default is to ignore the TAB entry.

B_EMULATE

Set to THEOS to emulate the following THEOS behavior:

1. Return THEOS INP values for LINPUT (USING) and INPUT
2. Use THEOS values for CRT$ functions
3. Map to THEOS characters for ASCII values greater than 160 decimal (See also B_THCONV.)
4. Enable default foreground/background for display as per THEOS
5. Change print Esc character from 255 to 27
6. Change PAGE(0),LINE(0) from 25->23 and 80->79 for normal screen
7. CRT("EU") erases foreground values 0,1,4,5,10,11,14,15
8. Return 3 for GET statement on Ctrl+C instead of ERROR 1
9. Return INP=9 for LINPUT statement on TAB, not insert mode
10. Fold unquoted or single quoted CMDARG() values to upper case
11. Initialize CSI.RETURN.CODE to -1 so RC has value for last CSI/CSH
12. Ignore ACTIVATE, SEMAPHORE, PUT COMMON and GET COMMON statements

Please note that divide by zero errors will be detected. This is an illegal or undefined operation for good reason. If allowed, your program could produce incorrect results without any warning message.

B_GINVERT

Set to 1 to invert the meaning of intersection characters when used with the CRT$ function as in (CRT$("UI") and CRT$("DI").

B_OPTLOCK

Set to 1 so that the file pointer is not moved after executing a RESUME from within a routine designed to handle a record lock condition. This way, (MAT) READNEXT and (MAT) READPREV statements will attempt to reread the desired record.

Note that if a lock condition is encountered when this variable is not set, BASIC will move on to the next or previous record after a RESUME statement.

B_SEARCH

Set to the THEOS drive search sequence to use, by default SABCDEFGH...Z.

B_THLOCK

Set to 1 to perform multi-user record locking and suspend program operation when a lock condition is detected while accessing a record in a direct or indexed file. Since this is the typical THEOS behavior, this variable will be automatically set under B_EMULATE.

When a record is locked, the operator must wait until it is unlocked. Since a lock condition is not an error, normal error trapping can not be used. To overcome this problem, set B_THLOCK to null. Then, use an ON LOCK or ON ERROR statement to define the special handling for a lock condition.

For information on handling file/record lock conditions, refer to the *CET BASIC Error Handling System* chapter in the *CET BASIC Language Reference Manual.*

B_THPON

Set to 1 to invert the default meaning of PON/POFF so that a high-intensity foreground is used. CET BASIC normally displays low-intensity text which is typical in the DOS environment.

B_THVAL

Set to 1 so that the VAL and NBR functions work as in THEOS. Refer to the *CET BASIC Language Reference Manual* for a definition of the standard method used by CET to handle these functions.

Miscellaneous Environment Variables

B_4DYEAR

Set to 1 to inform the DATE$ and DTE$ functions that a 4-digit year should be returned. Refer to the variable B_YR2000 if you wish to indicate which year should be the first one to be interpreted as 20XX.

B_DATEFORM

Set to 2 (European), or 3 (International) to override the default which is to display American-formatted dates. The OPTION DATEFORM statement may be used to change the format from within a BASIC program.

B_LANGUAGE

Set to a 2-character language code to force the YESNO$ function to generate the proper text for French (FR), Dutch (DU), German (GE), Swedish (SW), Finnish (FI), Spanish (SP), Portuguese (PO), or Italian (IT). The default is English (EN).

B_USER

Set to 1, 2, 3... in each **w32app.w32** file to specify a unique user on a network. This feature is convenient when a program must create temporary work files in a multi-user environment. The following code segment illustrates how to create an indexed file with a unique name for each user. The newly created file will be stored in the **\tmp** directory on drive Z on the remote server.

```
call Bgetenv(addrof(user$),"B_USER")
filename$ = "Z:\TMP\WORK"&user$
CALL Berase(filename$)
CALL Bcreate(filename$&" INDEXED RECL 100 KEYLEN 10")
OPEN #1: filename$, update indexed
```

This procedure will work equally as well with a sequential file. In that case, instead of using the Bcreate function, use the OPEN statement with the option OUTPUT to write the file, or with the EXTEND option to append the new data.

A special function called cLogin may also be used to get the user name. This feature is covered in the *W/32 Window Functions* chapter.

B_ODBCERR

Set to 1 to display messages for errors that may occur while accessing a file using the W/32 ODBC facility.

B_YR2000

Set to the 2-digit year which is to be the last year interpreted as 20XX. This feature allows you to have dates in the 1900's and 2000's without using the complete 4-digit year. For example, suppose you have no dates prior to January 1st, 1925. In that case, you could set

```
B_YR200O=24
```

Setting the above variable to the last year to be interpreted as 20XX would make the statements DAY("1/1/24") and DAY("1/1/2024") identical.

Using our example, the following dates will be interpreted as:

01/01/00	01/01/2000
12/31/24	12/31/2024
01/01/25	01/01/1925
12/31/99	12/31/1999

Variables Used For Debugging Output

The following environment variables are normally only used to generate low-level debugging information when reporting problems to your CET distributor. The one exception is B_ENVRPT which may be used to display a list of the variable settings used by the current program.

B_BASRPT

Set to 1 to obtain debugging information within the initial W/32 routine.

B_CSKRPT

Set to 1 to obtain debugging information on call stack reporting in Brtm.

B_ENDRPT

Set to 1 to obtain debugging information on program termination.

B_ENVRPT

Set to 1 to obtain a listing of the environment variable settings used by the current program.

B_IOPRPT

Set to 1 to obtain debugging information on I/O operations.

B_IPCRPT

Set to 1 to obtain debugging information on interprocess communications.

B_ISRRPT

Set to 1 to obtain debugging information on ISAM reads.

B_LCKRPT

Set to 1 to obtain information on the ISAM locks used during program execution.

B_LNKRPT

Set to 1 to obtain information on the execution of LINK, CHAIN or RUN statements.

B_OPNRPT

Set to 1 to obtain debugging information on ISAM file opens.

B_RTMRPT

Set to 1 to obtain debugging information on transactions performed by Brtm.

B_WRERRFIL

Set to 1 to write all file activity into the **\tmp\brtmdbg.log** file. Note that all errors will be logged, even those trapped by ON ERROR statements.

CHAPTER 4

The Windows Framework

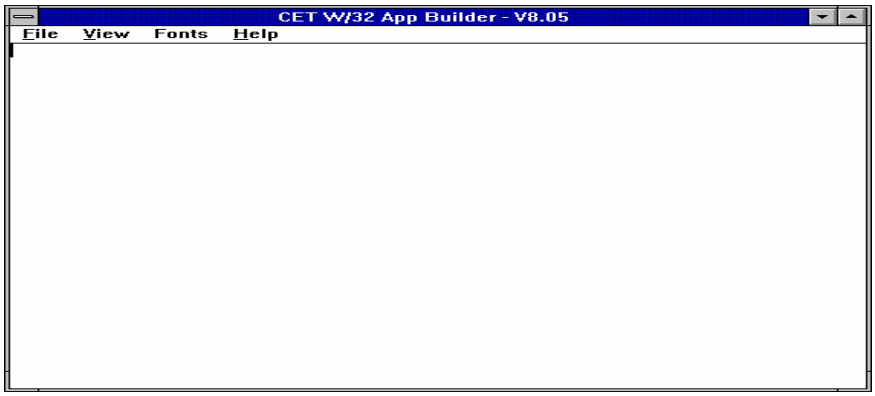
Introduction

When you create a program using the W/32 Application Builder, a set of editable files is read to determine how to open and manage the window used to display your application. These files are referred to as the Windows Framework.

This chapter covers the features in the Windows Framework that may be used throughout your application without having to modify your source code. If you have not already done so, we recommend that you compile a simple program that prints some text so you can use each of the features as they are discussed.

The Default Window

By default, your W/32 programs will have a main window with File, View, Fonts and Help menus that may be selected with or without a mouse.



The File Menu

The default File menu has only one command, Exit. Selecting Exit during normal program execution is similar to entering the system interrupt key (Ctrl+C). Unless an ON ERROR or ON INTERRUPT routine is available to process the interrupt, a message box will pop-up and display “Program Interrupted”. Otherwise, the message "Application Exit” will be displayed when you exit. In both cases, you will have to press OK to continue.

Message boxes are commonly used in Windows applications and will be covered in detail in the next chapter. It is important to note here, that these specific message boxes have been added because Windows programs typically close the window upon exit so you may not have a chance to see the last things that were written to the screen. The cForceExit function described in the *W/32 Window Functions* chapter may be used if you wish to suppress this behavior.

The View Menu

The View menu has two commands, Toolbar and Status Bar. The toolbar is a series of buttons displayed underneath the menu. These buttons are shortcuts to menu commands. By default, there are two buttons, one for the Font-Select command, and one for Help-About. For information on designing your own toolbars, refer to the cToolBar function in the *W/32 Window Functions* chapter.

The status bar is an area at the bottom of the screen that may be used to display status messages as well as help messages regarding the menu and toolbar commands.

Both the Toolbar and Status Bar are toggle commands. Select the command once to enable the feature and again to disable it.

The default is to not display either the toolbar or status bar when a W/32 program is executed. The Windows Framework may be modified if you want one or both of them to appear automatically. This process will be covered later in this chapter. (There are also W/32 Window functions that are available to enable/disable these features from within a BASIC program.)

The Fonts Menu

The Fonts-Select menu command allows you to change the type and size of the font that is being used to display the output directed to the application window. The default font is Courier 9-point, where point refers to the size of the individual characters.

The size of font which may be used depends upon the current screen resolution. If you are running with a standard VGA resolution of 640x480, then 9-point is the largest font you can display. If you are running in a high resolution mode (e.g. 1024x768), you can use a font larger than 9-point. (The cDisplayInfo function may be used to determine the display resolution at runtime.)

Although you can resize the window, the display area will never be larger than what is required to display 80x25 characters at the current font size. If you make the window smaller, scroll bars will appear so you can view the portion of the screen that is outside the displayed area.

You may want to prevent the operator from using the menu to select different fonts and control the selection at the application level. (This procedure is covered later in this chapter.)

It is important to note that the above information applies only to character-based screens created with PRINT statements. Currently, you cannot change the font in a form created with the W/32 Dialog Editor.

The Help Menu

The Help-About command displays a message box with version and copyright information about the W/32 Application Builder, by default. An example of how this text may be modified to provide information about your program will be given later in this chapter.

The Default Windows Framework

Every W/32 program uses the default features and initial conditions that have been covered so far because OBWIN links in the **cetwin.obj** and **cetwin.res** files during the compilation process. By default, these files are stored in the **\cetlib** directory during installation.

The **cetwin.res** file is referred to as the *resource* file. It contains the code for the default menus and text items which are displayed in your compiled programs. The source code for the resource file that may be edited is stored in **cetuser.rc.**

The source code for the other Windows Framework file called **cetwin.obj** is stored in **cetwin.b**. This file contains a number of REMed lines that are easily modified whenever you wish to change the default behavior and display the status bar and toolbar, select a specific display font and/or add other menu commands in your application programs.

If you list **cetwin.b**, you will see that all the operations are performed in a series of BASIC subroutines. Each subroutine begins with a $SUB statement which defines the name of the subroutine. The name of the first subroutine is cetPreInit since it is called before the first line in a W/32 program is executed.

A $EXIT statement defines the end of the subroutine and acts like a RETURN. $GLOBAL is available so that you may specify the variables that are to be shared with another program module. (In **cetwin.b**, CmdID% is defined as a global variable.)

After a BASIC subroutine is compiled it may be linked into your *main* program to produce a single executable. This ability to call BASIC subroutines is such a useful feature that it is covered in a separate chapter in this manual.

Altering the Default Framework

The Windows Framework is designed so that you can quickly implement the Windows features you want to use in your application.

To alter the default behavior, first copy the files that may be modified into your local or current working directory with a different name. For example, if your W/32 files are stored in **\cetlib** and your local directory is **\tst**:

```
copy \cetlib\cetwin.b \tst\mywin.b
copy \cetlib\cetuser.rc \tst\myuser.rc
```

The file names in the local directory must be different than the ones in **\cetlib.** Later, you will probably decide to have several copies of these files in order to vary the menu bar and toolbar (and other behaviors) used in specific programs.

After making any of the modifications covered in this chapter, use OBWIN to recompile the resource files. For example:

```
obwin -r myuser                    Create myuser.res
obwin -c mywin.b                   Create mywin.obj
```

The OBWIN command used to compile your application programs will indicate which framework file should be linked into the executable. The -wo flag must be used to override the default file **cetwin.obj**. Use the -wr flag to override the default **cetwin.res** file. For example:

```
obwin -o tst tst.b                          Use cetwin.res and cetwin.obj
obwin -o tst -wr myuser tst.b          Use myuser.res and cetwin.obj
obwin -o tst -wo mywin tst.b               Use cetwin.res and mywin.obj
obwin -o tst -wo mywin -wr myuser tst.b    Use myuser.res and mywin.obj
```

In the following sections, we will discuss how to remove the default Font menu and select the font to be used. You will also find information on using a customized toolbar and menu bar without changing your source programs.

Selecting Fonts

This section will explain how the cSelectFont routine may be used to select a font from within your (text-based) application instead of from the Fonts menu.

Since the fonts that are available will vary depending upon the ANSI fixed fonts that are on the system, it is not possible to document all the parameters for every font. (W/32 must use fixed-width fonts since the program must be able to determine where a character will appear by multiplying the character position times the width of the character.)

If you plan to use fonts other than the default Courier 9 or Terminal, compile a program like the one below. The call to the cFontValues routine will return the necessary parameters.

```
PRINT "ANY TEXT YOU WOULD LIKE TO SEE"
PRINT "AND AS MANY LINES AS YOU WOULD LIKE"
LINPUT A$
CALL cFontValues
END
```

Execute the program. Then, choose the Fonts-Select command when the program is waiting for input. The font dialog box will display the available fonts, styles and sizes. Select a font, and press Enter to get by the LINPUT statement.

The call to cFontValues will display all of the parameters that you need to pass to cSelectFont whenever you want to change the font from within a BASIC program. (Refer to the *W/32 Window Functions* chapter for more information on using these routines.)

Removing the Fonts Menu

If you plan on using the same Windows feature in all or a portion of your application, we recommend that you use the following procedure to modify a copy of the resource file instead of each one of your BASIC programs.

For example, to remove the Fonts menu:

1. Copy the source code for the resource file into your current working directory with a different name. For example, if your W/32 files are stored in

\cetlib and your local directory is **\tst**:

```
copy \cetlib\cetuser.rc \tst\myuser.rc
```

2. Edit the code in the **myuser.rc** file and remove the lines that display the Fonts menu:

```
POPUP "Fonts"
BEGIN
    MENUITEM "Select",        ID_FONTS_SELECT
END
```

3. Compile a new resource file named **myuser.res** using the modifications made in **myuser.rc**:

```
obwin -r myuser
```

4. Rebuild your application using this resource file (and not the default **cetwin.res**) by specifying its name along with the -wr flag as in:

```
obwin -wr myuser -o tst tst.b
```

The Fonts menu will not appear in any BASIC program that you compile using the modified **myuser.res** file.

Selecting Fonts From Within the Windows Framework

The **cetwin.obj** file contains the cetPreInit subroutine that is called when the Windows Framework is initialized. All of the statements in this subroutine will be performed before the first line of code in your program is executed.

If you plan on using one particular font in your application, we suggest that you modify this file to select the font and leave your existing program files as is. To modify the **cetwin.obj** file:

1. Copy the source code **cetwin.b** into the working directory with a different name.

```
copy \cetlib\cetwin.b \tst\mywin.b
```

2. Modify the copy of **cetwin.b** to use the Terminal 9-point font, by default. This is done by unremarking the following statement in the cetPreInit subroutine at the top of the file:

```
REM call cSelectFont("Terminal",-12,400,0,255,49)
```

If you want to use another font, enter the appropriate parameters determined

with a call to cFontValues. (Refer to the procedure given earlier in this section.)

3. Compile the modified program:

```
obwin -c mywin.b
```

The specified font will be used automatically in any program you compile in the **\tst** directory that uses the following command to link to the new **.obj** file.

```
obwin -wo mywin -o tst tst.b
```

Displaying the Default Toolbar

The Windows Framework may also be modified to automatically display a toolbar. The default toolbar consists of two buttons. The first one may be used as a shortcut key to the Fonts-Select command. The second button (the ?) is a shortcut for the Help-About command.

To display the toolbar, modify the cetPreInit subroutine in your copy of the **cetwin.b** file by unremarking the following statement (and recompiling):

```
REM call cShowToolBar(1)
```

Calling this routine will automatically *unhide* the toolbar when you execute any program compiled with the modified **cetwin.b** file. If your program (or the **cetwin.b** file) uses a new toolbar defined with the cToolBar routine (see the *W/32 Window Functions* chapter), it will be displayed instead. Otherwise, the default toolbar will be used.

Displaying the Default Status Bar

The status bar is a line at the bottom of the application window that displays, by default, the status of the current program ("running" and "waiting for input") along with prompting messages for the menu commands. The cetPreInit subroutine in the **cetwin.b** file contains the code to enable the status bar. To implement this feature, unremark the following line (and recompile):

```
REM call cShowStatusBar(1)
```

A special cStatusText function may be called from within your application program if you wish to replace the default messages with your own text. Refer to the *W/32 Window Functions* chapter for details on using cStatusText.

Modifying the Default Title in the Main Window

By default, a title will not be displayed in the main window. (We can not get rid of the dash!) To define your own title, simply call cAppTitle:

```
call cAppTitle("ABC Payroll System")
```

Take care that the text in the title does not exceed the width of the application window. Any excess characters will be truncated.

The cAppTitle function may be used from within any BASIC program. If it is used in the cetPreInit subroutine in the **cetwin.b** file, then any W/32 program that is linked with that **.obj** file will have the same title.

Modifying the Default Menu Bar

The W/32 Application Builder gives you many options on how to incorporate Windows features into existing programs. For example, you can continue to display your traditional menus, completely replace your existing menus or display your existing menus in addition to giving the operator the option of using a drop-down menu with or without a mouse.

The Windows Framework has been designed to assist you in implementing a Windows drop-down menu. By default, your W/32 programs will display File, View, Fonts and Help menus. This menu bar may be customized to display up to 50 different menus with a maximum of 250 menu commands.

For example, suppose you want to execute the Windows Notepad from the menu. First, you must create a copy of the **cetwin.b** and **cetuser.rc** files using the procedure described earlier for selecting a font.

Then, modify the copy of the **cetuser.rc** file by adding the new item "Notepad" to the menu code:

```
POPUP "&Notepad"
BEGIN
    MENUITEM "&Run",        ID_WINSUB1
END
```

Selecting the Notepad menu will display the Run command. Adding the ampersand before the "N" in Notepad and the "R" in Run will underline the characters and indicate that they are the shortcut keys which may be entered to execute the items.

Whenever you add a menu item, you must assign it a unique identifier such as the ID_WINSUB1 in the example above. Since you may have up to 250 different menu commands, ID_WINSUB1 through ID_WINSUB250 are valid.

To process the new menu command, modify the SELECT-CASE statement in the cetWINSUB subroutine located at the bottom of the **cetwin.b** file. Initially, the CASE statement is remarked out. To execute the Windows Notepad in a maximized window, enter:

```
CASE 1
    CALL cWinExec("notepad.exe",3)
```

The cWinExec routine is specially designed for executing Windows programs and is explained in the *W/32 Window Functions* chapter.

When Run is selected from the menu, the framework calls the cetWINSUB subroutine to process the command. The menu identifier "1" in ID_WINSUB1 will cause the statement following CASE 1 to be executed.

Note that the **cetwin.b** file contains a $GLOBAL statement which will pass the menu identifier (1 through 250) as the variable CmdID% to your program. If your program needs to use this information, refer to the *BASIC Subroutines* chapter for instructions on how to use $GLOBAL variables.

After making the modifications, you will need to recompile the files with OBWIN. For example:

```
obwin -c cetwin.b
obwin -r myuser
```

Notepad will now appear on the menu bar in any program you compile using these framework files.

Up to 50 different menu bars may be defined using the keywords:

MENU BEGIN END POPUP MENUITEM

The MENU keyword is used to identify a unique menu bar. The syntax is:

identifier MENU PRELOAD DISCARDABLE

The *identifier* is a unique value that refers to a specific menu bar. For example, the identifier for the default menu is IDR_MAINFRAME. Although you can modify this menu, every application must have one. If you want to add other menu bars, use the identifiers IDR_MENU2 through IDR_MENU50.

The BEGIN and END keywords are used to specify the beginning and end of a *group*. In the case of the MENU keyword, all items contained within the group are associated with that particular menu bar.

The POPUP keyword identifies an individual menu (on the menu bar). In this case, the BEGIN-END group specifies the menu commands to be displayed.

An example of a top-level pop-up is the default File menu. A top-level menu may also contain embedded popup menu groups that are identified by the → that appears at the end of the menu text. These separate menus are referred to as cascading menus since they are displayed to the right of the menu item.

The syntax for the POPUP keyword is:

```
POPUP "title"
```

The title is the name of the specific menu. The quotes are required. Within the title you can designate one character as the shortcut key (to be underlined) that may be entered to select the item. Precede the desired character with an ampersand. For example, the following code displays the default File menu with the "F" underlined to indicate the shortcut key.

```
POPUP "&File"
```

The MENUITEM keyword is used to define a menu item or command similar to the way POPUP defines a menu group. Associated with each menu item is a unique identifier which will be passed to the cetWINSUB subroutine when the item is selected (as in the Notepad example). The syntax for the keyword is:

```
MENUITEM "title",  identifier
```

The title is the text to be displayed on the menu. A specific shortcut key may also be associated with this title. The identifier is a unique value from ID_WINSUB1 through ID_WINSUB250.

There is a second form of the MENUITEM keyword which may be used to display a horizontal black line if you wish to group the items on a menu. That syntax is:

```
MENUITEM SEPARATOR
```

The SEPARATOR menu item has been used in the default File menu to draw a line before displaying the Exit item.

The following example defines a menu that has three top-level popups: File, Find Customer and Add Customer. This menu will appear by default since it uses the IDR_MAINFRAME identifier. Note that ID_APP_EXIT is a special menu identifier that will exit the application.

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM SEPARATOR
        MENUITEM "E&xit",              ID_APP_EXIT
    END
```

```
    POPUP "&Find Customer"
    BEGIN
        MENUITEM "By &Name",        ID_WINSUB1
        MENUITEM "By &Phone",       ID_WINSUB2
    END

    POPUP "&Add Customer"
    BEGIN
        MENUITEM "&New",            ID_WINSUB3
    END

END
```

W/32 gives you complete control over how your program responds to a Windows event such as a menu selection. For example, at a particular point you may disable a command with a call to cEnableItem so that it cannot be selected. (The text will appear in gray.) To re-enable the item, call cEnableItem again.

During the execution of a program, there may be times when you want to display an entirely different menu. This is done with the cNewMenu function explained in the *W/32 Window Functions* chapter.

Adding Prompt Messages for Menu Items

A special feature in the resource file allows you to add a short prompting message to describe the individual menu items. These messages will appear in the status bar when the item is highlighted.

Menu prompts are defined in a separate section of the **cetuser.rc** file called the STRINGTABLE. Here, you will find the comment "Add ID_WINSUB prompts here...". (A comment is a line that begins with two forward slashes.) The syntax of the prompt line is:

```
identifier      "string\ntooltip"
```

The *identifier* is the unique ID value associated with the ID_WINSUB*n* menu item. The *string* is the text to be displayed in the status bar. The *tooltip* is a short string of one or two words that will appear in a small window when the mouse cursor is resting over the toolbar button associated with the menu item. Note that the string and tooltip are separated with the newline character (\n).

The following defines a prompt for the Add Customer-New menu command from the example in the previous section.

```
ID_WINSUB3      "Add a new customer to the data base.\nAdd New"
```

When the menu item is highlighted, the text “Add a new customer to the database” will appear in the status bar (when it is visible). If a toolbar button has been defined for the menu item, the tooltip “Add New” will appear in a window when the mouse cursor is moved to the button.

Modifying the Help-About Message

The Help-About menu displays the application name, version and copyright information in a message box. By default, this information pertains to the W/32 Application Builder. Use the following steps if you wish to display specific information about your application.

1. Refer to the Menu section in the **cetuser.rc** file and note that the POPUP “&Help” menu displays the MENUITEM “&About W/32 App Builder...”. Change the text so that the name of your application will appear instead.

2. The Dialog section contains the caption or title for the help window and the lines of text that are displayed in the dialog box called IDD_ABOUTBOX DIALOG. The lines that may be modified are:

```
CAPTION “About W/32 App Builder”

LTEXT   “\nCET W/32 App Builder Version 8.13\n\nCopyright \251
        CET Software Inc. 1986-95.\nAll rights reserved.
```

Be careful when modifying this text. Any characters that will not fit in the window will be truncated. The newline character “\n” may be used to position text inside the dialog box. The “\251” is a special character that may also be used to display the copyright symbol.

Modifying the Title for a W/32 Message Box

W/32 programs will automatically display certain system information in a message box such as the one that appears when you exit the program. By default, the title for the message box is the name of the program. This text may be changed to something more meaningful to the operator in the STRINGTABLE section of the **cetuser.rc** file. The default lines are as follows:

```
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    AFX_IDS_APP_TITLE           “”
END
```

The null string may be changed to any text string that does not exceed the width of the window as any excess characters will be truncated. The newline character “\n” may be used to position text inside the dialog box.

Changing the Program Icon

By default, all W/32 programs will display the CET icon in the Help-About dialog and on the desktop when a program is minimized. The icon will also be used if you create your own program item in the Program Manager.

The CET icon is defined in the ICON section of the **cetuser.rc** file as:

```
IDR_MAINFRAME     ICON DISCARDABLE     "CETICON.ICO"
```

The default icon is stored in the **ceticon.ico** file in the **\cetlib\inc** directory if you used the default path during installation.

To design your own icon, copy the default icon into your local directory with a new name. For example:

```
copy \cetlib\inc\ceticon.ico \tst\myicon.ico
```

Now, use the IMAGEDIT program to modify the new file **myicon.ico**. When you are done, edit the line in **cetuser.rc** to use your new icon.

```
IDR_MAINFRAME     ICON DISCARDABLE     "MYICON.ICO"
```

Emulating Keyboard Input with Mouse Subroutines

Text-based applications under DOS or UNIX normally provide some menu options and wait for the user to indicate what to do next. Event-driven programs under Windows do essentially the same thing.

As a developer, you have complete control over how your program responds to Windows events such as a menu selection or a mouse click. For example, if a user clicks on a menu selection, your program will be notified. At that time, you may choose to process the selection, ignore the event or save the information for consideration later.

The Windows Framework is designed to notify an application when specific mouse events occur. The **cetwin.b** file contains BASIC subroutines that may be used to detect any of the following mouse events

- cetRBtnDn and cetLBtnDn — Right and left button down
- cetRBtnUp and cetLBtnUp — Right and left button up
- cetRBtnDClk and cetLBtnDClk — Right and left button double click
- cetMouseMove — Mouse movement

In general, only the routines that detect a right or left mouse button Double Click or Up will be used as it is the release of a mouse button that signifies an event. For example, you can *drag* the mouse through the items on a menu without

making a choice. It is when you release the mouse button (the button Up event) that the item is selected.

The button down and move routines are provided in case you require a detailed level of notification about the mouse. A call to cGetMouse may be used to return the character-based screen coordinates of the current mouse position.

The following examples use W/32 window functions to insert characters into the keyboard input queue. If your program is waiting for input (e.g. with a LINPUT statement) when the button is released, the statement will execute as if the characters were entered at the keyboard.

The first example places an F1 key into the buffer. Note the use of the two character code. The first code of zero says that an extended key (the F1 key) is coming next.

```
$SUB cetLBtnUp
REM Called when the left mouse button is released (UP)
REM Insert an F1 key into the input queue
call cInputQ(0)
call cInputQ(59)
$EXIT
```

The next example for cetRBtnUp will place an ENTER key into the input buffer as soon as the right button is released.

```
$SUB cetRBtnUp
REM  Called when the right mouse button is released to insert an ENTER key
REM  into the input queue
call cInputQ(13)
$EXIT
```

A table of input character codes may be found in the *Appendix* of the *CET BASIC Language Reference Manual.*

The following is another example of how a copy of the **cetwin.b** file may be modified to add mouse support to an application.

```
$SUB cetLBtnDn
$GLOBAL MROW%, MCOL%, MOUSEDOWN%, SELANCHOR%
REM        Called when the Left Mouse Button goes DOWN
MOUSEDOWN=-1
CALL cGetMouse(addrof(mrow%),addrof(mcol%))
CALL cInputQ(3007) \ rem MOUSE.DOWN%
$EXIT

$SUB cetRBtnDn
```

```
REM       Called when the Right Mouse Button goes DOWN
REM       Unused in this example
$EXIT

$SUB cetLBtnUp
$GLOBAL MROW%, MCOL%, MOUSEDOWN%, SELANCHOR%
REM       Called when the Left Mouse Button goes UP
MOUSEDOWN=0
CALL cGetMouse(addrof(mrow%),addrof(mcol%))
CALL cInputQ(3008 \ rem MOUSE.UP%
$EXIT

$SUB cetRBtnUp
REM       Called when the Right Mouse Button goes UP
REM       Unused in this example
$EXIT

$SUB cetLBtnDClk
$GLOBAL MROW%, MCOL%, MOUSEDOWN%, SELANCHOR%
REM       Called when the Left Mouse Button is double-clicked
MOUSEDOWN=0
CALL cGetMouse(addrof(mrow%),addrof(mcol%))
CALL cInputQ(3005 \ rem MOUSE.DBLCLICK%
$EXIT

$SUB cetRBtnDClk
REM       Called when the Right Mouse Button is double-clicked
REM       Unused in this example
$EXIT

$SUB cetMouseMove
$GLOBAL MROW%, MCOL%, MOUSEDOWN%, SELANCHOR%
REM       Called when the Mouse Moves
if mrow%>0 and mcol%>0 and selanchor%>0 and mousedown%<>0
      CALL cGetMouse(addrof(mrow%),addrof(mcol%))
      if row%<> mrow% or col% <> mcol%
            mrow%=row% \ mcol%=col%
            call cInputQ(3006) \ rem MOUSE.MOVE%
            ifend
      ifend
$EXIT
```

In order to pass the values of the $GLOBAL variables to your BASIC program, the following statements would have to be entered as the first lines of the program:

```
$MAIN
```

```
$GLOBAL MROW%, MCOL%, MOUSEDOWN%, SELANCHOR%
```

The variables MROW%, MCOL% and SELANCHOR% are used to determine the position where the mouse event was detected and mark it (using SELANCHOR%) so the cursor can be repositioned after the desired operation is performed. (The ROW% and COL% variables indicate the starting location of the input field.)

The $MAIN program will also need to define the mouse events. Large numbers are used to make it easy to distinguish between keyboard and mouse entries.

```
MOUSE.DBLCLICK%=3005
MOUSE.MOVE%=3006
MOUSE.DOWN%=3007
MOUSE.UP%=3008
```

Mouse support is automatically built into forms and dialog boxes created with the W/32 Dialog Editor. Refer to that chapter for specific information on how these mouse events are handled.

Special Considerations when using Windows

There are some important considerations to be aware of when using the Windows features.

Remember that menu selection and mouse events may occur at any time during execution except when the program is first initialized (by the Windows Framework). If you have customers who are used to typing ahead, they must not select a menu command until the screen is completely displayed. Otherwise, an event may be activated before the program is ready to process it.

When a user chooses an item on the menu, the Windows Framework immediately calls a subroutine in **cetwin.obj**. This is an asynchronous event similar to a hardware interrupt that may occur while executing a line of code. An improper calculation may result if the program is accessing a global variable that was suddenly changed by the event.

We suggest that you use the following guidelines to prevent these types of problems from occurring.

1. Temporarily disable any menu items that may interfere with the normal data entry process. (All menu items may need to be disabled until the screen is displayed to prevent type-ahead problems.)
2. Set variables that may be checked to determine when $GLOBAL variables shared between the **cetwin.obj** file and your program have been changed. If

a change is detected, then you can copy the shared variable into a local variable that only your program can access.

3. Only execute code from the **cetwin.obj** file that will set information about the program *state*. The application program should process the information.

4. Never write any information to a file or the screen from **cetwin.obj**. Otherwise, your program could be redrawing the screen when a menu item is selected. If you also update the screen from the **cetwin.obj** file, your output could become mixed together.

   For example, the following code in the cetMouseMove routine determines the current mouse position, sets the cursor to that position and then outputs the string "Hello World". This could have strange side-effects if it is executed at the same time your main program is printing to the screen.

```
call cGetMouse(addrof(row%),addrof(column%))
call cSetCursor(row%,column%)
print "Hello World";
```

A Sample Text-based Program

A special demo program has been documented in the *Appendix* to show how a text-based application can have the *look* of a Windows program. Many of the W/32 features that are covered may be implemented by simply modifying the Windows Framework files.

CHAPTER 5

W/32 Integrated Development Environment

Introduction

The W/32 Application Builder provides you with an integrated, graphical development environment (**w32app.exe**) so that you have all the functionality you need right at your fingertips. Without leaving the program, you can create

- √ Projects to build, test and maintain your executables.
- √ Source files using a special text editor designed for writing large programs. The W/32 Debugger is available to assist you in finding any errors that may occur during program execution.
- √ GUI input forms and dialogs to give your application the look and feel found in other Windows products.

The information in this chapter is organized into the following sections to help you learn how to create projects for your new and existing program executables.

- √ Using projects: the basic concepts.
- √ Creating projects.
- √ Adding files to projects.
- √ Building and maintaining projects.

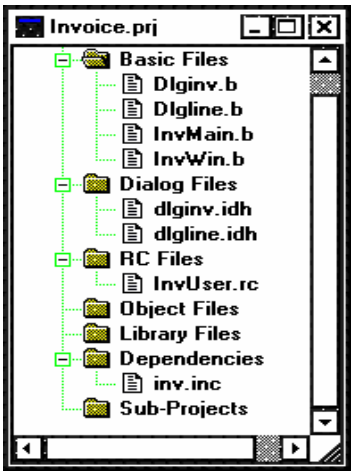
Please refer to the next chapters for details on the W/32 Debugger and the W/32 Dialog Editor.

Windows for Workgroups Note: Because of problems in the Win32s product, W32APP is not available for use under Windows for Workgroups. We recommend that you upgrade your development system to either Windows 95 or Windows NT so that you can take advantage of these features.

Using Projects -- The Basic Concept

W32APP allows you to build projects to simplify the generation, testing and maintenance of new program executables. You may also wish to take advantage of these features and build projects for any existing programs you may have created from the command line.

As you create a new project and add the required files, W32APP displays a project *tree* like the one displayed to the right. Folder icons are used to represent the various types of files (e.g. Basic, Dialog, Object files, etc.) so that it is easy to see which files are used by the project.



Various menu options are available so that you can compile one or more of the files or build and execute the project. W32APP will keep track of the names, locations and dates of each of the files. Each time you execute a project, the program will ask if you want to perform a rebuild if any of the files have been changed.

The basic concept behind building projects is that there is a natural dependency between files. For example, when you create a project to generate **mainprog.exe**, the executable MAINPROG is considered the target file which is dependent upon the source in **mainprog.b**. If the source file contains the statement #include "subtot.inc", MAINPROG is also dependent upon the **subtot.inc** file. Both **mainprog.b** and **subtot.inc** are considered dependencies. Any time one of them is modified, the project has to be rebuilt to regenerate the executable.

There is also a time relationship between an executable and its dependencies. W32APP considers an executable up to date when its date (of last modification) is newer than the date of the source file(s) upon which it is dependent.

W32APP uses the time and dependency relationship between files to determine when a recompilation is required. This means that, once you create a project, you do not have to keep track of the source files you have modified. When you run the executable, W32APP will detect any change and ask if you want to rebuild. If you rebuild the project, only the files that have changed since the last build (i.e. that have dates older than the executable) will be recompiled.

The W32APP Program

The W/32 integrated development environment may be invoked by running the program **w32app.exe** from the Cetbin folder or by selecting the item on the CET W/32 Application Builder group menu from the Windows Programs menu.

After the W32APP program is loaded, the typical Windows File, View and Help menus will be displayed. Although there are a number of toolbar buttons, most of them are initially disabled.

A variety of other menu commands will become available as you use the program to create projects, dialogs, source files or use the W/32 Debugger. Only the menu commands that pertain to projects will be covered in this chapter.

The Help Menu

The Contents command allows you to display the W/32 Help System. The help file corresponds to the W/32 Users Manual. As new features are released, the manual and help file will be updated as quickly as possible. Please refer to the release notes for any information that may not be included.

The Help-About command displays the W32APP version and copyright information.

The View Menu

The View menu has options to show or hide any of the toolbars or the status bar. The majority of the toolbar buttons are designed for use with the W/32 Dialog Editor. If you change the visibility or location of any of the toolbars, the program will be in the same state the next time it is invoked. The View-Reset Defaults command may be used to make the status bar and all the toolbars visible and docked at their default locations.

The View-Settings command may be used to specify global properties that you want to use for all projects. (This feature will be covered later in this chapter.)

The File Menu

The File menu initially contains the commands: New, Open, Find in Files, Import Dialog Template and Exit. It also contains the ten most recently used files that can be re-opened by clicking on them.

The File-New command displays a dialog box with three choices: source, dialog and project. (Creating source and dialog files are covered in separate chapters.)

The File-Open command allows you to select a specific folder and open an existing file with the following file types:

1. Source files with extensions of **.b**, **.h**, **.inc** or **.cpu**, which is used for files originally created with CONTROL Plus under the THEOS operating system. A special text editor is provided for creating these source files. See the next chapter on the W/32 Debugger for details.
2. Dialog files with extensions of **.idh**. Note that if you open a **.b** file, the program will load the source code for the dialog into the W/32 Editor.
3. Project files with extensions of .**prj**.
4. All files with any extension. The program will attempt to load files with extensions different than those listed above into the text editor.

Since the Find in Files command is provided for use when creating and debugging applications, this feature will be covered in the W/32 Debugger chapter.

The File-Import Dialog Template command is a special feature that helps you convert a text-based screen image into a dialog. This feature is covered in detail in the chapter on the W/32 Dialog Editor.

As different features are used, other menu commands will become available. For example, a Window menu is provided so you can cascade, tile or select one of the various windows you have opened while working on a project.

The specific menu commands (related to projects) will be covered under the appropriate section in this chapter.

W32APP Command Arguments

At times, you may want to run W32APP and have the program automatically open a specific file without using the File-Open command. This may be done by entering the name of the file as a command line argument.

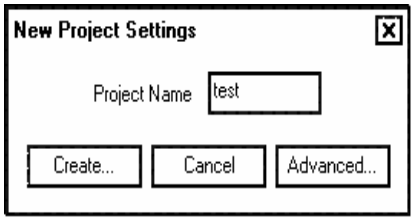
For example, to execute the command and open the project for sample dialog CETDEMO, enter:

```
w32app \cetlib\samples\cetdemo.prj
```

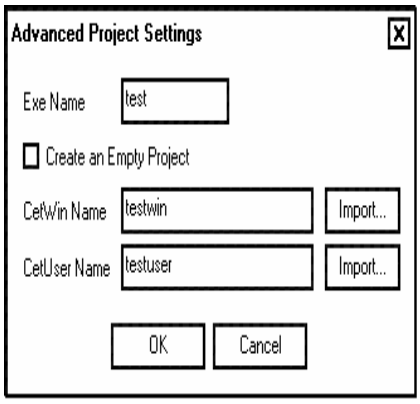
Creating a Project

The File-New command allows you to create a new project. Executing the command will automatically close any open window that contains a source, dialog or another project file. If changes have been made to the open file, the program will ask if the files should be saved before closing the window.

A New Project Settings dialog will prompt for a project name of up to eight characters in length. We have chosen to name our sample project TEST.



If you click the Advanced button, the Advanced Project Settings dialog will display the project name as the default name of the executable. If you change this name, do not use an extension, as **.exe** will be added automatically.



By default, W32APP will create a new set of resource files by copying **cetwin.b** and **cetuser.rc** from **\cetlib** into the local directory with the same name as the project. If the project and executable names are the same, you may need to change the name of the resource files in order to avoid a conflict with the name of the file containing your BASIC source code.

If you do not want to use the default files in \cetlib, you may either:

1. Type in the name of the file or click on the Import button and select the file using a file browser. This method is recommended when you have a customized file which needs additional modifications for a specific project.

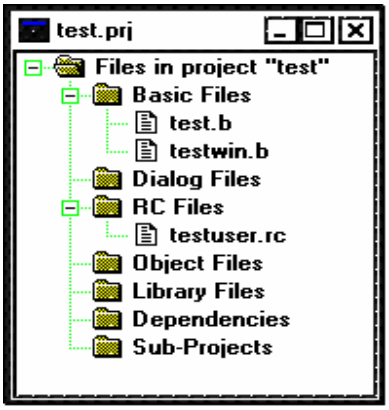
2. Check the Create an Empty Project property when you have already created the resource files, and will be adding the files (with the Add menu) to the project. This is the suggested method to use when creating a project to manage an existing executable.

   Note that if you add the compiled versions of these files (the **.obj** and **.res** files), W32APP will not create additional copies in the project directory.

In our example, the project and executable names are the same so we entered **testwin** and **testuser** as the names for the resource files. (The program adds the **.b** and **.rc** extensions.) This was done to avoid a conflict with the name of our BASIC source code file, **test.b**.

When you click on the Create button in the New Project Settings dialog, a File-Save dialog will appear so you can specify the directory for the project (**.prj**) file.

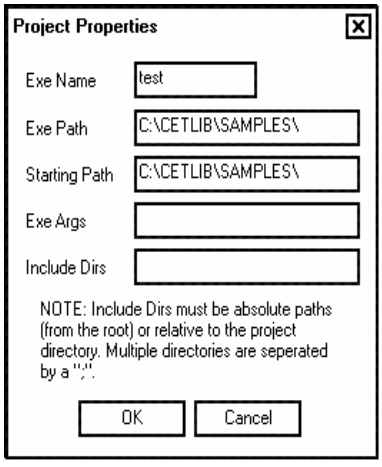
Once a project has been created, the project *tree* will be displayed. The folders in the tree view are just icons used to organize the files by type: Basic, Dialog, RC, Object and Library files, including Dependencies and Sub-Projects. (Refer to a later section for information on sub-projects.)



Folders preceded with a minus sign have been expanded. A plus sign indicates that the folder can be expanded. Click on the plus or minus sign or double click on the folder icon to expand or collapse the list.

Right-click on the folder for the "Files in Project xxx". Then, select the Properties option and note the global properties that are set for all the files in the current project:

1. The name of the executable.
2. Any command line arguments that should be passed to the executable.
3. The Include directories. If you are using #include files that are not in the project directory, be sure to indicate the absolute path from root. Otherwise, enter a path relative to the project directory. Separate multiple paths with a semicolon.
4. The starting path. Set the current working directory for the application to this directory on start-up.



5. The executable path. Have the linker store the **.exe** file somewhere other than the default for the project directory.

It is important to remember that these are the global properties for the currently displayed project. If you use the same directory for all of your #include files, you may find it convenient to set its absolute pathname as the "Global Project Include Directory" with the View-Settings command. Note that the value you enter will have no affect until the next time you run W32APP.

Each folder in the project has its own set of global properties that are used to compile the files in that folder. For example, -c is the default flag that is used when compiling BASIC files so that **.obj** files are created for any BASIC subroutines. If you add the -Zd flag to the global properties, debugging would be enabled in all the **.b** files.

Each file added to the project may have additional properties that will only affect the individual file. Any individual file properties will be appended to the global properties for the project during compilation.

Adding Files to a Project

The Add-Files to Project menu command (F10) displays a File-Open dialog to make it easy to add the necessary files. Consider the following when selecting one or more files to be added.

1. Dialogs or form views are added by selecting the **.b** file. The program will add the corresponding **.idh** file automatically. An error will be detected if you attempt to add an **.idh** file yourself.

2. Add source files with a **.b** extension. If the source contains #include statements, these files will automatically be added to the project as dependencies during the build operation. (If you create a new dependency file after a project is built, use the Build-Update All Dependencies command to add it to the project.)

   An error message will be displayed if you attempt to use the menu to add a file with an **.h**, **.inc**, or **.cpu** extension. (To avoid confusion with existing #include files with a **.b** extension, you may rename these files with another extension so that they will be easily recognizable as dependencies and not BASIC source files.)

3. Add resource files if the Create an Empty Project property was checked. If you add source files (the **.rc** and **.b**), W32APP will compile the files when you build the project and store them (the **.obj** and **.res**) in the project's **\bld**

directory. If you add the compiled resource files, additional copies will not be created for the project.

4. Although you may add a library you have created using the CETLNK32 utility, we recommend that you add the required **.obj** files instead. This way one **.obj** file may be modified and relinked during the Build operation.

   If the modified **.obj** file was part of a library file, the dates for both files would be changed. In that case, W32APP would relink all the files in the library since it would not be able to determine which file had changed.

5. Sub-projects may be added for executables that are called via a CHAIN, LINK or RUN statement. (Please refer to a later section for information on using this feature.)

A group of files may be added at once. If you click on one file and then press the Shift key while clicking on another, all the files in that range will be selected. To select a random group of files, hold down the Ctrl key and click on the ones to be added. Clicking on a selected file will *deselect* it. A message will appear if a file with the same name already exists (in the project).

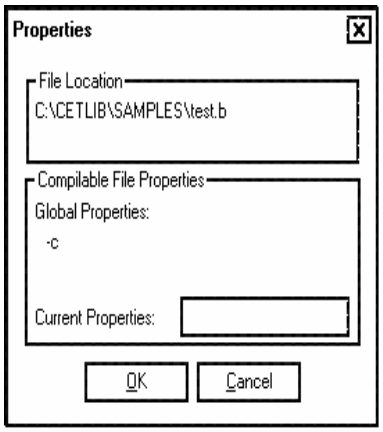
While a project is open, you may create a new source or dialog file. When you save the file, W32APP will ask if it should be added to the project. Responding with "Yes" will add the file to the appropriate folder in the project tree.

When you create a new dependency file (with a .**inc**, .**h** or **.cpu** extension), W32APP will also ask if you want to add it to the project. Unfortunately, we can not suppress this message even though we know that this type of file can not be added during the save operation.

We strongly suggest that you not use the File-Save As command and avoid possible problems in trying to update projects that use the file. For example, suppose you want to rename a dialog that has already been added to a project. Instead of using the File-Save As command, we recommend that you delete the dialog from the project, rename the **.idh** and **.b** files (from the command line), and then re-add the dialog with the new name to the project.

After adding a file to a project, right-click the name of the file to display a context menu with the following options. Any option that is not available for the specific file will be disabled.

Open	Opens the file for editing. A double-left click on a file icon will also open the file.
Delete	Deletes the file from the project, but not the disk.
Compile	Compiles the file using the special flags, if any.
Properties	Displays the default properties: the location of the file and the compiler flag(s) to be used. Since the -32 flag is always set, it is omitted from the display.



Building and Maintaining a Project

After a project has been created, the Build menu may be used to perform a wide variety of operations. There are commands to:

Compile a File

Compiles the selected file using the default flags specified in the file properties. Note that to compile a dialog you must select the **.b** file and not the **.idh** file. The -c flag will be used, by default.

Remember that there are two levels of properties: global and individual file properties. Actually, there are three levels if you consider the "Global Project Include Directory" that may be set to the path that will be used for the #include files in all projects.

The global properties affect all files in the project. During compilation, the compiler flags for the individual files will be appended to those specified as global (properties).

For example, suppose you want to use the W/32 Debugger on one file. To enable debugging, add the -Zd flag to the properties for the individual file. If you add the flag in the properties for the Source files, all the **.b** files will have debugging enabled. (This feature is covered in detail in the next chapter.)

	Individual file properties may also be set to override a global property. For example, even though -c is set (to create/link the **.obj** files) for the project, the -S flag may be used to direct W32APP to terminate after creating a specific **.asm** file.
Build the Project	Compiles all programs and generates the executable. The <project name>**.bld** folder is used to store the object (**.obj**), resource (**.res**) and the **w32error** files. (The contents of the **w32error** file is displayed if an error occurs during compilation.) During subsequent builds, W32APP will check all dependencies and recompile only those that have been updated since the last build. During this process, an attempt is made to delete the existing executable. If the file can not be deleted because of "Access Denied" (which usually means the file is in use), the program will display a message and abort. Note that W32APP can not detect changes made to files referenced by the resource files (**cetwin.b** and **cetuser.rc**). For example, if **cetwin.b** is modified to use a custom toolbar or icon, changes to the **.bmp** or **.ico** files are not detected. You must also open **cetwin.b** and save it in order for W32APP to recompile the resource file during the Build process.
Rebuild All	Forces a rebuild by deleting all target files and calling Build to recompile all the programs in the project.
Update all Dependencies	Searches the files in the BASIC folder for any file referenced with a #include statement and updates the appropriate dependencies. (Use this command to add new dependencies after the project has been built.)
Execute the Project	Runs the executable. If W32APP detects that a file has been modified since the last build (by comparing dates), you will be asked if you wish to rebuild the project before running the executable.
Build the Sub-Projects	Automatically opens and does a "Build the Project".
Rebuild All Sub-Projects	Automatically opens and does a "Rebuild All".

Whether you are compiling a file or building a project, a dialog box will appear to indicate the operation in progress. If the operation is successful, you will be returned to the editor without any message. When an error is detected, the contents of the **w32error** file are displayed so you can see what was wrong.

Note that the Build Project dialog has a Minimize and Cancel button. At any time during the compilation process, you may click the Minimize button and work in another application. When the operation is complete, the program will be restored automatically (under the current application). If you cancel, the program will terminate after the currently executing command has finished. It will not abort a command.

Creating a Sub-Project

Typically, W/32 programs consist of a number of different programs that are called via CHAIN, LINK or RUN statements. Although projects are created to build one target file or executable, the called programs may be added as sub-projects for ease of maintenance.

For example, suppose that **mainprog.exe** chains to **subprog.exe**. You would use the following procedure to add the sub-project:

1. Create a project for **mainprog.exe**.
2. Create a project for **subprog.exe**. Check the Create Empty Project property if the program uses the same menus (and other behaviors) as **mainprog.exe**. In that case, add the compiled resource files (the **.obj** and **.res**) that are in the project directory called **mainprog.bld**. Otherwise, the project will need to have its own copy of the **cetwin.b** and **cetuser.rc** files.
3. Re-open the project you created for MAINPROG (**mainprog.prj**). Use the Add-Files to Project menu command and select **subprog.prj** from the File-Open window.

The Build menu has a special Build Sub-Projects command. When used, W32APP will display a message to indicate that it is checking the dependencies. If the date on any of the files is later than the date of the executable, the sub-project will be rebuilt. If the date is older, the program will go on to check the next sub-project to be rebuilt, if any. Select Build-Rebuild All Sub-Projects to rebuild disregarding all dependencies. In either case, no user-interaction is required.

The W/32 Sample Projects

The W/32 App Builder includes a number of sample programs that have been created to illustrate the various features in the product. A special menu program called W32DEMOS may be used to run the demos.

Projects have been created for all of the sample W/32 programs. All of the files for these projects (including the executables) have been stored in the **\cetlib\samples** directory, by default.

Project	Description
w32demos.prj	A special menu program created to run the sample programs.
cetdemo.prj	The sample dialog described in the *W/32 Dialog Editor* chapter.
invoice.prj	The sample form view described in the *W/32 Dialog Editor* chapter.
calc.prj	The sample dialog used to describe button click processing in the *W/32 Dialog Editor* chapter.
odbc.prj	A program that uses the features described in the *ODBC* chapter.
suprint.prj	A sample dialog created to illustrate how the W/32 function cEnumPrinters may be used to get information about the available printers.

Renaming, Copying or Moving an Existing Project

When a project is created, all of the project information such as its name and location is saved in a binary file with a **.prj** extension. When a file is added to a project, the .**prj** file is updated to include its name and location. Since the individual source files may be located anywhere on the system, all file locations are stored as absolute paths.

The information in the **.prj** file becomes invalid when a project has been renamed, copied or moved to a new location (using an operating system command). Fortunately, W32APP can detect when these changes have occurred and update the project file accordingly.

Renaming a Project

Use the operating system command to rename the **.prj** file. The file must not be open while it is being renamed.

Invoke W32APP to open the project with the new name. The name of the file being opened will be compared with the project name stored in the **.prj** file. When the names are different, W32APP will display a message saying that the project name has changed and the names in the project file must be updated. If you select Cancel, then W32APP will not open the project (or make any changes). Select OK and the internal name in the project file will be changed to the one found in the **.prj** file.

W32APP also checks the name of the executable. If it is the same as the original project name, then it will also be changed to the new name. Otherwise, the name of the executable will remain as is.

Moving or Copying a Project

You can create a new project and add the files in their new locations or follow the procedure given to rename a project. If you do the latter, W32APP will detect that the project path is different and display a message indicating that the internal paths in the project file must be changed. If you select OK, the program will update the project path, and then check the paths of all the individual files. When it finds a path equal to the old project path, W32APP assumes that the file was moved with the project and changes its internal path to the new path.

Note that there may be cases in which this behavior does not work. It could be that (some) files were not moved with the **.prj** file. It could also be that there are files in the project that were not in the project directory (i.e. a shared file).

If there are only a few exceptions (i.e. the project has only one shared file that is not in the project directory), then let W32APP *move* the project, delete the shared file and then re-add the file. This is assuming that the absolute path to the shared file is no longer valid. If the path has not changed, you obviously do not need to do anything.

If the project is renamed and moved, W32APP will perform both procedures at once.

CHAPTER 6

The W/32 Debugger

Introduction

The W/32 Debugger is a special feature that is only available from within the W/32 integrated development environment (**w32app.exe**). After creating a project, this source level debugger may be used to help you track down any errors that may occur during program execution.

This chapter will cover the information you need to use the source code editor and debugging features. (It is assumed that you have already used the features covered in the last chapter and have created a project that you want to debug.)

Windows for Workgroups Note: Because of the problems encountered with Win32s, the features in the integrated development environment are not available for use under Windows for Workgroups. We recommend that you upgrade your development system to either Windows 95 or Windows NT to take advantage of these features.

Using the Source Code Editor

W32APP provides you with an editor that may be used to create, view or modify your source files. The File menu has the following commands to simplify this process:

Command	Function
File-New	Selecting "source" with the File-New command will load the editor with the default file name "Source*n*" where *n* corresponds to the number of new source files that have been opened.
File-Save	Save the file. If you are saving a new (*Sourcen*) file, enter the file name you wish to use. The **.b** extension will be added, by default.

File-Print Print the text file.

Find in Files Search for a specific text item. The "Match Case" option may be used if the search should be case sensitive. Indicate the starting directory and check whether the subdirectories below it should be searched.

The specific type of files to search (e.g. *.b and *.inc) may be selected from the predefined list or entered using your own custom filter in the form of *.ext. (Separate multiple items with a semicolon as in "*.foo;*bar".)

When the search begins, a results dialog will be displayed. As matches are made, the name of the file, the line number and the matching line of text will appear (as shown below). At any time, you may click on an item on the list and be positioned at that line in the file.

c:\pos\source\mainprog.b(234) This is the matching text…

The dialog may be minimized if you want to continue working on something else during the search. To cancel the operation, click the system Close button, the X in the upper-right corner. Clicking on Close will exit the dialog (and delete the output).

The Edit menu commands make it easy to edit source files. The commands are:

Command	**Shortcut Key**
Find	<Ctrl>+F
Find Next	<F3>
Replace	<Ctrl>+H
Select All	<Ctrl>+A
Go to Line	<Ctrl>+G

Note that the Select All command has been added so you can select all the text in the document without having to drag the mouse over all the lines.

In addition to the features that are common to Windows editors, the W32APP editor allows the Insert key to be used to toggle between insert and replace mode.

By default, tab stops are set to 8 characters like in the Windows Notepad. To change the default tab expansion, use the View-Settings command and enter a "Num Chars" value between 1 and 16 in the "Text Editor Tab Stops" group. (Note that this change will take affect the next time that W32APP is invoked.)

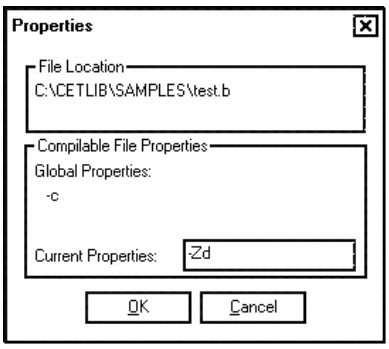
Using the W/32 Debugger

The W/32 Debugger allows you to set break points in your BASIC source code and examine (watch) the value of the variables that you have used in your program. When a break point is encountered, the source file where the break point occurred will be opened and the line number of the break point will be highlighted. The application being debugged is then suspended and control is passed to the debugger so you can examine the variables that are used.

Enabling Debugging

To enable debugging, you must use the -Zd flag (in that case mode) so that debugging information will be inserted into a BASIC source file during compilation.

Debugging may be enabled for one or more source files. To enable debugging for all of the BASIC files in a project, set the -Zd compiler flag in the properties for the "Basic Files". If you only want to include debugging information for some of the files, set the compiler flag in the properties for the individual files. When a source file has debugging enabled, the icon for that file will be displayed in red, instead of blue.



Note that debugging may only be used during development. You can not execute a program that includes debugging information. When you are ready to distribute your application, make sure you disable this feature and re-build (and test) your application.

After compiling the file(s), click on the Debug menu. The commands that are available for your use are:

Break Point	Set a break point at the current cursor position. (Ctrl+B)
Clear All	Clear all the break points that are set.
Show All	Show all the break points that are currently set.
Watch	Display the Watch dialog that lists the variables that are used. (Ctrl+W) This command is initially disabled.
Auto Watch	Automatically display the Watch dialog each time a break point is encountered. This is the default behavior. (Alt+W)

Continue	Resume execution after a break point was encountered. (Shift+F5) This command is initially disabled.
Step	Step to the next executable line of code and cause a break to occur. This command is initially disabled. During a debugging operation, it may be activated by selecting this menu command, by clicking on the Step button in the Watch dialog or by entering Alt+F5.
Start	Start the debugging operation.

Starting the W/32 Debugger

The Debug-Start menu command allows you to start debugging a project (executable) that was compiled with the -Zd flag. Generally, we recommend that you set at least one break point prior to starting the debugger (although you can set a break point while your application is waiting on input). If no break points are set, the application will run as *normal.*

To finish the debugging operation, use either Continue (if no more break points will be encountered) or select Debug-Clear All and then Debug-Continue. Currently, there is no way to stop debugging without letting your application run to completion.

Setting Break Points

Any break points that you set will only be in effect while the project is open. When the project is closed, all settings are removed. (On the other hand, debugging information is compiled into a file and remains there until the file is recompiled without the -Zd flag.)

To set a break point, load the source file into the editor (by double clicking on the file in the project tree view), place the cursor on the line where you want to break and select the Debug-Break Point command or the keyboard accelerator Ctrl+B. The source line will be displayed in red (instead of black) to indicate that a break point has been set at this line. To remove a break point, reposition the cursor on the line and repeat the Debug-Break Point command. The line will be redisplayed in black.

Consider the following when setting break points:

1. Break points always occur at the beginning of a line. To look at a variable after a statement is executed, place a break point at the next line.
2. Although you can set a break point on a non-executable line such as a REM or a $GLOBAL statement, no break point will occur.

3. When a break point is set, every time that line is executed, the break will occur. For example, you will get a lot of breaks if you set a break point inside of a looping structure such as a FOR-NEXT.

4. To break at a statement that is executed from multiple places in your application (i.e. after a statement that you GOTO from many places or inside of a user-defined function or subroutine), enable the break point just prior to the time you want the break to occur and click on Continue until you get to the point where you really want to stop.

5. You can add or remove break points at any time even when the application is executing and waiting for input. For example, suppose the program displays a data entry screen and you want to break when you get to a certain field. If the statement you want to break at is used for every input field, we recommend that you wait until the program prompts you to enter data into the field immediately preceding the one you want.

   At that point, minimize the program and open the program's source file (click on the **.b** file). Find the source line you want to break on and set a break point. Then, restore the application you are debugging and complete the current input operation to reach that break point.

   Select Continue and you will be on the field you are interested in. When you complete the input operation, you will be at the desired break point. Now, you can select the Debug-Watch command and look at the value of the variable(s) in use.

6. When you are debugging an application, you may still get a "Runtime Error at line 0" message (sigh!) due to an internal error condition (possibly in a Windows kernel DLL). Usually, this message refers to an "Access Violation" error such as a pointer to a NULL string when the runtime library was expecting an actual string.

   The error returns "line 0" if the lines in the source file are not numbered (and the -# compiler flag was not used). If you have used line numbers somewhere in your program, then the error message will display the number of the last numbered line that was executed successfully.

   If debugging is enabled, you will get a second message (after the "Runtime Error") that displays the name of the file and the actual line number that caused the error to occur. Note that this message is only accurate if the file containing the error was compiled with the -Zd flag. Otherwise, the second message will also refer to the last line that was executed successfully in a file that has debugging enabled.

Examining Watch Variables

When a break point is encountered, the application being debugged is suspended and control is passed to the debugger so you can examine the variables that are used. The source file where the break point occurred will be opened and the line number of the break point will be highlighted.

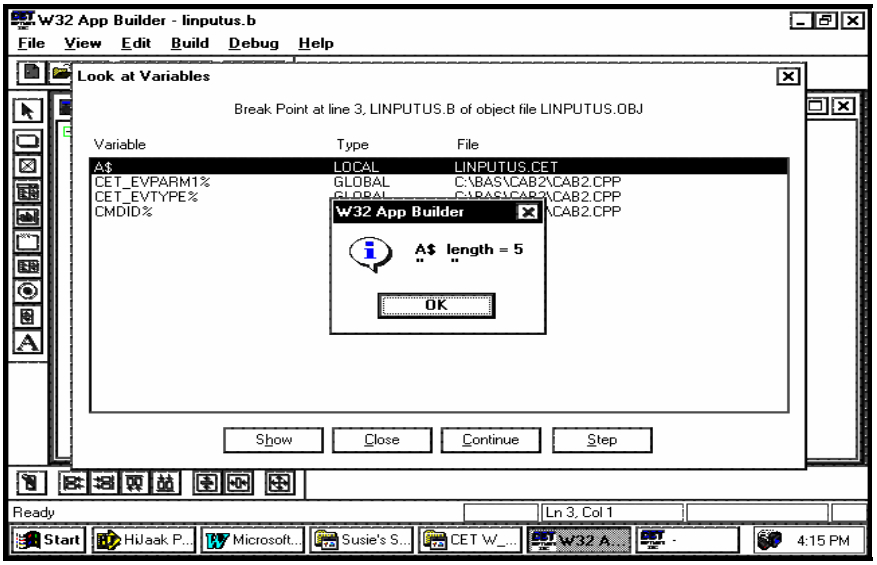
By default, the Debug-Auto Watch menu command is enabled so the Watch dialog is displayed automatically every time a break point is encountered. This is a toggle command.

The Watch dialog contains a list box with all of the variables that you can examine. The window is intentionally large in order to display as many variables as possible at one time.

Since the Watch dialog covers up the source file, you may wish to disable the Debug-Auto Watch feature and select the Debug-Watch command when you are ready to look at the variables. This is the recommended procedure if you plan on *stepping* through the break points in your code.

Note that the list box has three columns; the variable name, the variable type and the name of the source file that the variable was defined in. If the variable is an array, the name will have "()" appended to it.

Also note that when debugging is enabled, W32APP generates special **.cet** files to hold the information it needs about your program. There will also be some



GLOBAL variables in the file **c:\bas\cab2\cab2.cpp**. These are the variables (i.e. CMDID%) that are defined in the resource files your application is using.

The variable type may be either GLOBAL or LOCAL. For example:

Variable	Description
GLOBAL	A GLOBAL variable must have been defined with a $GLOBAL statement. In this case, the file name is the name of the $MAIN source file.
LOCAL	A LOCAL variable is one that is only known within an **.obj** file (local to that file). If your application contains multiple **.obj** files that use the same LOCAL variable name, you will have multiple entries in the variables list box. The file name identifies the specific LOCAL variable in use. This is a single selection list box, so you can only look at the value of one variable at a time. If you have selected an array to look at, all the elements of the array will be displayed. You may scroll the list to locate a particular element.

There are four buttons in the watch dialog: Show, Close, Step and Continue.

Button	Description
Show	The Show button will display the contents of the currently selected variable. You may also show a variable by double clicking it.
Close	The Close button will close the Watch dialog. The program being debugged is left in a suspended state so you may add or remove a break point before continuing execution. Program execution will not be resumed until you select either the Continue button in the Watch dialog or the Debug-Continue menu command.
Step	The Step button may be used to cause a break at the next executable line of code. Once you have encountered a break point, you can activate Step by clicking this button, selecting the Debug-Step menu command or by using the keyboard accelerator Alt+F5. As you Step through your code, the next line that is executed (in a BASIC source file compiled with the -Zd flag), will cause a break point. If you Step over a line that requires user interaction (e.g. LINPUT), the debugger will break after you have satisfied the input operation.

Note that Step will *step* over any calls to C language functions such as cSetFocus. It will also *step* over all the code in a dialog's **.idh** file. (You do not need to be concerned with the code which is generated by the W/32 Dialog Editor.)

Continue The Continue button will resume the program execution until either a new break point is encountered or the application quits.

To finish the debugging operation, use either Continue (if no more break points will be encountered) or select Debug-Clear All and then Debug-Continue. Currently, there is no way to stop debugging without letting your application run to completion.

CHAPTER 7

The W/32 Dialog Editor

Introduction

The W/32 Dialog Editor is a special tool that allows you to create input forms (referred to as form views) and dialog boxes so your programs will have a graphical user interface like other Windows applications.

A form view is a graphical (pixel-based) input screen that is used to replace the main, text-based application window. The form will either be displayed as a full-screen or be set to the size defined with the Dialog Editor. In either case, the window may be moved and resized. Any time a form view is active, you will have full access to any of the menu commands and toolbar buttons.

A dialog box is similar to a form view except that the dialog is modal. That means it must be closed (e.g. by clicking OK or Cancel) before you can continue working with the rest of the application. Menu items, for example, cannot be accessed when a dialog is still open.

You can create a wide variety of dialogs. A simple dialog is a message box that displays some text and waits for the operator to select a button. A complex dialog box could display a number of controls (input fields) along with an image and wait for user input via the mouse or the keyboard.

Basically, dialog boxes may be used anytime you want to get information to or from the operator without affecting what is currently displayed on the screen. In other operating system environments, you may have opened a text-based window (using the Phase One Window System), displayed some text, performed an operation such as LINPUT and then closed the window when you were ready to continue. Under Windows, this would be done with a dialog box.

Both dialog boxes and form views consist of a group of controls. Each type of control has a set of properties that define its behavior. Some controls simply display text that serves as a label to identify another control. Other controls are

designed to get input from the operator. Check, combo and list boxes are a few of the special types of controls that may be used to simplify the data entry process.

For each dialog or form view you create, the W/32 Dialog Editor will generate a program to handle the display and operation. When necessary, the BASIC code may be modified to meet your special needs. Sample programs have been provided to illustrate how this may be done.

The information in this chapter is organized into the following sections to help you learn how to use the editor to develop your own custom applications as quickly as possible.

- Using the general editing features and menu commands.
- Creating a dialog or form view.
- Defining the various types of controls.
- Modifying the layout and operation of the controls.
- Using the runtime functions to alter the look and behavior.
- Modifying and compiling the code generated by the dialog editor.

Creating a GUI Interface - A Developer's Tip

Whether you are creating a new application or converting an existing text-based program, the first step in creating a GUI interface is to familiarize yourself with the features provided in the dialog editor. Sample dialogs and form views have been included to illustrate some of the features may be used.

If you already have text-based programs, the next step would be to use the File-Import Dialog Template command to convert one of your main input screens so that it can be loaded into the dialog editor. After that, you can rearrange and modify the controls until you are satisfied with the new look and operation.

Now, you should be ready to modify the source code and create a *model* form view that includes all the record lookup and standard data validation/formatting routines that you would use throughout your application. These routines plus any that are unique to that program, such as the code to perform file I/O, may be stored in external files that are defined with #include statements (so that they may be easily used in your other programs).

You will probably want to customize the Windows Framework files to display your own menu and toolbar from within your application. Suggestions on how this may be done are covered in a separate chapter in this manual.

Once you are satisfied with the operation of your model program, create (or convert) your next input form. From this point on, it should be mainly a matter of cutting and pasting code (or inserting #include statements) from your model form to your new one. This process has been simplified since both source code files may be open at the same time. Code from your original source file may also be cut and pasted into new #include files just as easily. (In this case, we recommend that you unnumber the lines first.)

The advantages of using this procedure are two-fold. When you are done, you should have

- ✓ A Windows application with a GUI interface that looks and operates the way users expect.
- ✓ A highly structured, event-driven program that is easy to maintain.

Now, to get started…

The W/32 Dialog Editor

The dialog editor is invoked from within the W/32 Integrated Development Environment by running the program **w32app.exe** from the Cetbin folder or by selecting the item on the CET W/32 Application Builder group menu from the Windows Programs menu.

Windows for Workgroups Note: Because of problems with the Win32s product, W32APP is not available for use under Windows for Workgroups. A prior version of the W/32 Dialog Editor is available as a stand-alone executable called **dialog.exe**. This file has been uploaded to the WIN32 library on the BBS for your convenience. (We suggest that you upgrade your development system to either Windows 95 or Windows NT so that you can take advantage of the other features in W32APP.)

Note that although this chapter is not designed as a tutorial, we recommend that you invoke the editor and try out the features as they are discussed, especially if this is the first time you have used a tool like this one.

After executing W32APP, the File, View and Help menu commands will be displayed. Although there are a number of toolbar buttons, almost all of them are initially disabled.

Load the dialog editor by selecting the File-New command and the dialog option. Other menu commands and their corresponding toolbar buttons are now available for use.

In addition to the main toolbar that contains buttons for the File and Edit menus, the dialog editor has two special toolbars to streamline the design process: the Controls toolbar on the left side of the window and the Layout toolbar just above the status bar. All of the toolbars are dockable.

The View menu has options to show or hide any of the toolbars or the status bar. If you change the visibility or location of any of the toolbars, the program will be in the same state the next time you call up the editor. The View-Reset Defaults command may be used to make the status bar and all the toolbars visible and docked at their default locations.

The Controls and Layout commands will be covered in detail in the following sections.

For now, note that each new dialog contains two controls, an OK and Cancel button, in the upper-left corner. To move, resize or copy one of the command buttons, you must first select it by positioning the cursor anywhere over the button and clicking the left mouse button.

The selected button control will now be enclosed in a blue rectangle with resize handles (the solid blocks). Information about the selected control will be displayed in the status bar: its ID number (to be covered later), position (top-left coordinates) and size (height and width).

Moving a Control

To move a control, first position the cursor inside the selected area. After the cursor changes into a 4-way arrow, you can move the control to a new location by either:

1. Clicking the left mouse button and dragging the control. A dotted rectangle will appear to help you move the control.
2. Using the arrow keys to move the selected control in a specific direction. This is the recommended method as it gives you finer control over the movement.

When you are done, deselect the control by clicking in an empty area in the window.

Resizing a Control

To change the size of a control, select it and move the cursor over one of the resize handles. When a 2-way arrow appears, click and pull in the desired direction. You may also resize a control by pressing Ctrl and an arrow key. In either case, click outside the control to end the operation.

The Layout-Grid Size command is provided so you may change the increment used when controls are resized with the arrow keys. The default grid size is 5 pixels. Resizing a control with the mouse is always limited to +/- 5 pixels regardless of the grid size setting.

While you are moving or resizing a control, its size and position will be continually updated in the status bar.

Copying a Control

To copy a selected control, first use the Edit-Copy command. Then, select Edit-Paste and move the new control to the desired position. Since all controls have a set of properties, copying a control will also copy its properties with the exception of its ID value and its BASIC variable name.

Later, you will find that this is a handy feature when designing a form that contains a number of similar controls such as address1, address2, city, etc.

Deleting a Control

A selected control may be deleted by using the Edit-Cut command, the corresponding button on the main toolbar or the Delete key. (A Delete command is also available from the Controls menu.)

Selecting Multiple Controls

You can also move, copy or delete a group of controls at once. Multiple controls may be selected by either of the following methods:

1. Select a single control. Then, hold down the Ctrl key and click on other controls. If the new control is not part of the current selection, it will be added (and enclosed in a blue rectangle). If the control is already part of the group, it will be unselected.

2. Use the rubber-band technique of dragging the mouse around the outside of the desired controls. A dotted rectangle will appear to track the mouse. Release the mouse button when all of the controls are completely enclosed by the rectangle.

   Any control that lies completely within the rubber-band will be enclosed in a blue rectangle. If you did not get all the controls you wanted, quit the selection process (by clicking anywhere outside the selected controls) and start over.

When multiple controls are selected, "Multiple" will appear as the ID number in the status bar. The position and size information refers to the bounding rectangle, the smallest rectangle that encloses all the controls.

Undo and Redo Options

The W/32 Dialog Editor keeps track of the editing changes you make. To fix a mistake select the Edit-Undo command or click on the Undo button on the toolbar. If you decide that you would like to keep a change after all, click the Redo button.

All operations are undo-able, and any undone operation can be redone. Be aware that some operations such as changing a control's properties may not be visually apparent, but can be undone and redone.

The number of Undo/Redo operations is limited only by virtual memory size and can grow quite large. Saving the file will delete all Undo/Redo buffers. (We recommend that you save files periodically.)

Opening an Existing Dialog

The File-Open command displays a window that allows you to view files of a specific type and select the one you wish to open. There are two files for each dialog you create. They are:

1. Files with an extension of .**idh** contain the code used to create and display a dialog. Selecting a .**idh** file will load the dialog or form view into the dialog editor.
2. Associated **.b** files which contain the modifiable source code. Selecting a **.b** file will load it into the text editor.

You can have both the dialog and the source code open for viewing and modification at the same time. You will find that being able to move back and forth between the different windows is a valuable feature that really speeds up the development process.

Saving a Dialog

Each dialog has a *prefix* as one of its properties. By default, this value is used as the name of the dialog files. (The prefix property is covered in the *Defining a Dialog or Form View* section.)

When you attempt to save a new dialog, the editor will display a Save As dialog. This feature allows you to check for a file with the same name. If one exists, cancel the operation and change the prefix.

Importing a Dialog Template

This special feature has been provided for developers who want to create a GUI interface for their existing text-based applications. It allows you to build dialog *templates* from images of your data entry screens so you do not have to start from scratch.

The File-Import Dialog Template command converts all text fields into text controls followed by an optional edit control. At that point, you can rearrange and modify the controls to take advantage of the other features that are available.

Creating a dialog template is a simple 2 or 3-step process.

1. First, create an ASCII text file that contains the screen image to be converted. This is done by using the special argument -cetsd when you execute the W/32 program. For example:

   client -cetsd

   As soon as the program displays the input screen that you want to capture, press the Alt+PageDown keys (at the same time) to write the image to the file **cetsd001.txt**. If the application displays additional screens you want to capture, continue running the program and pressing Alt+PageDown. The images will be written to **cetsd002.txt**, **cetsd003.txt**, etc. (These files will be overwritten when you repeat this process with another W/32 program.)

   The text files are stored in the current working directory when the Alt+PageDown keys are entered. If the application changes its working directory on the fly, you may need to search to find the "cetsd*.txt" files.

   Note that any graphic characters used to draw lines and boxes on the screen will not show up as a graphic character in a text editor. Group controls may be added if you want to create the same effect in your new form view.

2. The image file may be edited. For example, you may wish to remove any numbers that were used in the text fields. If you plan to use shortcut keys, you can enter the ampersand character at the same time, as in:

Before:	After:
1. Company:	&Company

This is an optional step. If these are the only changes you plan on making, they may just as easily be done as you edit the dialog.

Note that W32APP uses three spaces as the default delimiter for text fields. For example, the following text will be considered as three separate items and converted into three separate text controls ("1. Company", "[" and "]"):

1. Company: [ ]

Although it is easy to delete extra controls from a dialog, you may use the Conversion Pairs property (described below) to convert extra characters such as the square brackets to spaces before the file is converted. This is especially recommended if you plan on having W32APP create an edit control for each text item. Otherwise, the form will be extremely cluttered.

3. Execute W32APP and select the Import Dialog Template command from the File menu. A file browser will be displayed to help you select the file to open. The default file extension is **.txt**.

   The file will be converted and loaded with the default dialog properties. All the text items will have been converted to text controls.

   The View-Settings command may be used to modify the conversion process to be used the next time the File-Import Dialog Template command is executed. The "Dialog from Text Template" properties that may be set are:

Property	**Effect**
Conversion Pairs	This property may be used to convert one character to another. Enter the decimal values for the conversion pairs as FROM, TO and separate multiple pairs with semicolon characters. For example, to convert all right and left square brackets to spaces, enter 93,32;91,32.
Separator String	By default, three spaces are used as the delimiter for a text item. Each space is represented by the 2-character sequence "\b". More spaces and/or special characters may be used, if necessary. For example, entering the sequence ":\b\b", will assume that each text item ends with a colon followed by two spaces.
Edit Control	An edit control will be added after each text control when this box is checked. Because edit controls are enclosed in a box, they will be taller than the default size used for the text controls. This means that the edit

controls will normally overlap when converting a screen with single-spaced text items.

All edit controls will be long enough to accept seven characters. Since you will be modifying the edit controls anyway, no attempt was made to create the controls with the same length they had on the text-based screen.

For example, where you used to display long input fields to accept 30 characters or more of name and address information, now you can have a 20-character edit control that scrolls horizontally to enter/display additional characters, if necessary.

Fixed Font — The fixed font property will be set for all text controls when this box is checked. Otherwise, a proportional font will be used.

Once you have converted your screen into a dialog you can easily rearrange the controls on the form and modify them to take advantage of the new control properties.

Defining the Dialog or Form View Properties

The W/32 Dialog Editor allows you to create form views and dialog boxes. A form view is a graphical (pixel-based) input screen that a program will use to replace the main, text-based application window. Having a graphical interface allows you to copy data from a windows application such as Microsoft Excel and paste it into one of your forms via the Clipboard.

Since a form view replaces the main application window, there can only be one active form at a time. Depending on whether the maximize property has been set, the form will either be displayed as a full-screen or set to the size defined with the dialog editor. In either case, the window may be moved and resized.

Any time a form view is active, you have full access to the menu commands and toolbar buttons. A form may be minimized or maximized.

A dialog box is similar to a form view except that the dialog is modal. That means it must be closed (e.g. by clicking OK or Cancel) before you can continue working with the rest of the application. Menu commands, for example, can not be accessed when a dialog is open.

You can create a wide variety of dialogs to be used anytime you want to get information to or from the operator without affecting what is currently displayed on the form or in other dialog boxes.

Each time you create a new dialog using the File-New command, the editor will assign a set of default properties. To display or change a property, double-click the left mouse button in an empty area of the window. (Double-clicking on a control will display its properties.)

All dialogs have the following properties:

Title

The default text "Dialog Title" may be changed to describe the contents or purpose of the dialog. The cDlgTitle function may be used to set or change this text at runtime.

If you define a form, the title "Form View" will be used while you are in the dialog editor. At runtime, this property is ignored, and the form will appear without a title. To display your own title call the cAppTitle function from within the main application program or from the cetPreInit subroutine in **cetwin.b** so that it is used during the initialization process.

Prefix

W32APP assigns a default prefix of "dlg" and a value indicating the number of dialogs that have been created. For example, "dlg1" is the prefix for your first dialog.

When modifying the prefix, it is important to remember that it must be unique since these characters will be added to the beginning of all the names of the $GLOBAL variables and subroutines used in the dialog program. If you change the prefix in a later session, you will be told to update the dialog using the File-Rewrite .IDH and .B (files) command.

The prefix is also used as the default name of the.**idh** and .**b** files that are generated when the dialog is saved. For example, the prefix for one of the sample dialogs is "dlgCET". The name of the disk file is **dlgCET.b**.

Type

The default is to create a dialog which will be called from a main program or another dialog. Dialog boxes are always centered in the application window.

If you specify a form view, the form will be used to replace the main application window. The maximize property may be set so that the form will be displayed as

a full-screen. Otherwise, the application window will be set to the size of the form and displayed starting in the upper-left corner. In either case, the user may move or resize the form. (Dialog boxes may only be moved.)

Note that there are a variety of W/32 window functions available if you wish to have the program resize, minimize, maximize or reposition the application window at runtime. Refer to the *W/32 Window Functions* chapter for details.

Size

In addition to forcing the main application window to full screen (by setting the maximize property), you can explicitly set the width and height values.

The size property is set to 300 x 200 pixels, by default. As you resize the dialog window this size will change accordingly. Refer to the *Implementation Notes* in the Appendix for information on how to think in terms of pixels and a description of the command line argument (-wsize) that may be used to override this property at runtime.

Source File

The dialog files are covered in detail later in this chapter. For now, note that there are two additional properties that may be set to affect the files that are generated. The available options may be set to:

1. NOT retrieve the values of the interactive controls. (Interactive controls are the ones that get user input.)

   By default, CALL statements are generated to retrieve the values of all interactive controls before the dialog ends. That means that you do not have to modify the source code in order to get the information to be passed back to the main program with $GLOBAL statements.

   When creating dialog boxes, you will often find it easier to let the dialog editor take care of this operation. On the other hand, with form views, you will typically write custom code to retrieve the values of the interactive controls depending upon your specific requirements. In that case, you may set this option so that the editor will not generate the code too.

2. NOT make the control variables $GLOBAL:

   By default, every control variable is defined with a $GLOBAL statement so that information may be retrieved from the dialog. These global variables must also be set in the main program. (The Edit-Copy Globals command may be used to copy the $GLOBAL statements to the Clipboard. After that, it is easy to paste them into the $MAIN program.)

If this option is set, the $GLOBAL statements will not be generated. You will need to manually define the variables you need to use (in both the dialog file and the main program). This procedure is documented in the examples provided in the *Dialog Source Files* section.

Defining Controls

All dialogs and form views consist of one or more controls. Some controls simply display text while others get input from the operator.

The Controls menu and its corresponding toolbar contain the commands that are used to enter specific controls. In addition, note:

Command	Function
Test	Tests the design and operation of the dialog without having to compile the program. The dialog will automatically have the gray, 3-D appearance that is common to all Windows applications.
Select	Quits the selection process. For example, if you select "Combo Box" from the menu, by mistake, go immediately back to the menu and pick "Select" to quit the operation.
Delete	Removes the selected control from the dialog. The Delete key may also be used to remove a control.

The rest of the menu commands may be used to enter a specific control. The available controls may be divided into two types: static and interactive. A static control does not accept input from the user. (Text and group controls are static.) An interactive control does accept input or selection.

All controls have a set of properties that affect their display and operation. The properties may be defined in the dialog editor or at runtime by using the special functions described in the *Runtime Functions* section.

To avoid redundancy, the two properties common to all interactive controls are described here.

1. The Basic Variable property contains a name such as EDIT100$ and LBOX300$ that indicates the type of control and the number which have been defined. These names may be changed to be more meaningful.

   By default, each interactive control will be defined with a $GLOBAL statement. The name of the global variable consists of the characters in the dialog's *prefix* (e.g. dlg1) and the name specified in the Basic Variable property so that it is unique to a specific dialog.

2. The unique ID value should not be changed as it is used to identify the control in the dialog program. These numbers will be within a certain range depending on the type of control used. For example:

Control Type	ID Value	
Radio Groups	1 to 99	
Edit Controls	100 to 199	
Command Buttons	200 to 299	(200 = OK and 201 = Cancel)
List Boxes	300 to 399	
Scroll Bars	400 to 499	
Text Controls	500 to 599	
Group Controls	600 to 699	
Combo Boxes	700 to 799	
Check Boxes	800 to 899	
Radio Buttons	900 to 999	

   These control ID values are processed separately so they will not conflict with the ID values associated with the menu commands (ID_WINSUB*n*) used in the resource files. (See the *Dialog Source Files* section.)

All controls also have a set of ten user-defined properties displayed by clicking the right mouse button. These properties may be used to enter special control specific information. An option is available so you can change the labels for these fields. For example:

Field	Label	Example
1	Data File Source	cp\data\custname
2	Data Field Source10	
3	Tag Name	cost
4	Format	N
5	Case Mode	
6	Input Required	Y
7	Default Value	99.99
8	Mask	####.##
9	Prompt Message	Enter the quantity ordered.

The Dialog Editor will load the information defined in these properties into an array that you can access when you need to display, validate or format data in a specific control. The procedure for using controls and their properties will be covered in detail in the *Dialog Source Files* section. Sample programs are provided for your convenience.

Adding controls to a dialog is an easy, 3-step procedure. Simply:

- Pick a control from the menu.

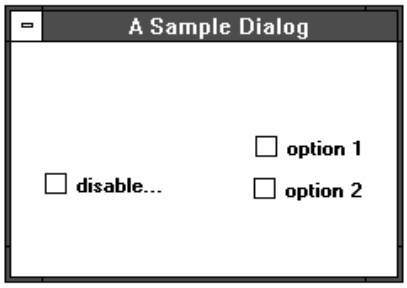
- Drop the cursor in the desired position and click to select the control.
- Double-click (with the left button) on the control to enter its properties. User-defined properties are displayed with a right click.

A sample dialog called dlgCET has been installed in **\cetlib\samples**, by default. Since this dialog contains a variety of different controls, it has been used in some of the following sections to illustrate the different features that are available. (To look at the dialog, run the CETDEMO program or use the Controls-Test command after loading the **dlgcet.idh** file into the editor.)

Check Box Controls

Check boxes are typically used to allow a user to select all, some or none of a group of options and see the result of their choice on the screen. The check box control can have two states: checked and unchecked, which is the default.

The check box controls that were used in the dlgCET dialog are shown here.



When the program is run, the user can select a check box by clicking anywhere within the control (on the box or on the caption). When selected, the state of the control will toggle. If it was unchecked, it will be checked and vice versa.

The (active) check box that has the *focus* will have a dotted-line box around the caption. (The control that has the focus is the one that was most recently tabbed to or clicked on with the mouse.)

To add a check box, pick the control from the menu and a rectangle with an arrow pointing to the top-left corner will appear. This is a special cursor called a *drop* cursor since you may move it to any position in the dialog window. The arrow is used to indicate the top-left position of the control.

Click the left mouse button to drop the control. The word “New” will be displayed along with a small box centered on the left edge of the control. Double-click the left mouse button on the selected item to display and/or modify the Check Box Control Properties.

The following information was entered for the “Disable” check box in the dlgCET dialog:

Property	Value	Effect
Check Box Label	Disable	The text to be displayed beside the check box to identify its contents or use.
BASIC Variable	DISABLE%	The variable used in a BASIC program to reference the data (after the variable has been defined with a $GLOBAL statement).
Control ID	800	The ID value associated with the control.
Initial State	off	The control is initially unchecked, the default.

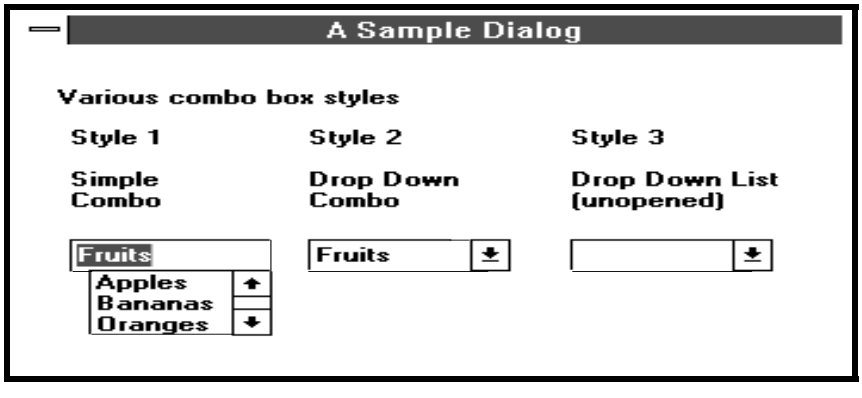
A check box control is an interactive control. Since it gets information from the operator, it has a Basic variable and control ID properties that may be used to get the input. The Basic variable for the "Disable" check box was changed from CKBX800% to DISABLE% to be more descriptive and easier to remember. Since the prefix for the dialog is added to the name to make it unique to that dialog, DISABLE% is defined as the global variable dlgCETDISABLE%.

Since the initial state may be set in the control's properties, the check box for "Option 1" in the sample dialog was set to *checked*. The other check boxes were left *unchecked*, by default. The *Dialog Source Files* section documents how the main program **cetdemo.b** was also modified so that the "Disable" box could be used to check or uncheck the two options simultaneously.

Combo Box Controls

Combo boxes are similar to list boxes in that they provide a list of choices except that they also allow the user to input an additional item. An *initial choice* may be defined as one of the properties of the control. For example, you may want the user to indicate how an order is to be shipped. The default value may be displayed, but the user could also pick from a list of other common methods or enter an alternative.

Three types of combo boxes are available depending on whether the user is allowed to enter a value that is not listed. The following dialog illustrates these styles.



Use either a simple or drop down combo box (with a scroll bar) to accept user input since these styles have an edit control associated with them. Otherwise, use a drop down list style which has an associated static text control.

A simple combo box has the list box control always visible so the control should be sized to show the number of choices you want to have displayed on the form. Drop down (list) styles have a property that allows you to specify the number of lines to be displayed. The default is to show three lines after the user clicks on the down arrow. In any case, the number of lines is a design issue since you can always scroll the list to see the rest of the items.

The strings to be displayed in the list may be specified as a property of the control or set at runtime using the cAddComboContents function. In either case, a call to cGetCombo will return the user's entry.

Note that the initial strings are limited to a total of 175 characters in length. If the total number of characters on the list exceeds that length, then you must initialize the contents of the combo box at runtime.

The sample INVOICE program documented at the end of this chapter illustrates how combo boxes may be used in an application.

To summarize, a combo box control may have the following properties:

Property	**Value**	**Effect**
Initial Strings	*strings*	The strings that will be displayed on the list.
Initial Choice	*default value*	The value that will be initially displayed.
BASIC Variable	CMBO700$	The variable used in a BASIC program to reference the data (after the variable has been defined with $GLOBAL).
Control ID	700	The ID value associated with the control.
Fixed Font	off	The entry will be displayed in a proportional font.
Styles	simple	The control will operate as a simple drop down.
Number lines	3	This value has no effect on a simple combo box.

Command Button Controls

The command button is one of the most commonly used controls. It provides the user with a visually intuitive means of making something happen - the user sees the label on the button and simply clicks on it with the mouse.

Because command buttons are so intuitive, they are often used for menus. The program will automatically detect which button was clicked so that the desired action can be performed. (The W32DEMOS menu program that is provided to run the sample programs illustrates how this feature may be used.)

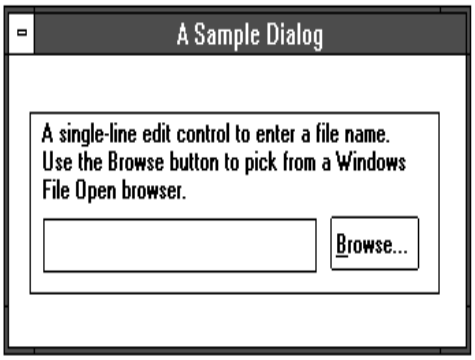
By default, every new dialog has two command button controls labeled OK and Cancel. The properties of the OK button are:

Property	Value	Effect
Button Label	OK	Displays the label on the face of the button.
Control ID	200	The ID value associated with the control.
Default Button	on	Button selected with the Enter key.
End Dialog	on	Button selection will end the dialog.

Notice that the OK button has the Default Button property set. Windows dialogs usually have a default button that is selected when the Enter key is pressed. There can only be one default button per dialog.

Both the OK and Cancel buttons have the End Dialog property set. When either button is selected, the dialog will end and return the values of any interactive controls to their variables. (The ESC key operates like the Close command on the (Windows) System menu and closes the dialog.)

You can add other buttons to your dialog. In the sample dlgCET, a button labeled "Browse" was added so that the operator can enter a filename (in the specified edit control) from a Windows File Open browser.



Note that the first character in the label "Browse" is underlined so the button may be selected with a shortcut key. The text was entered as "&Browse".

Whenever a button is selected, the dialog program detects the event and calls the "BtnClick" subroutine with the global variable CMDID% set to the ID of the button. This allows you to add custom code that will check for an ID value and conditionally perform the appropriate operation. (This process is covered in detail in the *Dialog Source Files* section.)

Edit Controls

An edit control is used to get data from the operator. All the common editing keys such as Delete, Insert, Backspace, Home and End are recognized. If another line is available, it automatically *word wraps* and moves words that will not fit down to the next line.

To add an edit control, pick the item from the Control menu and drop the cursor in the desired position. When you click the left mouse button, the word "Value" will be displayed in the control. Double-click on the selected item to display the Edit Control Properties dialog.

All edit controls have the following properties in common:

Property	Value	Effect
Initial String	"*data*"	The data that appears by default when the dialog is displayed. If no default value is specified, the control will initially be empty.
Max Chars	"*number*"	Limits the characters that may be input.
BASIC Variable	EDIT100$	The string variable that may be used in a BASIC program to reference the data (after the variable has been defined with $GLOBAL).
Control ID	100	The ID value associated with the control.

Note that, in addition to an initial string or default value, you may set the maximum number of characters that may be entered into the edit control. When the default value of zero is used, the maximum is determined by the width of the control. If the Auto HScroll property is checked, an unlimited number of characters may be entered. (There probably is a limit. If so, it is a *big* number.)

Edit controls can be either single or multiple lines. The default is single line. In dlgCET, a single-line edit control was created so that the operator can enter a filename. The Basic Variable property was changed from EDIT100$ to FILENAME$ to be more descriptive and easier to remember. (FILENAME$ is referred to as dlgCETFILENAME$ in the BASIC source code.)

The following properties are available for single-line controls:

Property	Value	Effect
Auto HScroll	on	The entry will automatically scroll horizontally if the width of the control is exceeded.
Password	on	An asterisk will be displayed whenever a character is typed.
Read Only	on	The contents of the field may not be modified.
Fixed Font	on	The characters will be displayed in a fixed font instead of a proportional (true-type) font so that the user's input may be displayed in a columnar format.

When multi-line controls are created, additional properties may be specified as follows:

Property	Value	Effect
Want Return	on	An Enter key may be pressed when entering data.
VScroll	on	A vertical scroll bar will be displayed.
HScroll	on	A horizontal scroll bar will be displayed.
Auto VScroll	on	The entry will automatically scroll vertically if the width of the control is exceeded.
Text Left	on	Text will be left-justified within the control. The default.

Text Center	on	Text will be centered within the control.
Text Right	on	Text will be right-justified within the control.

Notice that the Text Center and Text Right properties are only available for multi-line edit controls. Single-line edit controls are always left-justified.

By default, a multi-line control will word-wrap to the next line when the entry exceeds the width of the control. If the Auto VScroll property is used, a scroll bar will appear if the control is not sized large enough to display all the data. Setting VScroll or HScroll automatically sets the corresponding Auto VScroll. or Auto HScroll property.

It is important to note that any time an Enter key is pressed in a dialog, the "Default Button" is automatically selected. Only a multi-line edit control with the Want Return property set will allow the Enter key to be used during data entry. This feature also allows you to parse the field (using the newline character '\n') when you need to load the input into an array.

Also note that whenever any change occurs in an edit control, a special event is generated. That means that you can add code to verify data as it is entered (on a character-by-character basis) or wait until the user selects another control. This feature is covered in the *Dialog Source Files* section.

Group Controls

A group control may be used to draw boxes to visually separate groups of related controls. Placing controls inside a group control (or placing a group control around other controls) does not in any way alter the normal behavior of the controls.

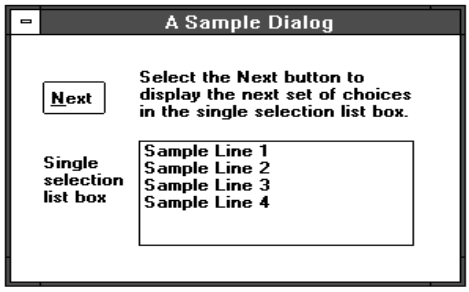
To add a group control, pick the item from the menu and drop the cursor so that its upper-left corner is in the desired position. Click and resize the control until it is the desired size.

Group controls have only two properties: a title and a control ID. Double-click to change the default title of "Group". Entering a title or label is optional.

In the sample dlgCET dialog, a group control with the label "Disable/Enable Controls" was created around the check boxes. A group control without a label was created around the text, edit and button controls that prompt the operator to enter a filename.

List Box Controls

A list box control displays a list of items and allows the user to select one or more items from the list.



The sample dialog dlgCET uses two list boxes; one to make a single selection and another to select multiple items. The single selection list box is shown here along with the Next button which may be used to display the next set of choices.

List box controls accept an array of strings with virtually an unlimited number of elements, displays the items on a list and allows the user to make a selection. By default, the list box will have a vertical scroll bar.

The single selection list box properties illustrate those that may be defined:

Property	Value	Effect
Basic Variable	LBOX300%	The variable that may be used in a BASIC program to reference the data (after the variable has been defined with $GLOBAL).
Control ID	300	The ID value associated with the control.
Sorted	on	Sorts the items before they are displayed.
Multiple Selection	on	Allows the user to select more than one item.
Fixed Font	on	Displays characters in a fixed font so that the items on the list may be arranged into columns.
No Integral Height	off	Resizes the list box at runtime so that only full lines are visible.

A special Initial Strings property is available for design use only. We recommend that you use this feature to make sure that the size of the list box is adequate and that any 'labels' in the text control above the list box are aligned with the columns in the strings.

The fixed-width font property is also available to make it easier to display the list of items in columns.

The No Integral Height option allows you to specify the exact height of the list box although partial lines may appear if you have not sized it correctly. The default is to use an integral height which, when necessary, resizes the list box at runtime so that only full lines are visible.

By default, a list box only allows one item to be selected. The Multiple Selections property may be checked to allow a group of items to be selected by entering a Ctrl-LeftMouse click. A range of items may be selected by clicking on the first item and then clicking on the last with a Shift-LeftMouse.

A list box can also have a Sorted property. When specified, the items will be sorted before they are displayed. If this feature is used, it is important to remember that the index value that is returned for the selected string will probably not match the index in the initialization array. Although you can use the cGetLBoxString function to return the actual string that was selected, we suggest that it is easier to use the BASIC MAT SORT statement to sort the array of strings before initializing the list box contents. This procedure would simplify your code since both index values would always be the same.

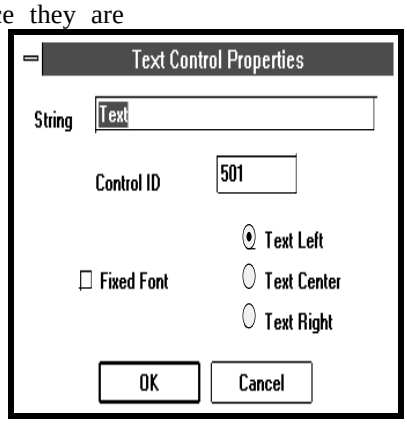
When adding a list box control, remember that its contents are not set in the dialog editor. This control must be initialized in the dialog source file. In the sample dialog, this is done in the dlgCETINIT subroutine with a call to cAddLBoxContents. The actual strings to be displayed are stored in the main program file **cetdemo.b**.

The sample dialog illustrates how the contents of the list box may be changed at runtime (by clicking on the Next button). Special functions are also available to add, delete and insert an item on the list at runtime. All of these features are covered in detail in the *Runtime Functions*section.

Radio Button Controls

A radio button looks like a check box with a circle. While check boxes are separate controls that can be enabled or disabled independently, radio buttons are defined as a group. Only one button can be selected at any given time. If another radio button in the group is selected, the currently set button is unset.

Radio buttons are sometimes referred to as option controls since they are normally used for displaying mutually exclusive options. In the dialog editor, radio buttons have been used to specify the alignment property for a text control because only one of the three options can be set at a time.



A dialog may contain any number of radio button groups. Each radio button must belong to a group (and only one group). When multiple radio buttons are grouped, they will be assigned a Radio Group ID. Each button in the group will then have that group ID as

one of their properties. (A button with a group value of 0 is not a member of a group.)

The group ID is used to retrieve the ID of the checked button in a group (using the cGetRadioGroup function). The return value will be zero when no button is checked (you did not set an initial choice and the user did not select one). Note that a button must be checked in order for it to *tabbed* to.

To create a radio group, first make sure all the buttons are in consecutive tab order starting from the upper-left button in the group. Then, simply rubber-band (drag to select) all of the button controls you want in the group and enter the Layout-Make Radio Group command.

We suggest that you check the operation of each radio group with the Controls-Test command. If the buttons do not behave as expected:

Select the Undo All Radio Groups command
Check the Tab Order
Remake all of your groups

The properties for a radio button control may be:

Property	Value	Effect
Label	"text"	Text is displayed to the right of the button.
Basic Variable	RDIO900%	The variable that may be used in a BASIC program to reference the data (after the variable has been defined with $GLOBAL).
Control ID	900	The ID value associated with the control.
Group ID	1	The control will belong to the first radio group.
Initial State	off	The button is initially unchecked, the default.

The initial state of a button may be set as one of its properties or with a call to either the cSetCheck or cSetCtrlValue function from the dialog's Init subroutine.

Scroll Bar Controls

Vertical and horizontal scroll bar controls are typically used to provide the operator with a quick-and-easy means of selecting an approximate value from a predefined range. The selected value is determined by the position of the scroll bar, a relative value based on the minimum and maximum properties.

The Initial Value property may be used to specify the beginning position. The scroll bar may be moved by clicking on either of the arrows. This changes the value by either +1 or -1 depending on which arrow is clicked. (If the scroll bar is very small, only the arrows will appear.)

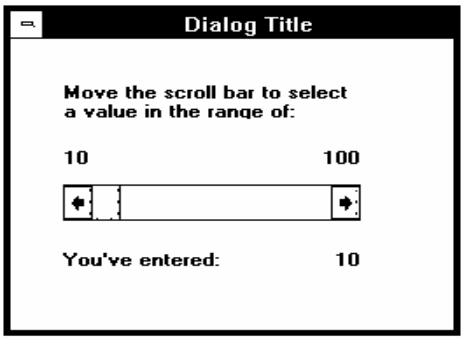
If the scroll bar control is large enough, it may also be moved using the *thumb* or the rectangle displayed between the arrows. When the thumb is moved, the new scroll bar value will be relative to the thumb position. For example, if the thumb is in the middle of the scroll bar, the new value will be half of the range calculated by subtracting the Min from the Max value.

If the user clicks between the thumb and the arrow (referred to as a *page*), then 1/10 of the range will be added or subtracted, by default. You can change the default page value by calling the cSetScrollPage function.

The properties for a scroll bar may be:

Property	Value	Effect
Basic Variable	SCRL400%	The variable that may be used in a BASIC program to reference the data (after the variable has been defined with $GLOBAL).
Minimum Value	0	The minimum value entered will be 0.
Maximum Value	100	The maximum value entered will be 100.
Initial Value	50	The thumb will appear in the middle of the bar.
Control ID	400	The ID value associated with the control.
Horizontal	on	The bar will be displayed horizontally, by default.
Vertical	off	This option will have no effect on the display when the horizontal button is on.

As the scroll bar is moved, the Event routine is called with a notification of the new position. The CMDID% value is the ID of the scroll bar that changed and the CET_EVPARM1% variable will contain the new position. The cGetScrollPos function may also be used to get the scroll bar position. When necessary, you may change the range (the Min and Max values) and the position at runtime by performing calls to cSetScrollRange and cSetScrollPos. (These functions are covered in the *Runtime Functions* section.)



Text Controls

Text controls may display a string of up to 255 characters. Although

this type of control may be used as a heading or caption describing a group of controls, it is commonly used as a label for a corresponding edit control. For this reason, these control may be referred to as label controls in other software development tools.

Text controls may be used to set a mnemonic or shortcut key for an interactive control that has no label associated with it. To give a text control a mnemonic value enter an ampersand before one of the characters in the text. Then, when the mnemonic key is entered, the focus will move to the next interactive control in the tab order. (You can not *tab* to a text control.)

To add a text control, pick the item from the menu, move the drop cursor to the desired location and click to *drop* the control. The word "text" will be displayed in a box with a default size.

Double-click on the selected item to display the Text Control properties which may be defined as follows:

Property	Value	Effect
String	text	The characters to be displayed in the dialog box.
Control ID	500	The ID value associated with the control.
Fixed Font	off	The text will be displayed in a proportional font.
Text Left	on	Text will be left-justified within the control. The default.
Text Center	on	Text will be centered within the control.
Text Right	on	Text will be right-justified within the control.

You may specify the alignment of the text within the control by selecting either the Left, Center or Right property. The default is left-justified.

Text controls can be single or multiple lines. By default, the control will only have enough space for a single line, 8-characters long. When you enter a longer string, the control will have to be lengthened before it can be displayed.

In dlgCET, all of the text controls are multi-line. To create a multi-line text control, simply select the control and enlarge it. Words will automatically wrap to the next line (if available) when the text exceeds the length of the control. To force a line break, enter the 2-character sequence “\n” when specifying the String property. Use “\t” to insert a tab if you wish to create a columnar effect.

Modifying the Layout and Operation of the Dialog

The Layout menu and its associated toolbar contain a variety of commands that may be used to correct design flaws that distract from the overall appearance and operation of the form.

The commands have been organized by category on the menu. They are covered here in the same order.

Alignment

Lining up controls by adjusting them one at a time is often difficult and time consuming. This first group of commands may be used to perfectly align a selected group of controls according to the following rules. If only one control has been selected, these commands will be grayed-out on the menu.

Command	Effect
Align Left	Lines up a group of controls arranged vertically so that they are even with the one furthest to the left.
Align Right	Lines up a group of controls arranged vertically so that they are even with the one furthest to the right.
Align Top	Horizontally arranged controls will be aligned with the one on top.
Align Bottom	Horizontally arranged controls will be aligned with the one on the bottom.

Depending on how the controls are arranged, it may be difficult to use the rubber band method to select the particular controls to be aligned. In that case, click on one of the controls you want. Then, hold down the Ctrl key and click on the others that you want to add to the group. (Clicking on a currently selected control will remove it from the group.)

For example, imagine that you have a number of text and edit controls on the same row. Select the entire row and use the Layout-Align Bottom command to arrange the controls in a straight line, even with the one that was on the bottom.

It is important to emphasize the effect on groups arranged vertically versus horizontally. If you select a group of three buttons arranged in a horizontal row and use the Layout-Align Left command, the editor will use the button furthest to the left as a guide and move the other buttons so that the left edge of the controls are lined up. You will have three buttons, one on top of the other! (The Edit-Undo command is provided to correct this type of mistake.)

Centering

This group of commands may be used to center one or more controls horizontally or vertically in the dialog box. Note that a group of multiple controls is considered a single unit and will be centered. The space between the individual controls will not be affected.

Make Same Size

After adding a number of similar controls, you may decide that the appearance of the dialog box would be improved if the controls were all the same size. In that case, the Layout-Make Same Size command may be used to resize the selected group to be the same size as the upper-left control.

If you want to make the new size match one of the other controls in the group, move that particular control to the upper-left corner temporarily. After resizing the controls, move it back to its correct position.

Tab Order

The Layout-Tab Order command may be used to modify the order in which the Tab key moves between the controls in the dialog. By default, the tab order is the order in which the controls were added. Since the OK and Cancel buttons are the first two controls in a new dialog, they will always come first in the tab order.

To change this order, select the Layout-Tab Order command. By default, each interactive control will be displayed with a small window in the upper-left corner. A number will appear in the window to indicate the current sequence.

To change the sequence, click in each window starting with the control you want to use first. If you want to leave a control in the same order, you must click in its window at the appropriate point. If you make a mistake, just quit and start again. To quit this process simply type any character on the keyboard (or press ESC).

If you want to include text and group controls in the tab order, turn on that menu command before selecting Layout-Tab Order. Although you can not *tab* to a text or group control, this feature allows you to use these controls to set a mnemonic (shortcut) key for an interactive control that has no label associated with it. To give a text control a mnemonic value enter an ampersand before one of the characters in the text. Then, when the mnemonic key is pressed, the focus will move to the next edit control (in the tab order).

Resequence IDs

Each control you create is given a sequential ID value starting with the first number in the range assigned to the specific type of control. For example, edit controls have control ID values between 100 and 199.

Imagine that you have created three edit controls with the ID values of 101, 102 and 103. If you delete the control with the ID value of 102, the next edit control you add will still be assigned a value of 104.

By the time you finish designing a dialog, you may often find that the ID values are out of sequence. While this situation has no affect on the operation of the dialog, you may find it easier to add custom modifications when all of the controls are in the correct sequence.

The Layout-Resequence IDs command may be used to resequence the ID values for specific types of controls or for all controls based on their tab order. When controls are resequenced, the first control of that type in the tab order is assigned the first available ID (within its range) and the next control in the tab order is assigned the next ID, etc.

The range of ID values for the various controls are as follows:

Edit	100-199	Group	600-699
Command Button	200-299	Combo Box	700-799
List Box	300-399	Check Box	800-899
Scroll Bar	400-499	Radio Button	900-999
Text	500-599		

Display Resolution

This command may be used to create a dialog with one of the following display resolutions:

640x480
800x600
1024x768

This feature allows you to design dialogs on a monitor that has a greater resolution than what your customers may be using. For example, if you are using a display resolution of 1024x768, a dialog you design may not fit on a 640x480 screen. To avoid possible problems, use the Layout-Display Resolution command to resize the dialog editor window to that resolution.

If your application uses a form view with a toolbar and/or a status bar, then make sure that the editor window also has the main toolbar and status bar displayed. Otherwise, all of your form may not fit in the application window.

Radio Group Commands

Menu commands are available to either make or undo a group of radio buttons. When you have defined more than one group of buttons, you may find it easiest to use the Layout-Undo all Radio Groups command and remake them if you find that they do not operate as expected. (Refer to the *Radio Button Controls* section for information on using these commands.)

Grid Size

The dialog window is set up with a grid size of five pixels, by default. This means that a control will be moved or resized by five pixels at a time so that you can see the effect. Although it is recommended that you not decrease this value, you may want to make the grid size larger (temporarily) when aligning controls.

Note that this command only has an affect when you are moving or resizing controls with the arrow keys. Mouse movements are always +/- 5 pixels.

Runtime Functions

A variety of functions may be used in the custom code sections of the BASIC source file to modify the appearance and operation of the controls used in the dialog. In all cases:

HDLG% refers to an integer variable used as the dialog handle. HDLG% will begin with the characters in the prefix defined for the dialog. (The sample dialog uses the variable dlgCETHDLG%.)

ID% is a decimal value that identifies a specific control. This value will be with a certain range depending on the type of control used.

Control Type	ID Value	
Radio Groups	1 to 99	
Edit Controls	100 to 199	
Command Buttons	200 to 299	(200 = OK and 201 = Cancel)
List Boxes	300 to 399	
Scroll Bars	400 to 499	
Text Controls	500 to 599	
Group Controls	600 to 699	
Combo Boxes	700 to 799	
Check Boxes	800 to 899	
Radio Buttons	900 to 999	

The use and syntax of the currently available functions are grouped here by the type of control they affect.

Check Box Controls

cGetCheck(HDLG%, ID%, ADDROF(STATE%))

Gets the state of the control specified by ID% and returns it to the variable STATE%. If the control is checked, this value will be 1, otherwise it is 0.

cSetCheck(HDLG%, ID%, STATE%)

Sets the state of the check box specified by ID% to the variable STATE%. If STATE%=1, the check box will be initially checked. If this value is 0, then it will be unchecked.

Combo Box Controls

cAddComboContents(HDLG%, ID%, MATADDROF(ITEMS$))

Sets the current value of the combo box control specified by ID% to the strings in the STR$ array.

cComboAddString(HDLG%, ID%, STR$)

Adds the contents of the string STR$ to the end of the combo box.

cComboDelString(HDLG%, ID%, INDX%)

Deletes the string with the index value of INDX% from the combo box.

cComboInsString(HDLG%, ID%, STR$, INDX%)

Inserts the new string before the item with the index value of INDX%. Set INDX%=-1 to add the string at the end of the combo box.

cComboSetChoice(HDLG%, ID%, STR$)

Sets the default selection for the specified combo box to the string STR$.

cGetCombo(HDLG%, ID%, ADDROF(STR$))

Gets the user's selection for the combo box specified by ID% and returns it to STR$. This string will be empty if no selection was made.

cLimitComboText(HDLG%, ID%, MAX%)

Sets the maximum number of characters that may be entered into the edit control for the specified combo box to the value of MAX%.

cSetCaret(HDLG%, ID%, POS%)

Sets the cursor or caret position in the edit control associated with the specified combo box to POS%. Set POS%=0 to position the cursor before the first

character. Entering a value greater than the number of characters in the control will position the cursor at the end of the data.

Note that this function must be used from within the Event subroutine. If the cursor is not currently in the control specified by ID%, the results are unpredictable.

Command Button Controls

cDefaultBtn(HDLG%, ID%, NEWDEFAULT%)

Changes the current default command button specified by ID% to the button with the ID value of NEWDEFAULT%.

Edit Controls

cGetEditText(HDLG%, ID%, ADDROF(STR$))

Gets the current value of the control specified by ID% and returns it to STR$.

cLimitEditText(HDLG%, ID%, MAX%)

Limits the number of characters that may be entered into the specified control.

cSetCaret(HDLG%, ID%, POS%)

Sets the cursor or caret position in the edit control to POS%. Set POS%=0 to position the cursor before the first character. Entering a value greater than the number of characters in the control will position it at the end or the data.

Note that this function must be used from within the Event subroutine. If the cursor is not currently in the control specified by ID%, the results are unpredictable.

cSetEditText(HDLG%, ID%, STR$)

Sets the contents of the edit control specified by ID% to the variable STR$.

cShowEditWrap(HDLG%, ID%, FLAG%)

Sets the inclusion of soft line-break characters in a multiple-line edit control to on or off. When FLAG%=1**,** two carriage returns followed by a line-feed will be added at the end of each line that was broken due to word-wrapping. If FLAG%=0**,** word-wrapped lines will appear as one long line.

List Box Controls

cAddLBoxContents(HDLG%, ID%, MATADDROF(STR$))

Sets the current value of the list box control specified by ID% to the strings in the STR$ array.

cLBoxAddString(HDLG%, ID%, STR$)

Adds the contents of the string STR$ to the end of the list box. The items will be resorted if the control is defined with the sorted property.

cLBoxDelString(HDLG%, ID%, INDX%)

Deletes the string with the index value of INDX% from the list box.

cLBoxInsString(HDLG%, ID%, STR$, INDX%)

Inserts the new string before the item with the index value of INDX%. Set INDX%=-1 to add the string at the end of the list box. In either case, no resorting will be done.

cGetLBoxSelection(HDLG%, ID%, MATADDROF(RESULTS%))

Gets the current selection from the list box control specified by ID% and returns the index value(s) in the integer array RESULTS%. Since the number of items selected is always returned in element 1, this may be referred to as the count field. The index value(s) of the selected strings start in element 2.

If the list box is set for a single selection (the default), then the array is dimensioned to 2. If an item is selected, the first element is set to one and the index value of the item is stored in element 2.

When a multiple selection list box is used, the size of the array is determined by the number of selected items. Your program will need to loop on the count field (element 1) to retrieve the index values for each of the items.

If no selection is made, the first element of the array will be set to zero and the second element will be empty no matter which type of list box is used.

cLBoxSetSel(HDLG%, ID%, SEL%)

Sets the selected string to the index value of SEL% and makes that string the last one visible in the list box.

cGetLBoxString(HDLG%, ID%, INDX%, ADDROF(STR$))

Gets the selected string associated with the index value INDX% from the list box control specified by ID% and returns it to the variable STR$.

cLBoxTabStops(HDLG%, ID%,STOPS%)

Sets the stops for tab expansion to every STOP% characters.

cLBoxTopIndex(HDLG%, ID%, TOP%)

Sets the first item visible in the list box to the index TOP%. If you are also using a call to cLBoxSetSel, be sure to perform a call to cLBoxTopIndex afterward as cLBoxSetSel changes the top index value.

Radio Button Controls

cGetCheck(HDLG%, ID%, ADDROF(STATE%))

Gets the state of the control specified by ID% and returns it to the variable STATE%. If the control is checked, this value will be 1, otherwise it is 0.

cGetRadioGroup(HDLG%, ID%, ADDROF(SEL%))

Returns the ID value of the currently selected radio button from the group with the specified ID% to the variable SEL%.

The return value will be zero if no button is checked. This would only be possible if the user did not make a selection and no button (in the group) had its Initial State property set to checked.

cSetCheck(HDLG%, ID%, STATE%)

Sets the state of the control with ID% to the variable STATE%. If STATE%=1, it will be initially checked. If this value is 0, then the button will be unchecked. No other buttons in the group will be affected.

The general purpose routine cSetCtrlValue may be used to set the state of a button to checked and uncheck any previously checked button in the group at the same time.

Scroll Bar Controls

cGetScrollPos(HDLG%, ID%, ADDROF(POS%))

Returns the current position of the scroll bar to the variable POS%.

cSetScrollPos(HDLG%, ID%, POS%)

Sets the scroll bar position to POS%.

cSetScrollPage(HDLG%, ID%, PAGE%)

Sets the *page* value for a scroll bar control to PAGE%. The default page value is 1/10 of the range. Assuming that the range is 100, then each thumb movement is

equal to +|- 10 and you would see it move 1/10 of the length of the scroll bar. To set a page value of 1/20, set PAGE% equal to the range divided by 20.

cSetScrollRange(HDLG%, ID%, MIN%, MAX%)

Sets the minimum and maximum values of the scroll bar to MIN% and MAX%.

General Purpose Routines

cAppTitle(TITLE$)

Sets the title of the form view to the string TITLE$.

cDlgTitle(HDLG%, TITLE$)

Sets the title of the dialog to the string TITLE$.

cEnableCtrl(HDLG%, ID%, STATE%)

Enables (STATE%=1) or disables (STATE%=0) a control. A disabled control is grayed out and cannot obtain the input focus.

cEndDlg(HDLG%, ID%)

Calls the routine (in the **.idh** file) to end the dialog with the ID of the control that generated the event. This operation is normally done by setting the End Dialog property for the command button. The PROGRESS sample program at the end of this chapter illustrates how this feature may be used.

cFormBtnClick(VAL%)

Sends a user-defined event such as a button click to the currently active form. For example, you may have a menu item that invokes a new form. If the user selects the item, this function could be called from the ID_WINSUB handler (in **cetwin.b**, by default) to tell the active form to save and exit (with CMDID% set to VAL% in the BtnClick subroutine). If there is no active form, the call will be ignored.

cGetCtrlText(HDLG%, ID%, ADDROF(TEXT$))

Gets the value of static text. Static text areas are the contents of text controls, the title of a group control, and the label of command buttons, check boxes and radio buttons. An empty string will be returned for all other control types.

cGetCtrlValue(HDLG%, ID%, ADDROF(STR$))

Retrieves the contents of interactive controls to STR$. This routine returns all values as strings. Command buttons, text and group controls always return an empty string. The return values for the other controls are:

Control Type	Returned Value
Edit	The contents of the control.
Check boxes	The selected state; either “1” (checked) or “0” (unchecked).
Radio buttons	The selected state; either “1” (checked) or “0” (unchecked).
Scroll bars	The numeric value of the position as a string.
Combo boxes	The user’s selection.
List boxes	The selected string.

Note that the string is returned for list boxes instead of the index value that is returned when using the list box specific routines. Also note that a multiple selection list box control will only return the first string. You must use the cGetLBoxSelection function to retrieve the other selections.

cSendBtnClick(HDLG%, VAL%)

Sends a button click to the BtnClick subroutine with CMDID% set to VAL%. For example, you may want a double-click event from a list box to end the dialog (as if the user selected an item and then clicked the OK button). In that case, add code to the Event subroutine to send a button click (e.g. if the OK button ID is 200, set VAL% to 200). Note that this function may not be used from the Init subroutine since the dialog is not active at that time.

Refer to the PROGRESS sample program at the end of this chapter to see how this feature may be used to display a dialog without requiring any user interaction. The dialog will show a moving progress bar as long as the started process (the CALC demo program) is active. A Cancel button was added to allow the user the cancel the process. This is a useful feature when you want to monitor a process running in the *background.*

cSetCtrlText(HDLG%, ID%, TEXT$)

Sets or changes the value of a static text control with the specified ID value to a new string of characters. Static text areas are the contents of text controls, the title of a group control, and the label for command buttons, check boxes and radio buttons. The call will have no effect on other types of controls.

cSetCtrlValue(HDLG%, ID%, STR$)

Sets the contents of interactive controls to STR$. This routine expects all values as strings. For example, set STR$=1 if a checkbox is to be checked. Otherwise, set STR$=0. Other controls may be set as follows:

Control Type	Value
Edit	The contents of a string.
Check boxes	The state; either “1” (checked) or “0” (unchecked).
Radio buttons	The state; either “1” (checked) or “0” (unchecked).

Scroll bars	The numeric value of the current position as a string.
Combo boxes	The default selection.

Command buttons, list boxes, text and group controls may not be modified with this function. Use a call to cSetCtrlText instead.

cSetFocus(HDLG%, ID%)

Sets the input focus to the control with the specified ID%.

cShowCtrl(HDLG%, ID%, STATE%)

Shows or hides the control with the specified ID value. If STATE%=1, the control will be displayed. The control will be hidden (and disabled) if STATE% is zero.

The Dialog Source Files

The W/32 Dialog Editor generates the code necessary to display the controls according to the properties you have defined, detect events such as a button click and return the values of the interactive controls.

Two files are created when you save a dialog or form view. They are:

File	Function
prefix.idh	The dialog header file contains all the code to display and process the dialog. Do not modify the **.idh** file or you will be unable to load the dialog into the editor. By default, each interactive control is defined as global in this file. The names of the global variables are created by concatenating the dialog's prefix with the BASIC variable name. $GLOBAL statements are not case sensitive so mixed case may be used for readability. In dlgCET, the button that ends the sample dialog is defined as: $GLOBAL dlgCETENDBTN%
Prefix.b	The dialog source file which may be modified to process events according to your specific needs. It is important to note that this file contains no code that affects the display and operation of the dialog. Therefore, you can open the dialog (the **.idh** file) and make modifications to it at the same time you have the source (the **.b** file) open in another window. The only lines in the source file that pertain to a specific dialog are the names of the BASIC subroutines. If you ever change the prefix, you

will be told to update the dialog using the File-Rewrite .IDH and .B (files) command.

For general information on using BASIC subroutines, please refer to a separate chapter in this manual.

The dialog source file is organized into the following parts:

Top of Program	Includes the **.idh** file that the dialog editor uses. If the option to not make the control variables $GLOBAL was checked in the dialog properties, the variables you want to use must be defined here (before the #include statement).
Main Subroutine	Performs a GOSUB statement to call the routine that creates and displays the dialog. This routine is stored in the **.idh** file.
Button-Click Subroutine	Processes button clicks.
Event Handling	Processes special events triggered by a change in the value or state of a control.
Initialization Subroutine	Sets or changes the initial state of list boxes and other controls.

The code for the sample dialog (**dlgCET.b**) is used to explain the individual parts of the program. CETDEMO, the main program that calls the dlgCET dialog, is also listed for your convenience. A variety of other sample programs are documented at the end of this section.

Top of Program

The W/32 Dialog Editor uses the characters "REM@#" to mark special lines in the source file. One of the first of these lines identifies the dialog and the version of the editor that was used to create it.

The REM@# statements that define the dialog controls and layout are stored in the **.idh** (Include Dialog Header) file specified with an #include statement. By default, the **.idh** file also contains the code that returns the interactive control values and the $GLOBAL statements that define the shared variables.

At the top of the **dlgCET.b** file you will find the following statements:

```
REM@# Dialog dlgCET V2.3

REM      first dialog save area
REM@# 1ststart
```

```
REM     The Next button allows you to change the contents of the
REM     list box.  The following statements define the global variables
REM     that are used in processing the list boxes.
$GLOBAL MAT FIRST$, MAT SECOND$, MAT MULTI$
$GLOBAL USESECOND%
REM@# 1stend

#include "Dlgcet.IDH"
```

Note the lines above the #include statement. These REM@# statements mark the first *save area* where you can add custom code. Any code added outside these areas will not be saved the next time you load the file into the dialog editor.

Since all of the interactive controls in the dlgCET dialog are used to pass information to the main program (CETDEMO), we found it convenient to use the default feature and have the dialog editor generate the $GLOBAL statements (in the **.idh** file). That way we only had to define the extra $GLOBAL variables we used to define the array of strings in the list boxes and process the Next button:

```
REM@# 1ststart
$GLOBAL MAT FIRST$, MAT SECOND$, MAT MULTI$
$GLOBAL USESECOND%
REM@# 1stend
```

If the dialog property to "NOT make control variables $GLOBAL" is used, you must define all the required variables in this save area. (We used this feature in the INVOICE program since only a few of the variables had to be shared.)

All $GLOBAL variables must be defined before the #include statement is used to read the **.idh** file and create the dialog. Otherwise, a compiler error will be detected. Note that if you have the same variable defined first as $GLOBAL and later as COMMON, the compiler will detect a duplicate symbol condition and display a warning message. This is not a fatal error.

The W/32 Dialog Editor creates four BASIC subroutines for every dialog. The prefix specified in the dialog's properties is used as the name of the first subroutine. The names of the other subroutines begin with the same prefix. Since the prefix for our sample dialog is "dlgCET", the names of the other subroutines are:

```
$SUB dlgCET
$SUB dlgCETBtnClick
$SUB dlgCETEvent
$SUB dlgCETInit
```

Each of these subroutines will be covered here.

The *prefix* Subroutine

This is the subroutine you need to call from your main program (that starts with a $MAIN statement) whenever you want to display the dialog.

The main program (CETDEMO) invokes the sample dialog with the following statement. Remember that the CALL statement is case sensitive so using a name such as DLGCET would fail.

```
CALL dlgCET
```

Calling this subroutine will create and display the dialog using the following code:

```
$SUB dlgCET
REM@# dlgstart
REM@# dlgend
GOSUB dlgCET.CREATE
$EXIT
```

Note that a save area is provided so that you may enter any special code that must be processed before the dialog is created (and before calling the dialog's Init subroutine). In the INVOICE sample, we used this area to dimension the array to hold the line items.

The GOSUB statement transfers control to the dlgCET.CREATE routine which is stored in the **.idh** file. From there, other calls are made to the subroutines that initialize the controls, detect events, display the dialog and close it.

The statements at the top of the dlgCET.CREATE subroutine define the number of controls (*prefix*CTRLS.CNT%) and dimension the arrays of control IDs (*prefix*CTRLS$) and their corresponding user defined properties (*prefix*PROPS$). The controls array will contain the ID values in tab order.

```
REM   Create the dialog
REM   Add the controls and initialize the CTRLS% and PROPS$ arrays
dlgCET.CREATE:

REM     The controls array
dlgCETCTRLS.CNT% = 16
DIM dlgCETCTRLS%(dlgCETCTRLS.CNT%)
DIM dlgCETPROPS$(dlgCETCTRLS.CNT%, 10)
```

Suppose for example, one of the user-defined properties for the interactive controls is used to indicate that input is required. In that case, when a button click to *end* data entry is detected, you could *walk* through the *prefix*PROPS$ array to find out if there are any required fields which need to be entered.

Please refer to the sample VALIDATE program documented at the end of this chapter to see how the user-defined properties may be used in a real application.

The *prefix*BtnClick Subroutine

The second subroutine named dlgCETBtnClick is called whenever a button is selected. The global variable CMDID% will be set to the ID value of the selected button.

Code is automatically generated (in the **.idh** file) to process the OK and Cancel buttons or the Close command on the system menu. If you use command buttons that should do something other than end the dialog, add code to process the selection in the save area between the REM@# btnstart and REM@# btnend statements.

In the sample dialog **dlgCET.b**, the following code was added to process the Browse and Next buttons. Since check boxes are special types of buttons, the Disable check box is also processed here. Remarks have been used to describe the operations. The syntax of the individual CALL statements is covered in the *Runtime Functions* section.

```
REM@# btnstart
SELECT CMDID%
CASE 202
    REM  The Browse button was selected to display a File Open
    REM  browser and put the selected file into the edit control.
    REM  Change the working directory for the open browser to
    REM  C:\WINDOWS and restore it afterwards.
    CALL Bgetcwd( ADDROF(savdir$) )
    CALL Bchdir( "C:\WINDOWS" )
    CALL cOpenBrowser( ADDROF(SELECTION$), "" )
    CALL Bchdir( savdir$ )
    IF SELECTION$ <> ""
        REM A file selection was made.
        CALL cSetEditText( dlgCETHDLG%, 101, SELECTION$ )
    IFEND
CASE 800
    REM  Use the value of the Disable check box to enable/disable the
    REM  option check boxes.
    CALL cGetCheck( dlgCETHDLG%, 800, ADDROF(STATE%) )
    IF STATE% = 1
        REM Box is checked - disable controls.
        CALL cEnableCtrl( dlgCETHDLG%, 801, 0 )
        CALL cEnableCtrl( dlgCETHDLG%, 802, 0 )
    ELSE
        REM Box is not checked - enable controls.
```

```
            CALL cEnableCtrl( dlgCETHDLG%, 801, 1 )
            CALL cEnableCtrl( dlgCETHDLG%, 802, 1 )
        IFEND
    CASE 207
        REM  The Next button may be used to change the contents of the
        REM  list box.  The $globals for the second list are defined at the
        REM  top of the file.
        IF USESECOND% = 1
            MSGBOX "Already displaying second choices"
        ELSE
            CALL cAddLBoxContents(dlgCETHDLG%,300, mataddrof
(SECOND$))
            USESECOND% = 1
        IFEND
    CEND
    REM@# btnend
```

Note that none of the above buttons will end the dialog (with a $EXIT statement). A dialog ends whenever the operator selects a command button that has the End Dialog property set (like the default OK and Cancel buttons). Pressing the ESC key has the same affect as selecting Cancel. A special case exists when the operator ends the dialog by selecting the Close command from the System menu. Then, a value of 1 is returned.

Special code in the **.idh** file automatically detects an *end* condition, assigns the ID value of the *end* button to the variable dlgCETENDBTN% and retrieves the values of all interactive controls. Since these controls are defined as $GLOBAL, this information is passed to the $MAIN program for further processing.

The code produced for our sample dialog is included here for your information.

```
DlgCETENDDLG0:
call cGetCheck( dlgCETHDLG%, 800, addrof(dlgCETDISABLE%))
call cGetCheck( dlgCETHDLG%, 801, addrof(dlgCETOPT1%))
call cGetCheck( dlgCETHDLG%, 802, addrof(dlgCETOPT2%))
call cGetEditText( dlgCETHDLG%, 101, addrof(dlgCETFILENAME$))
call cGetLBoxSelection(dlgCETHDLG%,300,mataddrof(dlgCETSINGLE%))
call cGetLBoxSelection(dlgCETHDLG%, 301,mataddrof(dlgCETMULTI%))
dlgCETENDBTN% = CMDID%
call cEndDlg( dlgCETHDLG%, CMDID% )
RETURN
```

When creating dialog boxes, you will often find it convenient to have the editor generate the code to retrieve the control values. On the other hand, when creating form views, you will typically write your own code to get these values depending upon your specific requirements. In that case, you may set the dialog property so

that these lines are not generated unnecessarily. (This procedure is documented in the INVOICE example at the end of the chapter.)

The *prefix*Event Subroutine

This a general purpose subroutine that is called whenever a specific type of event occurs. For example, when the program detects a change in an edit control, this subroutine is called so you can validate and format the user's input.

The subroutine consists of a SELECT-CASE statement which may be modified to process the special events. The sample program did not use this feature so no extra code has been included here. (Refer to the sample forms program at the end of this chapter to see what may be done.)

```
$SUB dlgCETEvent
SELECT CET_EVTYPE%
REM@# evtstart
REM@# evtend
CEND
$EXIT
```

The following global variables are used to process these events:

Variable	Value
CMDID%	The ID of the control that generated the event.
CET_EVPARM1%	The new scroll bar position, the next control to get the input focus or the current cursor position, depending on the type of event which ocurred.
CET_EVTYPE%	The code that indicates which type of event occurred (i.e. the type of control that changed).

The following events are recognized:

Event	Type of Change
1	An Edit control has been changed by entering a character or by cutting/pasting a new value. The current cursor (caret) position is returned in the variable CET_EVPARM1%.
2	A List Box control has been changed by selecting a new item with a single click.
3	A List Box selection has been made with a double click.
4	A Scroll Bar position has changed. The new position is returned in the variable CET_EVPARM1%.

5 A Combo box's edit control has changed. (See event 1.)

6 A control has received the input focus. The operator selected a new control by using a mnemonic key, a mouse click or the Tab key.

7 A control has lost the input focus. (The user has selected another control.) The ID of the next control to receive the focus is passed in CET_EVPARM1%. This ID value will be zero if the next control is not part of the dialog. (A user may have activated another application by clicking its window.) If you are validating input fields, always make sure that the next control is not a Cancel button before you notify the user of an illegal entry.

8 A combo box selection has been made with a single click.

9 A combo box's list box is about to drop down. This feature is especially handy when you want the user to be able to enter a new record or look up an existing one. In that case, you can check the contents of the edit control and dynamically alter the items on the (drop down) list based on what (if anything) the user has entered.

Since "An Edit control has changed" event #1 is generated whenever any change occurs in an edit control, you can add code here to verify data as it is entered instead of waiting until the dialog is closed.

The global variable CMDID% will be set to the ID value of the edit control. The following function gets the current value of the multi-line edit control in the sample dlgCET dialog and returns it to the variable dlgCETFILENAME$:

```
cGetEditText( dlgCETHDLG%, 101, ADDROF(dlgCETFILENAME$) )
```

This feature allows you to validate the contents of a control on a character-by-character basis without having to wait for the user to select another control. For example, suppose the edit control prompts for an entry in a specific format. If the user types in the string "ABC", three calls will be made to this subroutine - one for each character.

The same method may be used to check for the first character of an entry and dynamically change it to upper case, if necessary.

The sample INVOICE program uses the following procedure to process the data in the edit control after the field has been entered (and has lost focus):

1. Set a 'modified' flag in CASE 1 ("An Edit control has changed" event).
2. Check the modified flag and (conditionally) perform the validation routine in a CASE 7 statement ("A Control has lost the input focus" event).

The *prefix*Init Subroutine

The last subroutine is called as the dialog is being initialized (before it is displayed). This is where you should set or change the initial state of any controls.

If you have defined a list box, the initial strings must be added here. The following code is taken from the **dlgCET.b** for the sample dialog. In this case, the array of strings to be displayed in the two list boxes are actually set in the CETDEMO program. The arrays have been defined as $GLOBAL so a call to cAddLBoxContents may be made in **dlgCET.b** to set the current value of the controls to the arrays.

```
$SUB dlgCETInit
REM@# inistart
REM Initialize the contents of both of the list boxes.
CALL cAddLBoxContents( dlgCETHDLG%,300, mataddrof(FIRST$) )
CALL cAddLBoxContents( dlgCETHDLG%,301,mataddrof(MULTI$) )
REM@# iniend
$EXIT
```

At the bottom of this subroutine is a special save area that may be used to store any custom (GOSUB) routine or user-defined functions. This feature was used in some of the other sample programs such as INVOICE.

It is also important to note that no error trapping was done in this program. In a *real* application, each BASIC subroutine where an error might occur should have an "ON ERROR GOTO" statement. In this case, the error routine would also be stored in the last save area.

Compiling a Dialog

A dialog program is a BASIC subroutine since it contains $SUB statements. The following example shows how you would use OBWIN to compile the individual modules into separate **.obj** files and then link them together.

```
obwin  -c  dlgcet.b

obwin  -o  cetdemo  cetdemo.b  dlgcet.obj
```

This operation is automated when you use W32APP, the W/32 Integrated Development Environment utility, to create projects to generate and maintain your program executables. Projects have been created for CETDEMO and all of the other W/32 sample programs to illustrate the benefits of using this feature.

The Main Program

A simple BASIC program called CETDEMO was created to display the sample dialog that was used to illustrate the features that were covered in this chapter. $GLOBAL variables have been defined so that the information entered in the dialog will be passed back to the main program where it will be displayed on the screen.

Note that all $GLOBAL variables must be defined immediately following the $MAIN statement. If these variables are defined in the **.idh** file, you may open the dialog and select the Edit-Copy Globals command. The $GLOBAL variables will be copied to the ClipBoard so they can be pasted into the main program.

The **cetdemo.b** program has been heavily remarked and included here for your convenience.

```
$MAIN
REM A sample program that displays a multi-function dialog box.
REM The following global variables define the controls in the dialog.
$GLOBAL dlgCETENDBTN%
$GLOBAL dlgCETFILENAME$
$GLOBAL dlgCETOPT1%
$GLOBAL dlgCETOPT2%
$GLOBAL dlgCETDISABLE%
$GLOBAL MAT dlgCETSINGLE%
$GLOBAL MAT dlgCETMULTI%
REM The following global variables are defined to pass the list box contents
REM to the dialog and allow the program to determine which set of strings
REM to display in the single selection list box.
$GLOBAL MAT FIRST$, MAT SECOND$, MAT MULTI$
$GLOBAL USESECOND%
REM Dimension the list box arrays.
DIM FIRST$(10), SECOND$(3), MULTI$(10)
REM Initialize the second array flag to false and initialize the list box items.
USESECOND% = 0
FIRST$(1) = "This is selection 1"  \  FIRST$(2) = "This is selection 2"
FIRST$(3) = "This is selection 3"  \  FIRST$(4) = "This is selection 4"
FIRST$(5) = "This is selection 5"  \  FIRST$(6) = "This is selection 6"
FIRST$(7) = "This is selection 7"  \  FIRST$(8) = "This is selection 8"
FIRST$(9) = "This is selection 9"  \  FIRST$(10) = "This is selection 10"
SECOND$(1) = "This is the second selection 1"
SECOND$(2) = "This is the second selection 2"
SECOND$(3) = "This is the second selection 3"
MULTI$(1) = "This is multi selection 1"
MULTI$(2) = "This is multi selection 2"
MULTI$(3) = "This is multi selection 3"
```

```
MULTI$(4) = "This is multi selection 4"
MULTI$(5) = "This is multi selection 5"
MULTI$(6) = "This is multi selection 6"
MULTI$(7) = "This is multi selection 7"
MULTI$(8) = "This is multi selection 8"
MULTI$(9) = "This is multi selection 9"
MULTI$(10) = "This is multi selection 10"
REM  Invoke the dialog by calling the BASIC subroutine.
CALL dlgCET
REM Check to see if the dialog was canceled and display the appropriate
REM message. A canceled dialog will return either ID=1 if the dialog was
REM closed from the system menu or ID=201 when the Cancel button was
REM selected.
IF dlgCETENDBTN% = 1
      MSGBOX "Dialog canceled from the system menu!"
      QUIT
ELSE
      IF dlgCETENDBTN% = 201
            MSGBOX "Dialog canceled with the Cancel button!"
            QUIT
            IFEND
IFEND
REM Display the dialog results on the console.
PRINT
PRINT "The state of the disable... check box was: "; dlgCETDISABLE%
PRINT
PRINT "The contents of the filename edit control is: "; dlgCETFILENAME$
PRINT
REM Display the single selection list box choice. First check to see if a
REM selection was made. (The count will be 0 for no selection.)
IF dlgCETSINGLE%(1) <> 0
      IF USESECOND% = 1
            REM Get element 2 from the second array.
            PRINT "The single selected item was: "; SECOND$
(dlgCETSINGLE%(2))
      ELSE
            REM Get element 2 from the first array.
            PRINT "The single selected item was: "; FIRST$
(dlgCETSINGLE%(2))
      IFEND
IFEND
REM Display the multi selection list box choice(s).
PRINT
PRINT "The number of strings selected in the multiple selection list box was:
"; dlgCETMULTI%(1)
FOR I% = 2 TO dlgCETMULTI%(1) + 1
```

```
        PRINT "     "; MULTI$(dlgCETMULTI%(I%))
        NEXT I%
    WAIT
    END
```

INVOICE - A Sample Program Using a Form View

The INVOICE program illustrates how a form view may be used to replace the text-based, application window. All the invoice *header* information is entered into the form. A dialog box is used to enter up to 99 line items. $GLOBAL variables are used to pass information between the BASIC subroutines.

A variety of other features are used in this program. For example, INVOICE shows how a combo box and special command buttons may be used to facilitate entering new records or looking up ones that already exist. Custom code has also been added to handle records when a File, Cancel or Exit button click is detected. Special data validation and formatting examples are provided.

W32APP, the W/32 Integrated Development Environment utility, was used to create a project for the INVOICE executable called **invoice.prj**. We recommend that you load the project into W32APP, run the executable and look through the source files to see what was done. The source files for this program are:

invmain.b The source code for the main program.

dlginv.b The source file for the form view (**dlginv.idh**) which displays the input screen, formats and validates the user input, calls the dialog to get the line items and performs all the file I/O operations.

dlgline.b The source file for the dialog (**dlgline.idh**) that gets the invoice line items.

All of the required files are installed automatically with the W/32 Application Builder (in **\cetlib\samples**, by default). This software has also been included in the **demoprj.zip** file which may be downloaded from the WIN32 library on the BBS in Colorado.

INVOICE was compiled using the default Windows Framework so no special menu or toolbar is displayed. One modification was made to **invwin.b** (a copy of **cetwin.b**) to add a line to set the background color to white. This was done so that the white-on-black, text-based window would not appear during the initialization process. (We could have also set the default foreground and background colors to black on white in the **w32app.w32** file.

The Main Program

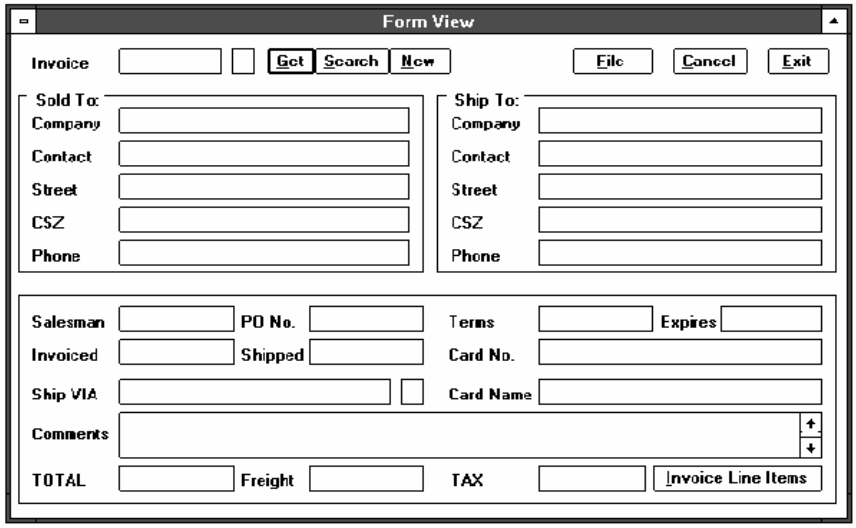
The main program, **invmain.b**, defines the global variables that are passed between the subroutines in the form view, calls the function to display the title on the form, calls inv1 which is the name of the main subroutine for the form stored in the **dlginv.b** file and quits.

```
$MAIN
$GLOBAL mat rec$
$GLOBAL docModified%
$GLOBAL newFlag%
$GLOBAL dataEntered%
CALL cAppTitle("Sample W/32 App Builder Program - INVOICE")
CALL inv1
CALL cForceExit
QUIT
```

Although you may want to open and close the data files you need from within the main program, we suggest that all other processing is done in the source file for the form. Otherwise, a flash will appear each time you leave the form to return to the text-based, application window used by the main program.

The Source File for the Form View

The form stored in **dlginv.b** displays the following input screen to get invoice information.



The modifications made in the save areas in each of the subroutines have been included in this manual for your convenience. The code has been heavily remarked to document what was done.

The dialog property to not make the controls global was used because all control processing was handled within the form view itself by using the control ID values. Only a few $GLOBAL variables were needed to pass information to and from the subroutines in the form (**dlginv.b**) and the line item dialog (**dlgline.b**). These are defined in the first save area - before the #include statement.

```
REM@# Dialog inv1 V2.3
REM@# 1ststart
$GLOBAL mat rec$          \ REM the invoice record array
$GLOBAL docModified%      \ REM used to check changes in record
$GLOBAL newFlag%          \ REM used in the line item dialog
$GLOBAL dataEntered%      \ REM used in the line item dialog
REM@# 1stend
#include "inv1.IDH"
```

Making all the control variables $GLOBAL would not only have added unnecessary lines of code to the **.idh** file, but the statements would also have had to be duplicated in the main program.

$SUB inv1

Only one line of code was added to the save area in the main subroutine:

```
REM Dimension the invoice record array to hold 373 elements
REM (23 items in this Form and 350 Line Items in the “line” dialog).
dim rec$(373)
```

$SUB inv1BtnClick

Code has been added to this subroutine to handle a number of command buttons. Please refer to the remarks to see exactly what was done.

Note that if we had used any check boxes or radio buttons, they would have been processed in this subroutine too.

```
SELECT CMDID%
CASE 200
    REM Search Button
    msgbox "not implemented"
    $exit
CASE 201
    REM Invoice Line Items Button
    REM Invoke the line item dialog. If any line items are updated,
    REM the dialog will update the global invoice record array and set
```

```
      REM the doc modified flag which is also global.
      CALL line
      REM Recalculate invoice total after returning from line item dialog
      REM and display the result.
      GOSUB calc.total
      CALL cSetEditText( inv1HDLG%, 120, total$)
      $EXIT
CASE 202
      REM File Button
      REM Retrieve all values and write the record.
      REM First get the invoice number for the key.
      CALL cGetCombo( inv1HDLG%, 701, ADDROF(key$))
      REM Copy all control values to the record array.
      ctrl% = 101
      FOR i% = 1 to 22
         CALL cGetEditText( inv1HDLG%, ctrl%, ADDROF(rec$(i%)))
         ctrl% = ctrl% + 1
      NEXT i%
      CALL cGetCombo( inv1HDLG%, 700, ADDROF(rec$(23)))
      REM Write the record array to the isam file and clear the doc
      REM modified flag and all of the controls.
      MAT WRITE #1,key$: rec$
      docModified% = 0
      GOSUB clear.ctrls
      REM If it's a new invoice, add the number to the combo box
      REM list and get the next available number.
      IF newFlag%
          CALL cComboAddString(inv1HDLG%,701, str$(newInvoice%))
          newInvoice% = newInvoice% + 1
          newFlag% = 0
          dataEntered% = 0
      IFEND
      REM Wait for another invoice number request and then move
      REM back to the invoice control.
      GOSUB WaitForInvoice
      CALL cSetFocus( inv1HDLG%, 701 )
      $exit
CASE 203
      REM Cancel Button
      REM Cancel the current operation. Clear all edit controls and doc
      REM  modified and new invoice flags.
      GOSUB clear.ctrls
      docModified% = 0
      newFlag% = 0
      dataEntered% = 0
      REM Wait for another invoice number request and then move
```

```
        REM back to the invoice control.
        GOSUB WaitForInvoice
        call cSetFocus( inv1HDLG%, 701 )
        $exit
    CASE 204
        REM Exit Button
        REM Detect when data has changed and warn the user that the
        REM record should be filed or canceled before quitting.
        REM $EXIT is used to exit the form view.
        if docModified%
            msgbox "You have a modified record in progress" + chr$(10) +
"Select either File or Cancel first"
            $EXIT
        IFEND
        REM Clear the "exit app" dlg flag and end the form
        call cForceExit
        call cEndForm( inv1HDLG% )
        $EXIT
    CASE 205
        REM New Button
        REM Create a new invoice. First set the doc modified flag and
        REM clear the edit controls. Then, copy the new invoice number
        REM to the combo control.
        docModified% = 1
        newFlag% = 1
        GOSUB clear.ctrls
        CALL cSetCtrlValue( inv1HDLG%, 701, STR$(newInvoice%) )
        REM Set default values for some controls. These may also have
        REM been set in the control properties.
        CALL cSetCtrlValue( inv1HDLG%, 115, "NET 10" )
        CALL cSetCtrlValue( inv1HDLG%, 700, "UPS BLUE" )
        REM Initialize the new record by adding the first field
        REM (the line item number) of each line item.
        MAT rec$ = ( "" )
        num% = 1
        FOR i% = 24 to 373 step 7
            rec$(i%) = STR$(num%) + "."
            num% = num% + 1
        NEXT i%
        REM Call the routine to enter the invoice and then set the
        REM focus to the first edit control.
        GOSUB EnterInvoice
        CALL cSetFocus( inv1HDLG%, 101 )
        $EXIT
    CASE 206
        REM Get Button
```

```
REM Get an existing record. First, clear the doc  modified flag
REM and get the invoice number from the combo control. The
REM invoice number is the key.
docModified% = 0
CALL cGetCombo( inv1HDLG%, 701, ADDROF(key$))
MAT read #1,key$: rec$
IF eof(1)
    msgbox "can't find record"
    call cSetFocus( inv1HDLG%, 701 )
    $EXIT
IFEND
REM Copy the record fields to the edit controls and call routine to
REM enter invoice. After returning, set the focus to lst edit control.
GOSUB display.ctrls
GOSUB EnterInvoice
CALL cSetFocus( inv1HDLG%, 101 )
$EXIT
```

$SUB inv1Event

This subroutine is called when special events occur. INVOICE illustrates how code may be added to calculate, validate and format the user's input. When a change in an edit control is detected (CASE 1), the editModified flag is set. The actual processing is done when the control has lost its focus (CASE 7).

The VALIDATE program documented at the end of this chapter illustrates another method for validating and formatting data. You probably have similar routines that are common to all your programs. In that case, you could store them in individual files and use #include to merge the code during compilation.

```
case 1
    REM An edit control has changed so set the flag used to detect
    REM that the invoice has been modified.
    docModified% = 1
    REM Set flag to check field formatting and calculate the inv total.
    editModified% = -1
case 5
    REM A combo control has changed so set the document has been
    REM modified flag. We only care about  "Ship Via" - not "invoice".
    if CMDID% = 700 then docModified% = 1
case 7
    REM Check flag to see if an edit control has changed.
    if editModified%<>0
        REM Reformat when data is entered in a  control defined as a
        REM date or phone in the user-defined properties.
        i% = 1
        while 1
```

```
                REM Search CTRL%() for a matching CMDID%.
                if inv1CTRLS%(i%)=CMDID%
                   if inv1PROPS$(i%,8) = "DATE"
                        gosub date.format
                        call cSetEditText( inv1HDLG%, CMDID%, date.fld$)
                   ifend
                   if inv1PROPS$(i%,8) = "PHONE"
                        gosub phone.format
                        call cSetEditText( inv1HDLG%, cmdid%, phone.fld$)
                   ifend
                break
                ifend
             i% = i%+1
             wend
          REM If freight or tax fields changed, set flag to recalculate total.
             if CMDID%=121 or CMDID%=122
                gosub calc.total
                call cSetEditText( inv1HDLG%, 120, total$)
             ifend
          editModified% = 0
          ifend
       case 8
          REM If an invoice is selected from the combo list box, then get it.
          if CMDID%=701 then call cSendBtnClick( inv1HDLG%, 206)
```

$SUB inv1Init

This subroutine is used to initialize the various controls that are used. In the INVOICE program, we load the pattern masks used to validate and format the phone field. We also initialize the combo box list by opening the invoice file and reading the existing invoice numbers. The last key read is assigned to the variable used to get the next available number.

```
REM Load pattern masks for validating and formatting PHONE fields.
dim phone.msk$(6), phone.map$(6)
for i% = 1 to 6
   read phone.msk$(i%),phone.map$(i%)
   next i%
data "(###) ###-####","!""#$%&'()*+,-."
data "(   ) ###-####","!""#$%&'()*+,-."
data "##########","""#$'()+,-."
data "### ###-####","""#$&'()*+,-."
data "#######","'()+,-."
data "###-####","'()*+,-."
REM Initialize the combo list box with the current invoice numbers.
dim invcs$(10000)
```

```
REM Open the isam file and retrieve the keys (the invoice number).
REM Use the last valid key to assign a new invoice number.
open #1:".\invtst", update indexed
i% = 1
lastKey$ = ""
newInvoice% = 1
while 1
    readnext #1,key$: dummy$
    if eof(1) then break
    invcs$(i%) = key$  \  i% = i% + 1  \  lastKey$ = key$
wend
if lastKey$ <> "" then newInvoice% = VAL(lastKey$) + 1
REM Initialize the combo control. Make sure the last array  element is
REM set to NULL since cAddComboContents will read until a NULL
REM entry or the end of the array.
invcs$(i%) = ""
call cAddComboContents( inv1HDLG%, 701, mataddrof(invcs$) )
REM Clear the doc modified and new invoice flags and any arrays that
REM are no longer needed.
docModified% = 0
newFlag% = 0
dataEntered% = 0
clear invcs$
REM Disable the controls until an invoice number is entered.
gosub WaitForInvoice
```

The routines called with a GOSUB statement are entered in the last save area between the usrstart and usrend remarks. They are:

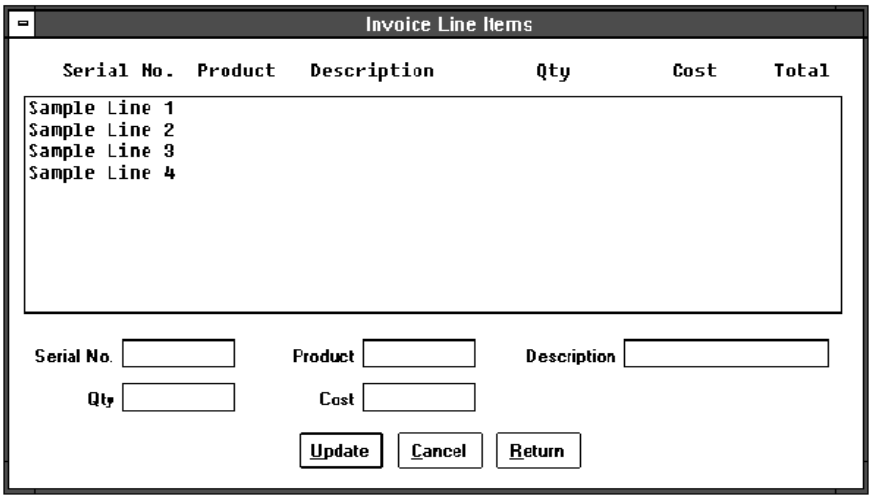
```
clear.ctrls:
    REM Clear all the controls...
    REM First, the edit controls and then combo boxes:
    for ctrl% = 101 to 122
        call cSetCtrlValue( inv1HDLG%, ctrl%, "" )
    next ctrl%
    call cSetCtrlValue( inv1HDLG%, 701, "" )
    call cSetCtrlValue( inv1HDLG%, 700, "" )
    return
display.ctrls:
    REM Display all control values in the data record.
    ctrl% = 101
    for i% = 1 to 22
        call cSetCtrlValue( inv1HDLG%, ctrl%, rec$(i%) )
        ctrl% = ctrl% + 1
    next i%
    REM Set the combo control with the keys.
    call cSetCtrlValue( inv1HDLG%, 700, rec$(23) )
```

```
        return
    WaitForInvoice:
        REM Disable all the controls except for Invoice, Get, Search,
        REM New, Cancel and Exit. First the edit controls:
        for ctrl% = 101 to 122
            call cEnableCtrl( inv1HDLG%, ctrl%, 0 )
        next ctrl%
        REM The combo controls:
        call cEnableCtrl( inv1HDLG%, 700, 0 )
        REM The Line Items/File button:
        call cEnableCtrl( inv1HDLG%, 201, 0 )
        call cEnableCtrl( inv1HDLG%, 202, 0 )
        REM Enable Get/search/new/invoice:
        call cEnableCtrl( inv1HDLG%, 206, 1 )
        call cEnableCtrl( inv1HDLG%, 200, 1 )
        call cEnableCtrl( inv1HDLG%, 205, 1 )
        call cEnableCtrl( inv1HDLG%, 701, 1 )
        return
    EnterInvoice:
        REM Enable all the controls except Invoice, Get, Search, New,
        REM Cancel and Exit. First, the edit controls:
            for ctrl% = 101 to 122
            call cEnableCtrl( inv1HDLG%, ctrl%, 1 )
        next ctrl%
            REM The combo controls:
            call cEnableCtrl( inv1HDLG%, 700, 1 )
            REM The Line Items/File button:
            call cEnableCtrl( inv1HDLG%, 201, 1 )
        call cEnableCtrl( inv1HDLG%, 202, 1 )
        REM Disable Get/search/new/invoice:
        call cEnableCtrl( inv1HDLG%, 206, 0 )
        call cEnableCtrl( inv1HDLG%, 200, 0 )
        call cEnableCtrl( inv1HDLG%, 205, 0 )
        call cEnableCtrl( inv1HDLG%, 701, 0 )
        return
    calc.total:
        REM Calculate total field by adding freight, tax, and line items.
        frt$ = "" \ tax$ = ""
        call cGetEditText( inv1HDLG%, 121, addrof(frt$)) \ frt = val(frt$)
        call cGetEditText( inv1HDLG%, 122, addrof(tax$)) \ tax = val(tax$)
        total = frt + tax
        for num% = 1 to 50
            REM Line items start at rec$(24) and have 7
            REM fields. Retrieve field #7:
            idx% = 24 + ((num% - 1) * 7) + 6
            total = total + val(rec$(idx%))
```

```
        next num%
        total$ = str$(total)
        return
date.format:
        REM Format date fields.
        date.fld$ = ""
        call cGetEditText( inv1HDLG%, cmdid%, addrof(date.fld$))
        date.fld$ = dte$(date.fld$)
        return
phone.format:
        REM Format phone fields.
        phone.fld$ = ""
        call cGetEditText( inv1HDLG%, cmdid%, addrof(phone.fld$))
        patt$ = "" \ pmap$ = ""
        for i% = 1 to 4
           for j% = 1 to 6
              if match(phone.fld$,phone.msk$(j%))
                 patt$ = phone.msk$(j%) \ pmap$ = phone.map$(j%)
                 j% = 6 \ i% = 4
                 ifend
              next j%
           next i%
        if patt$<>""
           ofmt$ = "(510) 000-0000"
           temp$ = ofmt$
           if patt$="#####" and pmap$[5:5]="%" then temp$=temp$[1:5]
              for i% = 1 to len(patt$)
                  chr.pos% = asc(pmap$[i%:i%])-32
                  temp$[chr.pos%:chr.pos%] = phone.fld$[i%:i%]
                  next i%
              phone.fld$ = temp$
              ifend
        return
```

The Source File for the Dialog

When the operator selects the "Invoice Line Items" button on the form view, the program will perform a "call line" statement to display a dialog to get the line items for the invoice.



The source code for this dialog is stored in the **dlgline.b** file. The following code was added to the save areas to customize this file:

The first save area contains the following $GLOBAL statements used to pass information between the subroutines in the form view and the line item dialog. These are the same global variables that were declared in the **dlginv.b** file.

```
REM@# Dialog line V2.3
REM@# 1ststart
$GLOBAL mat rec$          \ REM the invoice record array
$GLOBAL docModified%      \ REM used to check changes in record
$GLOBAL newFlag%          \ REM used in the line item dialog
$GLOBAL dataEntered%      \ REM used in the line item dialog
REM@# 1stend
#include "dlgline.IDH"
```

$SUB line

The save area in the main subroutine is used to dimension an array to hold 50 line items.

```
REM Dimension the list box strings array to hold 50 items.
dim LineItems$(50)
```

$SUB lineBtnClick

The following code has been added to the save area to handle the Update, Cancel and Return button selections:

```
select CMDID%
case 200
    REM Update button
    REM Update means to update the current line item with the data in
    REM the edit controls. We will always have a valid current line
    REM number as the Update button will be disabled until a line item
    REM has been selected.
    LineItems$(CurrentItem%) = fn.GetFromEdit$( CurrentItem% )
    REM Set the global doc modified flag so that the invoice form can
    REM detect that the invoice line items have changed.
        docModified% = 1
    REM Update the line item list box string by deleting the current
    REM string and inserting the new one.
    call cLBoxDelString( lineHDLG%, 300, CurrentItem% )
    call cLBoxInsString( lineHDLG%, 300, LineItems$(CurrentItem%),
CurrentItem% )
    REM Clear the edit controls, disable the Update and Cancel
    REM buttons and set the focus back to the list box.
    call cSetEditText( lineHDLG%, 100, "" )
    call cSetEditText( lineHDLG%, 101, "" )
    call cSetEditText( lineHDLG%, 102, "" )
    call cSetEditText( lineHDLG%, 106, "" )
    call cSetEditText( lineHDLG%, 107, "" )
    call cEnableCtrl( lineHDLG%, 200, 0 )
    call cEnableCtrl( lineHDLG%, 201, 0 )
    call cSetFocus( lineHDLG%, 300 )
    $exit
case 201
    REM Cancel button
    REM Clear the edit controls, disable the Update and Cancel
    REM buttons and set the focus back to the list box.
    call cSetEditText( lineHDLG%, 100, "" )
    call cSetEditText( lineHDLG%, 101, "" )
    call cSetEditText( lineHDLG%, 102, "" )
    call cSetEditText( lineHDLG%, 106, "" )
    call cSetEditText( lineHDLG%, 107, "" )
    call cEnableCtrl( lineHDLG%, 200, 0 )
    call cEnableCtrl( lineHDLG%, 201, 0 )
    call cSetFocus( lineHDLG%, 300 )
    $exit
case 203
    REM Return button
```

```
        REM If this is a new record, set a flag to indicate that data has
        REM already been entered on the line item screen. This is done so
        REM the program will know that it's been here before in case we
        REM return to the line item dialog.
        if newFlag%<>0 and dataEntered%=0 then dataEntered% = -1
cend
```

$SUB lineEvent

The only code that has been added to this subroutine is used to detect that an item in the list box has been selected. The code to process the selection is found in the save area at the end of the file.

```
case 2
    REM Process a selection from the list box for the combo control.
    GOSUB select.an.item
```

$SUB lineInit

Code has been added to this subroutine to initialize the list box with any existing line items. The Update and Cancel buttons are disabled when there are no items.

```
REM Display the line items in the list box. There are 50 line items with
REM 7 fields/item starting at rec$(24).
for item% = 1 to 50
    LineItems$(item%) = fn.GetFromRec$( item% )
    next item%
call cAddLBoxContents( lineHDLG%, 300, mataddrof(LineItems$) )
REM First time into dialog for a new invoice, the first item is selected
REM automatically. Subsequent items must be selected manually.
if newFlag%<>0 and dataEntered%=0
   call cLBoxSetSel( lineHDLG%, 300, 1)
   call cLBoxTopIndex( lineHDLG%, 300, 1)
   gosub select.an.item
else
   REM Disable the Update and Cancel buttons.
   call cEnableCtrl( lineHDLG%, 200, 0 )
   call cEnableCtrl( lineHDLG%, 201, 0 )
ifend
```

The following code has been entered between the usrstart and usrend remarks at the end of the file:

```
REM The format of the list box string is line item #, serial number, REM
product, description, qty, cost, and total.  This function that gets a
REM item from the record array and formats it as a list box string takes
REM one argument, the line item number.
def fn.GetFromRec$( num% )
```

```
    REM line items start at rec(24) and have 7 fields
    idx% = 24 + ((num% - 1) * 7)
    fn.GetFromRec$ = RPAD$(rec$(idx%),4) + ~
                RPAD$(rec$(idx%+1),12) + ~
                RPAD$(rec$(idx%+2),10) + ~
                RPAD$(rec$(idx%+3),20) + ~
                RPAD$(rec$(idx%+4),6) + ~
                FORMAT(val(rec$(idx%+5)), "#######.##") + ~
                FORMAT(val(rec$(idx%+6)), "#######.##")
fnend

REM The function to get a line item from the edit controls, update the
REM record array and format it as a list box also takes one argument,
REM the line item number to get.
def fn.GetFromEdit$( num% )
   REM Line items start at rec(24) and have 7 fields.
   idx% = 24 + ((num% - 1) * 7)
   REM Copy the edit control contents to the record array.
   REM The line item number (field 1) never changes.
   call cGetEditText( lineHDLG%, 100, ADDROF(rec$(idx%+1)) )
   call cGetEditText( lineHDLG%, 101, ADDROF(rec$(idx%+2)) )
   call cGetEditText( lineHDLG%, 102, ADDROF(rec$(idx%+3)) )
   call cGetEditText( lineHDLG%, 106, ADDROF(rec$(idx%+4)) )
   call cGetEditText( lineHDLG%, 107, ADDROF(rec$(idx%+5)) )
   REM Field 7 (Total) is calculated (Qty * Cost)
   rec$(idx%+6) = STR$(VAL(rec$(idx%+4)) * VAL(rec$(idx%+5)))
   REM Format a list box string with the results.
   fn.GetFromEdit$ = RPAD$(rec$(idx%),4) + ~
               RPAD$(rec$(idx%+1),12) + ~
               RPAD$(rec$(idx%+2),10) + ~
               RPAD$(rec$(idx%+3),20) + ~
               RPAD$(rec$(idx%+4),6) + ~
               FORMAT(val(rec$(idx%+5)), "#######.##") + ~
               FORMAT(val(rec$(idx%+6)), "#######.##")
fnend
select.an.item:
   REM Get the index of the selected item.
   call cGetLBoxSelection( lineHDLG%, 300, mataddrof
(lineITEMS%))
   CurrentItem% = lineITEMS%(2)
   REM Break the string into its respective fields and set the edit
   REM controls.
   call cSetEditText( lineHDLG%, 100, TRIM$(LineItems$
(CurrentItem%)[ 5:16]) )
   call cSetEditText( lineHDLG%, 101, TRIM$(LineItems$
(CurrentItem%)[17:26]) )
```

```
   call cSetEditText( lineHDLG%, 102, TRIM$(LineItems$
(CurrentItem%)[27:46]) )
   call cSetEditText( lineHDLG%, 106, TRIM$(LineItems$
(CurrentItem%)[47:52]) )
   call cSetEditText( lineHDLG%, 107, TRIM$(LineItems$
(CurrentItem%)[53:62]) )
   REM Set the focus to the first edit control.
   call cSetFocus( lineHDLG%, 100 )
   REM Enable the Update and Cancel buttons.
   call cEnableCtrl( lineHDLG%, 200, 1 )
   call cEnableCtrl( lineHDLG%, 201, 1 )
return
```

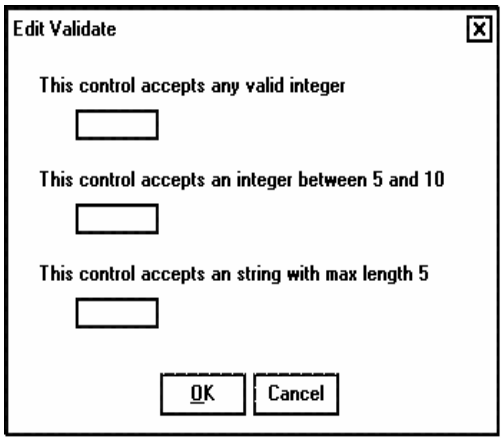
VALIDATE - A Sample Data Validation Routine

This sample dialog has been provided to illustrate how you may validate the data entered into an edit control. Three edit controls are used to accept an integer, an integer in a specific range and a string of five characters or less. The type of field that may be entered is determined by the value of the first user-defined property for the control.

Two command buttons are used. The Cancel button will cancel the dialog and quit the program. The OK button will close the dialog, display the user's input (defined as $GLOBAL) and call the dialog again.

The following source code for the $MAIN program is stored in **valmain.b**.

```
$MAIN
$GLOBAL ValENDBTN%
$GLOBAL ValEDIT100$
$GLOBAL ValEDIT101$
$GLOBAL ValEDIT102$
again:
call Val
if ValENDBTN% = 201 \  rem the Cancel button
      call cForceExit
      quit
else
      print ValEDIT100$
      print ValEDIT101$
      print ValEDIT102$
ifend
goto again
```



The VALIDATE program may be run from the command line or from W32APP after loading the project file (**validate.prj**). The program will display the dialog shown here.

The source code for the dialog is stored in the file called **val.b**. Only two of the BASIC subroutines have been modified (ValEvent and ValInit). This code has been heavily remarked and included in this manual for your convenience.

Note that the data *type* is defined in the first user-defined property for the control. Two user-defined functions are used here: fn.Parse$ to determine the index for the CMDID% into the ValPROP$ array and fn.FindIdx% to read the data type from ValPROP$. These functions are stored in the last save area in ValInit subroutine.

```
$SUB ValEvent
SELECT CET_EVTYPE%
REM@# evtstart
case 7
    REM If it's the Cancel button or NOT an Edit control or the next
    REM control to receive the focus is 0, then ignore the Lost Focus
    REM event - otherwise validate the Edit field.
    if cet_evparm1%=201 OR (cmdid%/100)<>1 OR cet_evparm1%=0
        $exit
    ifend
    REM Get the edit field and get and parse the control type string
    REM defined in the first user-defined property for the control.
    call cGetCtrlValue( ValHDLG%, CMDID%, ADDROF(val$) )
    select fn.Parse$( ValPROPS$( fn.FindIdx%( CMDID% ), 1 ) )
    case "INT"
        REM Validate as an integer.
        if val$ = ""  OR  NOT NBR(val$)
            msgbox "Please enter an integer"
            goto bad
        ifend
        $exit
    case "INTRANGE"
        REM Validate as an integer range (min,max)
        msg$="Please enter an integer between "+min$+" and "+max$
        if val$ = ""  OR  NOT NBR(val$)
            msgbox msg$
            goto bad
        else if VAL(val$) < VAL(min$)  OR  VAL(val$) > VAL(max$)
```

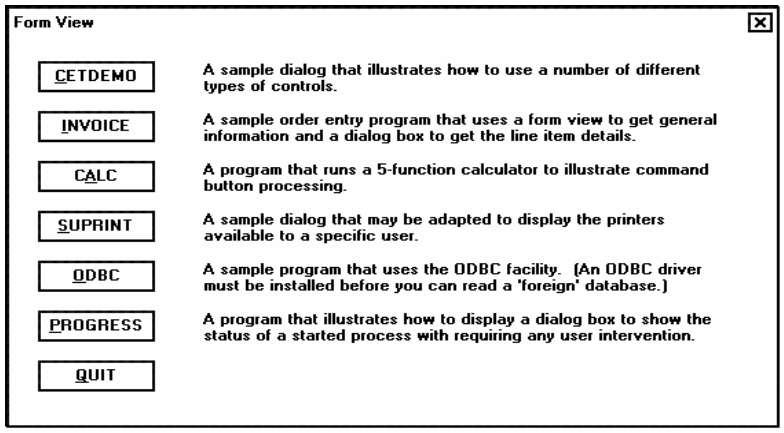
```
                msgbox msg$
                goto bad
                ifend
            ifend
            $exit
        case "STRINGLEN"
            REM Validate as a string of max characters.
            msg$="Please enter a string of "+max$ + " characters or less"
            if val$ = ""  OR  LEN(val$) > VAL(max$)
                msgbox msg$
                goto bad
            ifend
            $exit
        otherwise
            $exit
        cend
bad:
        REM If an entry is determined to be incorrect, focus is set back to
        REM the field for re-entry.
        call cSetFocus( ValHDLG%, CMDID% )
        $exit
REM@# evtend
CEND
$EXIT

$SUB ValInit
REM     Local function/gosub save area
REM@# usrstart
REM Find the index of the control with ID=idx%
def fn.FindIdx%( idx% )
for i% = 1 to ValCTRLS.CNT%
        if ValCTRLS%(i%) = idx%
            fn.FindIdx% = i%
            goto fn1end
        ifend
next i%
msgbox "Control "+STR$(idx%)+" was not found!"
fn.FindIdx% = 0
fn1end:
fnend

REM Parse the control type string into its substrings.
REM We use a "brute force" method as all of the validation
REM occurs during user interface code where blinding speed
REM is not important. (Remember that processors are rated in
REM millions of instructions per second.)
def fn.Parse$( src$  )
```

```
aa$ = src$
pos% = SCH( 1, src$, ":" )
if pos% = 0
      type$ = src$
else
      type$ = LEFT$( src$, pos% - 1 )
ifend
select type$
case "INTRANGE"
      REM format is: "INTRANGE:min,max" with no embedded spaces
      pos% = SCH( 1, src$, ":" )
      src$ = RIGHT$( src$, pos% + 1 )
      pos% = SCH( 1, src$, "," )
      min$ = LEFT$( src$, pos% - 1 )
      max$ = RIGHT$( src$, pos% + 1 )
case "STRINGLEN"
      REM format is: "STRINGLEN:max" with no embedded spaces
      pos% = SCH( 1, src$, ":" )
      max$ = RIGHT$( src$, pos% + 1 )
case "INT"
      REM nothing else to do
otherwise
      msgbox "Unknown control type: " + src$
      type$ = "UNKNOWN"
cend
fn.Parse$ = type$
fnend
REM@# usrend
```

W32DEMOS - A Sample Menu Program

The W32DEMOS program has been provided to make it easier to run the W/32 sample programs. It also illustrates how simple it is to create menus that use command buttons to execute the various options. Each command button has a corresponding text control describing the menu item.

The following menu is displayed:



The code for the $MAIN program stored in **w32demos.b** is as follows. Since no information needs to be shared, no $GLOBALs were defined.

```
$main
call cAppTitle("W/32 App Builder Demo Programs")
call cForceExit
call w32dlg
quit
```

The w32dlgBtnClick subroutine was modified to process the command button selections. Note that each time a button was clicked to run one of the sample programs, a call is made to minimize the application window before the desired process is started. The cCreateProcess function was used to start the sample programs so we could determine when the user quit the program (and terminated the process) and the application window should be restored (in the STATUS routine.

```
$SUB w32dlgBtnClick
REM     Button Click save area
REM@# btnstart
select cmdid%
case 201
      call cAppWindowMin
      call cCreateProcess(addrof(hdle%),".\cetdemo",3)
      gosub status
case 202
      call cAppWindowMin
      call cCreateProcess(addrof(hdle%),".\invoices",3)
      gosub status
case 203
      call cAppWindowMin
      call cCreateProcess(addrof(hdle%),".\calc",3)
      gosub status
case 204
      call cAppWindowMin
```

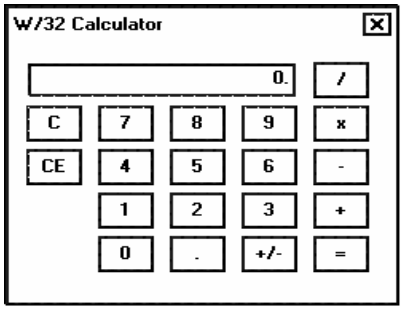
```
        call cCreateProcess(addrof(hdle%),".\suprint",3)
        gosub status
    case 205
        call cAppWindowMin
        call cCreateProcess(addrof(hdle%),".\odbc",3)
        gosub status
    case 206
        call cAppWindowMin
        call cCreateProcess(addrof(hdle%),".\progress",3)
        gosub status
        cend
    REM@# btnend
    GOSUB w32dlg.BTNEND
    $EXIT
```

The STATUS routine is stored in the last save area in the w32dlgInit subroutine. Please refer to the *W/32 Window Functions* chapter for more information on using the cCreateProcess and cProcessStat functions.

```
    $SUB w32dlgInit
    REM     Local function/gosub save area
    REM@# usrstart
    status:
        if hdle%=0
            msgbox "Program not started"
            call cAppWindowNormal
        ifend
        stat%=998
        while stat%=998
            sleep .25
            call cProcessStat(hdle%,addrof(stat%))
        wend
        call cAppWindowNormal
    return
    REM@# usrend
```

CALC - A Program With Special Command Button Processing

This sample program displays a 5-function calculator. Each key on the calculator is a command button. Special modifications have been made so that the program works as expected.



At the end of the calcBtnClick subroutine a call to cSetCtrlValue is made to update the display each time a button is clicked - just like on a real calculator.

Code has been added to the user save area in the calcInit subroutine to position the cursor to the end of the edit control. As you would expect, the data is right-justified.

The code in the $MAIN program is quite simple:

```
$main
call calc
call cForceExit
quit
```

The source code that was modified for the dialog **calc.b** has been included here for your convenience. Note the ON ERROR GOTO statement at the top of the calcBtnClick subroutine. Since each BASIC subroutine handles it own errors, similar statements should be used at the beginning of every subroutine where an error may occur. (This is the only place in this example.) The error handling routine is stored in the user save area at the end of the program.

```
$SUB calcBtnClick
REM     Button Click save area
REM@# btnstart
on error goto calcerr
call cGetCtrlValue( calcHDLG%, 100, ADDROF(value$) )
value = VAL(value$)
key% = CMDID% - 200
select
case key% <= 9
      REM a numeric key
      if state$ = "Clear" OR state$ = "Result1"
```

```
            disp$ = str$(key%)
            state$ = "Append"
      else if state$ = "Append"
            disp$ = value$ + str$(key%)
      ifend
      ifend
case key% = 10
      REM a decimal point
      if state$ = "Clear" OR state$ = "Result1"
            disp$ = "0."
            state$ = "Append"
      else if state$ = "Append"
            disp$ = value$ + "."
      ifend
      ifend
case key% = 11
      REM the change sign key
      value = 0 - value
      disp$ = str$(value)
case key% = 12
      REM the equal key
      select cmd%
      case 13
            result = num1 + value
      case 14
            result = num1 - value
      case 15
            result = num1 * value
      case 16
            result = num1 / value
      cend
      disp$ = str$(result)
      state$ = "Clear"
      cmd% = 0
case key% <= 16
      REM a command key
      num1 = value
      state$ = "Result1"
      disp$ = value$
      cmd% = key%
case key% = 17
      REM the Clear Entry key
      disp$ = ""
case key% = 18
      REM the Clear all key
      disp$ = "0."  \  state$ = "Clear"
```

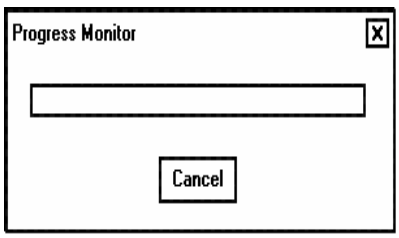
```
otherwise
      MsgBox "Calc: unknown button"
cend
REM Update the display each time a button is pressed.
call cSetCtrlValue( calcHDLG%, 100, disp$ )
REM@# btnend
GOSUB calc.BTNEND
$EXIT
```

The following modifications have been made so that the cursor is initially set to the end of the edit control. The initial value of "0" is displayed right-justified as it would be on a real calculator. The error routine is stored in the last save area.

```
$SUB calcInit
REM     Initialize dialog save area
REM@# inistart
state$ = "Clear"
REM set the focus to the edit control and remove the selection
call cSetFocus( calcHDLG%, 100 )
call cSetCaret( calcHDLG%, 100, -1 )
REM@# iniend
$EXIT

REM@# usrstart
calcerr:
MsgBox "Calc: Error err="+str$(err)+" line="+str$(erl)
quit
REM@# usrend
```

PROGRESS - A Sample Dialog With a Progress Bar

The PROGRESS program illustrates how a button click may be sent to activate a dialog without any user interaction. In this case, the dialog displays a progress bar with solid black bars while the started process is still active or until the user clicks on the Cancel button to close the dialog (but not the started process).



The source code for the $MAIN program stored in **prgmain.b** is a follows:

```
$MAIN
call prgmon
msgbox "done"
```

```
call cForceExit
quit
```

The prgmonBtnClick subroutine in the dialog source file **prgmon.b** has been modified to detect a special *button* selection with an ID of 299 (sent from the prgmonInit subroutine). If detected, a call to cCreateProcess is made to start the W/32 CALC program. While CALC is active, a progress bar will be displayed. Note that CHR$(135) was used to display the character on the bar, but any nonprintable character could have been used with the same affect.

Note that a call to cEndDlg is made to *automatically* end the dialog when the process ends. This operation is usually done by setting the End Dialog property for the command button. Since we did not want to alter the normal Cancel operation, we chose to use the call to cEndDlg instead.

```
$SUB prgmonBtnClick
REM@# btnstart
REM Check for our "special" button ID.
if CMDID% = 299
    REM You can also start the new process in your $MAIN
    REM and make hndl% a global variable.
    call cCreateProcess( ADDROF(hndl%), "calc.exe", 1 )
    progress$ = CHR$(135)  \  stat% = 998
    while stat% = 998
        call cSetCtrlValue( prgmonHDLG%, 100, progress$ )
        if  LEN(progress$)<40  then  progress=progress$+CHR$(135)
else progress$=CHR$(135)
        sleep .5
        call cProcessStat( hndl%, ADDROF(stat%) )
    wend
    REM We are done so end the dialog…
    call cEndDlg( prgmonHDLG%, CMDID% )
ifend
REM@# btnend
GOSUB prgmon.BTNEND
$EXIT
```

The button click to activate the progress bar is sent from the user save area in the prgmonInit subroutine. We chose to send an ID value of 299 which is in the valid range for command buttons, but it is not the ID value for a *real* button.

```
$SUB prgmonInit
REM@# inistart
REM Send an initial button click to start the monitor, a value in the
REM button ID range (200-299) that is NOT the ID of a real button.
call cSendBtnClick( prgmonHDLG%, 299 )
REM@# iniend
```

$EXIT

CHAPTER 8

W/32 Window Functions

Introduction

The W/32 Application Builder provides you with a wide variety of C callable functions that enable you to perform operations which would be difficult or impossible to code using BASIC alone. This chapter covers the functions you may use to implement *standard* Windows features within your application.

Information on the special functions available for use in forms and dialog boxes is provided in the *W/32 Dialog Editor* chapter. The functions that may be used to get information about the environment or specific files are covered in the *Built-in C Language Routines* chapter.

The W/32 functions have been described here in alphabetical order for your convenience. A chart is included at the end this chapter to give you an overview of the functions that are available.

Using C Language Functions

The CALL statement is used to transfer control to an external C, BASIC or assembly language routine. The names of the routines **must** be in exactly the same case mode as shown in this manual in order for the linker to resolve the reference during compilation.

There are two modes to the CALL statement. Mode 1 uses the syntax "CALL *routine"* to transfer control to a specific routine. No information is passed as arguments to the routine or back to the calling program. A few of the W/32 routines such as cFontValues use this syntax.

The majority of the C routines use mode 2 of the CALL statement to pass one or more arguments to the routine using the syntax "CALL *routine* (*arg-list*)". When

using this mode, it is important to pass the correct type and number of arguments in the expected order. Otherwise, a runtime error will be detected.

For example, if a routine such as cDIBWindow expects an integer value (handle%), you must pass it an integer variable or constant and not a floating point.

The special ADDROF function is used to pass the memory address of a variable to a C routine rather than the value itself. This variable may be an element of an array or a simple variable. MATADDROF is a similar function that is used to pass the memory address of an array.

The C routine will evaluate the address of the variable and assign it a new value. For example, in the following statement the ADDROF two integer variables are passed to cGetCursor. The values assigned to the variables row% and column% will then be passed back to the BASIC program.

```
call cGetCursor(ADDROF(row%), ADDROF(column%))
```

For more information on the CALL statement, refer to the *CET BASIC Language Reference Manual*.

cAppTitle

Since W/32 programs will not display a default title in the main application window, this function should be used to display some text such as the program name. (Actually a dash will appear only because it cannot be suppressed.)

The syntax of the CALL statement is:

```
call cAppTitle( title$ )
```

where:

title

specifies the text to be used. Take care that the text does not exceed the width of the window as any excess characters will be truncated.

Note that this function may also be called from the cetPreInit subroutine in a copy of **cetwin.b** when you want to change the title for all of the programs you compile using that particular resource file. Refer to the *Windows Framework* chapter for information on how this may be done.

cAppWindowInfo

This routine may be used to get the location and size of the main application window. The syntax of the CALL statement is:

```
call cAppWindowInfo(addrof(x%),addrof(y%),addrof(w%),addrof(h%))
```

where:

x, y

returns the window's upper-left column and row position.

w, h

returns the width and height of the window.

Refer to the cSizeAppWindow function if you need to set the window size.

cAppWindowMax

This routine may be used to maximize the main application window after a call has been made to cAppWindowMin. There are no arguments passed to the routine. The syntax is:

```
call cAppWindowMax
```

cAppWindowMin

This routine may be used to minimize the main application window. There are no arguments passed to the routine. The syntax is:

```
call cAppWindowMin
```

cAppWindowNormal

This routine may be used to restore the main application window to its normal size and position (i.e. before the call to cAppWindowMin or cAppWindowMax. No arguments are passed to the routine. The syntax is:

```
call cAppWindowMax
```

cAppWindowTop

This routine may be used to bring the application window to the top of the window stack. There are no arguments to be passed. The syntax is:

```
call cAppWindowTop
```

cCheckItem

The cCheckItem routine may be used to enter or remove a check mark next to a menu item to indicate its current state. For example, if a menu item represents something that has only two states such as on/off or enabled/disabled, this feature may be used to place a check mark next to an item that is active.

The syntax of the statement is:

call cCheckItem(item%, state%)

where:

item

is the ID value (an integer 1 through 250) associated with the menu item. For example, the ID value for ID_WINSUB1 is 1 and 250 for ID_WINSUB250.

state

is an integer value, either 1 for checked or 0 for unchecked.

For example:

```
REM  Enter a check mark next to the menu item associated with ID_WINSUB4.
call cCheckItem(4, 1)
```

cChoiceList

This routine will display a scrolling choice list dialog window and allow a user to either select an item (and press the OK button) or double-click on an item.

The items will be automatically sorted before the window is displayed. Although you can have up to 15,000 items on the list, the practical limit is about 2,000 (or less depending on the speed of the computer and the resources available) since it takes a while to sort the items. (See the cChoiceListNS function.)

The syntax of the CALL statement is:

call cChoiceList(title$, mataddrof(items$), addrof(choice$))

where:

title

specifies the string to be displayed as the title of the window.

items

specifies the array of strings to be displayed in the choice list. Note that the function will display a list of items up to the dimensioned size of the array or until an empty array element is encountered.

choice

is the string selected from the choice list. A null string will be returned if the operator left the choice list by selecting Cancel.

For example, to input a number of elements and create a choice list:

```
INPUT j% \ DIM item$(j%)
FOR i%=1 TO j%
      item$(i%) = "line"+STR$(i%)
      NEXT I%
title$ =STR$(j%)+" Item Choice List"
CALL cChoiceList(title$, MATADDROF(item$), ADDROF(choice$))
PRINT "Your selection was: ";choice$
CLEAR item$
```

cChoiceListNS

This routine operates similar to cChoiceList and displays a scrolling choice list dialog. The one difference is that this routine will not sort the items before they are displayed.

For greater flexibility and speed, we recommend that you use this routine instead of cChoiceList to display a list of items that has already been sorted by your program.

The syntax of the CALL statement is:

call cChoiceListNS(title$, mataddrof(items$), addrof(choice$))

Please refer to cChoiceList for information on usage.

cCloseDIBWindow

This routine should be used to close a DIB window used to display an image file in a bitmap format. The syntax of the CALL statement is:

call cCloseDIBWindow(handle%)

where:

handle

is the handle for the DIB window opened with a call to cOpenDIBWindow.

Note that attempting to close a DIB window using an invalid handle will generate an error message.

cCreateProcess

This function may be used to start a process. Under Windows NT and Windows 95, you can start any type of process (i.e. a DOS process, a 16-bit Windows app or a 32-bit Windows app). Under other Windows platforms, you can only use cCreateProcess with a 32-bit Windows application, otherwise the return value will indicate that the process was not started when it actually was.

The syntax of the statement is:

call cCreateProcess(addrof(handle%), cmd$, show%)

handle

is the handle returned for the process. A value of zero means that the process was not started.

cmd

is the name of the executable including any required arguments. The search sequence is as follows:

the directory where the calling program was executed
the current working directory
the \windows or \windows\system directory
the directories set in the PATH variable
the directories set in the B_LKPATH variable

Note that this argument only affects the executable. Windows programs will normally only search the current working directory for a file unless a complete pathname is specified.

show

is the Windows show state where 1 indicates a normal window, 2 is a minimized window and 3 is a maximized or full-screen window.

This argument will have no effect when starting a DOS application. In that case, the full screen will be used unless a .**pif** file has been defined with a windowed property.

Refer to the cProcessStat function to see how to query the status of a process after it has been started. (Refer to the W32DEMOS program for a *real* example.)

cDIBInfo

This routine may be used to get the height and width of a DIB image. The syntax of the CALL statement is:

```
call cDIBInfo( name$, addrof(w%), addrof(h%) )
```

where:

name

specifies the full path name of the DIB file to be read. The image file must be in a Microsoft Windows bitmap format.

w

returns the width of the DIB image.

h

returns the height of the DIB image.

cDIBWindowInfo

This routine may be used to get the location and size of a DIB window. The syntax of the CALL statement is:

```
call cDIBWindowInfo(hdle%,addrof(x%),addrof(y%),addrof(w%),addrof(h%))
```

where:

hdle

specifies the handle for the DIB window you need to get information about. This must be the same handle used in the call to cOpenDIBWindow.

x, y

returns the upper-left column and row position for the specified window.

w, h

returns the width and height of the specified window.

cDIBWindowTop

This routine may be used to bring the specified DIB window to the top of the window stack. The syntax of the CALL statement is:

```
call cDIBWindowTop(handle%)
```

where:

handle

specifies the handle for the DIB window to be displayed on top.

Refer to the cOpenDIBWindow function for information on using DIB windows to display an image file in a Microsoft Windows bitmap format.

cDisplayInfo

This routine may be used to get the screen resolution and number of colors. This feature is useful when you need to verify that the screen resolution for the device is high enough to display text in a specific font.

The syntax of the CALL statement is:

call cDisplayInfo(addrof(x%), addrof(y%), addrof(colors%))

where:

x, y

is the vertical and horizontal resolution for the device in use (e.g. 640x480).

colors

is the number of colors available. This value is returned as a number of bits per pixel. A value of 4 indicates that 16 colors are available, and 8 indicates that there are 256. A value of 1 is returned for a monochrome monitor.

Note that the return value for color is dependent upon the driver on the particular video adapter in use. We have found that sometimes a value of 1 is returned for a color monitor.

cEnableDocking

This routine may be used to enable a *dockable* toolbar that may be dragged off the application window and displayed in a window of its own. The syntax of the statement is:

call cEnableDocking

When docking is enabled, there will be a vertical black line to the right of the last button. To drag the toolbar, the operator must move the mouse into the background area between the buttons and the black line. To reconnect the toolbar, just drag it back to the top or bottom and release it (the dragging outline will change to a solid line when it is over a docking site).

Note that this function only has an effect in the current program. The toolbar will always be returned to its default position after a CHAIN or LINK.

cEnableItem

The cEnableItem routine may be used to enable or disable an ID_WINSUB menu item. This feature is especially useful when you want to ensure that the operator will not select an item that could interrupt a particular operation and cause problems. The syntax of the statement is:

call cEnableItem(item%, state%)

where:

item

is the ID value (1 through 250) associated with the menu item. For example, the ID value would be 1 for ID_WINSUB1 and 250 for ID_WINSUB250.

state

is an integer value, either 1 for enabled or 0 for disabled.

For example:

```
REM  Disable the menu item associated with ID_WINSUB4.
call cEnableItem(4, 0)
```

By default, all menu items are enabled so that they may be selected. Disabling a menu item turns the text gray and makes it inaccessible to the user.

Refer to the *Windows Framework* chapter for information on how to modify the default menu bar and/or create new menu items.

cEnableMainWin

The cEnableMainWin routine may be used to control the activation state of the main application window. This feature is useful if you plan on moving the main window and want it to be invisible until it is in its final position (with calls to **cShowMainWin**). During that process, you would need to disable any mouse or keyboard input. The syntax of the statement is:

call cEnableMainWin(state%)

state

is an integer value, either 1 to enable the main window (mouse and keyboard input) or 0 to disable the window.

cEnumPrinters

This routine will provide you with information about the available printers. Normally, this printer information would then be used to set the B_PRINTER*n* variable with a call to the Bputenv function. (Refer to the *Implementation Notes* in the Appendix for information on printing.)

The syntax of the CALL statement is:

call cEnumPrinters(state%,addrof(prt$),addrof(drvname$),addrof(port$))

where:

state

is the state where zero will start the specification process, 1 will continue and 2 will free the dynamic memory associated with the enumeration process.

prt

is the string used to return the name of the printer which will be used to set B_PRINTER*n*. Under Windows NT and 95, this is the printer name. Under Windows, it is the name of the driver.

drvname

is the string used to return the driver name under NT and 95 or the name of the file that the driver uses under Windows. In the latter case, the filename is returned for your information only. It is not required by Print Manager.

port

is the string used to return the port that the printer is connected to. Under Windows NT and 95, the port number is not important.

Note that the last three parameters will be empty when all the printers have been specified.

Currently, the cEnumPrinters function may only be used to get information about the printers recognized by Print Manager. Although there may be more printers available, such as the network printers which are not connected in Print Manager, Windows does not provide a way to determine if a printer is local or networked. Under NT and 95, the name of a networked printer usually starts with "\\" as the network pathname to the printer so it may be possible to implement this feature in a later release.

The following code segment illustrates how this function may be used. Also note that one of the W/32 sample programs uses this feature. A project (**suprint.prj**) has been created for the SUPRINT program so that it may be run from within the W/32 Integrated Development Environment.

```
REM start enumerating printers
call cEnumPrinters(0,addrof(prt$),addrof(drvname$),addrof(port$))
while prtname$ <> ""
      print prtname$, drvname$, port$
      call cEnumPrinters( 1, addrof(prt$), addrof(drvname$), addrof(port$) )
      wend
REM free the dynamic memory that is being used
call cEnumPrinters(2, addrof(prt$), addrof(drvname$), addrof(port$))
```

cFontValues

This routine may be used to get the values to pass to cSelectFont in order to change the current font in a text-based window. No arguments are required.

call cFontValues

For example:

```
PRINT “Any text or line drawing characters you wish to see displayed”
LINPUT a$
CALL cFontValues
END
```

Run the compiled program and select the desired font from the Fonts menu. All of the values required by the cSelectFont routine for that font will be displayed in a message box. This procedure is covered in detail in the *Windows Framework* chapter.

cForceExit

This routine may be used to force a program to exit without displaying an OK message box. No arguments are required. The syntax of the CALL statement is:

call cForceExit

When an END statement is reached, the default behavior is to display a message box and wait for the OK button to be selected. Calling cForceExit prior to the END statement will immediately exit the program and close the application window.

cGetCursor

This function may be used to get the current cursor column and row position in screen character coordinates. The returned values will be relative to the current window. The syntax of the CALL statement is:

call cGetCursor(addrof(row%), addrof(column%))

row

returns the row or y coordinate.

column

returns the column or x coordinate.

cGetMouse

This function may be used to return the current mouse position in screen character coordinates relative to the window. The syntax of the statement is:

call cGetMouse(addrof(row%), addrof(column%))

row

returns the current row or y coordinate.

column

returns the current column or x coordinate.

cInputQ

This function may be used to put a key code into the input queue or buffer just as if the key had been entered by the operator. The syntax of the statement is:

call cInputQ(key%)

key

is the ASCII value of the key to be placed into the input buffer.

For example:

```
REM place an ESC into the input queue
call cInputQ(27)
REM place an F1 key into the input queue
call cInputQ(0)
call cInputQ(59)
```

Note that two key codes must be placed into the buffer for a special key such as F1 (e.g. a zero, and the extended ASCII code for the F1 key). Any key that is not a printable character (e.g. function keys, cursor keys, numeric keypad keys, insert, home, etc.) is considered a special key.

The extended ASCII values are the same as those returned by the BASIC GET statement. Refer to the Appendix of the *CET BASIC Language Reference Manual* for a table of the character code values to use.

cLogin

This routine may be used to display a Login dialog box that prompts the operator to enter a user name and password. These values are returned for use within your application. The syntax of the CALL statement is:

call cLogin(addrof(user$), addrof(password$))

where:

user

is the specific user name entered by the operator.

password

is the user's password.

For security purposes, cLogin will display asterisks in the password field instead of the actual characters that are typed.

This feature is convenient when a program must create temporary work files in a multi-user environment. The following code segment illustrates how to create an indexed file with a unique name for each user. The file will be stored in the **\tmp** directory on drive Z on the remote server.

```
call cLogin(addrof(user$), addrof(password$))
REM  Depending upon the user name, the string may need to be padded or...
filename$ = "Z:\TMP\WORK"&user$
call Berase (filename$)
call Bcreate(filename$&"INDEXED RECLEN 100 KEYLEN 10")
OPEN #1: filename$, update indexed
```

This procedure will work equally as well with a sequential file. In that case, instead of using the Berase and Bcreate functions, use OPEN with the option OUTPUT to overwrite the file, or with the EXTEND option to append the new data.

cMaxSize

This routine may be used to resize a window to the maximum size needed to display 25 lines of text using the current font. If the window is already the maximum size, this function will have no effect. There are no arguments required. The syntax is:

call cMaxSize

cMessageBoxOkC

The cMessageBoxOkC routine is available so you may display a message box with both OK and Cancel buttons. The program will be suspended until the user makes a selection.

cMessageBoxOkC requires two arguments; a string of characters to be displayed as the message and an integer variable which is used to return the result.

The syntax of the statement is:

call cMessageBoxOkC(msg$, addrof(result%))

where:

msg

specifies the string of characters to be displayed.

result

is a value indicating which button was selected. The return value is 1 for the OK button and 0 for Cancel.

For example:

```
call cMessageBoxOkC( "OK to print the report?", addrof(result%))
IF result%
      THEN MSGBOX "Report is printing."
      ELSE
      MSGBOX "The print job was canceled."
IFEND
```

The MSGBOX statement may also be used to display a message box with an OK button. This statement is covered in the *Implementation Notes* in the Appendix.

cMoveAppWindow

This routine may be used to move the main application window to a new location. The syntax of the CALL statement is:

call cMoveAppWindow(x%, y%)

where:

x, y

specifies the new upper-left column and row position for the window.

For example:

```
REM move the upper left corner of the main application window to location 50,50
call cMoveAppWindow(50, 50)
```

cMoveDIBWindow

This routine may be used to move a DIB window to a new location. The syntax of the CALL statement is:

call cMoveDIBWindow(handle%, x%, y%)

where:

handle

is the window handle returned by the call to cOpenDIBWindow. Attempting to move a DIB window using an invalid handle will generate an error.

x, y

are the upper-left coordinates for the new window position.

For example:

```
REM Move the upper left corner of the window to screen location 50, 50
call cMoveDIBWindow(dibwin1%, 50, 50)
```

See the cOpenDIBWindow function for information on using DIB windows to display an image file in a Microsoft Windows bitmap format.

cNewMenu

This routine may be used to display a new menu. The syntax is:

call cNewMenu(menu%)

menu

is an integer value 1 through 50 that specifies the menu ID. For example, the ID value is 2 for IDR_MENU2, 50 for IDR_MENU50 and 1 for the default menu, IDR_MAINFRAME.

For example:

```
REM   Load the new menu associated with IDR_MENU5.
call cNewMenu(5)
REM   Return to the initial menu associated with IDR_MAINFRAME
call cNewMenu(1)
```

When you start your application, the initial menu associated with the identifier IDR_MAINFRAME is displayed. Multiple menus may be defined in the **.rc** file using the predefined identifiers IDR_MENU2 through IDR_MENU50. To load a new menu, call cNewMenu any time during program execution. To return to the initial menu, use another call with menu% = 1.

See the *Windows Framework* chapter for details on using drop-down menus.

cOnWin32s

The cOnWin32s routine may be used to determine if the current system is running under Win32s (Windows or Windows for Workgroups). The syntax is:

call cOnWin32s(addrof(flag%))

where:

flag

is equal to one for a system running under Win32s. A zero is returned when Windows 95 or NT is in use.

cOpenBrowser

This routine may be used to display a File-Open browser and return the user's selection to the program. The syntax of the statement is:

call cOpenBrowser(addrof(name$), filter$)

name

is the string used to return the full pathname of the selected file. This string should either be null or set to the name of the specific file to be used as the default value.

filter

is a string used to restrict the files that will appear on the list.

For example:

```
REM  Display an open browser and restrict the files to basic source files (*.b).
call cOpenBrowser( addrof(""), "Basic Files (*.b) | *.b ||" )
```

This function displays a standard Windows File-Open browser. If a file is selected, the full pathname to the file (including the drive code) is returned. If the user selects cancel, name$ will be null. When filter$ is a null string, every file will appear on the list as though the string *.* had been specified.

The filter string is formatted with two entries separated by a vertical bar and ending with two more vertical bars. The first entry is an informative message that describes the filter. The second entry should indicate the file extensions to use in the same way you would restrict file listings from a DOS DIR command. The second entry may have multiple extensions separated with a semicolon as:

```
"Source Files (*.b; *.asm) | *.b; *.asm ||"
```

You may also include multiple filters with each filter being accessible via a drop-down window. For example:

```
"Basic Files (*.b) | *.b | Assembler Files (*.asm) | *.asm ||"
```

The initial directory that is shown in the browser is the current working directory. If you want to use a different directory, change the directory with a call to the built-in function Bchdir prior to calling cOpenBrowser. A call to Bgetcwd may be made to retrieve the current working directory so that it can be reset after the browser function has returned. For example:

```
REM Save the current directory
call Bgetcwd( addrof(curdir$) )
REM Change to the directory you want the browser to start with
call Bchdir( "C:\MYAPP\SUBDIR" )
REM Set the default file name
name$ = "DEFAULT.TXT"
call cOpenBrowser( addrof(name$), "" )
REM Restore the current directory
call Bchdir( curdir$ )
```

cOpenDIBWindow

This routine may be used to open a new window to display a Microsoft Windows bitmap file in either a **.dib** or **.bmp** format. This is a powerful feature since it allows you to read an image file and display a picture or diagram without leaving your program. For example, if you have an insurance package, you might have a button on the toolbar that may be selected to display a picture of the insured item.

When a call to cOpenDIBWindow is made, the new window is inactive and the user must click on it to bring it to the front. (Make a call to cDIBWindowTop to move the window to the top without any user input.)

The bitmapped image will be sized to fit in the window. If the width and height arguments are zero, then the window will be made the same size as the bitmap. The syntax of the CALL statement is:

call cOpenDIBWindow(addrof(handle%),path$,title$,x%,y%,w%,h%)

where:

handle

is the handle for the DIB window. This handle should be used to reference the window in all other calls to DIB functions (e.g. cDIBWindowTop).

path

is the full path name of the **.dib** or **.bmp** file to be displayed.

title

is a string to be displayed as the title of the new window.

x, y

is the upper-left column and row position of the new window.

w, h

is the width and height of the window. If these values are zero, then the window will be made the same size as the bitmap.

For example:

```
REM  open \tmp\picture.bmp in a 256x256 size window at location 0,0 with the
REM  title "Graphics Window 1"
call cOpenDIBWindow(addrof(dibwin%),"\tmp\picture.bmp","Graphics Window 1",
0, 0, 256, 256 )
```

The value returned as the window handle must be used to reference the window in any of the other functions that operate on it.

Note: DIB windows that are not explicitly closed by the application will be closed at exit. They will not remain open across a LINK or CHAIN.

cProcessStat

This routine may be used to query the status of a process that has been started with a call to cCreateProcess. The syntax of the statement is:

call cProcessStat(handle%, addrof(stat%))

where:

handle

is the handle returned from cCreateProcess.

stat

is a value indicating the status of the process. Any value except for 998 or 999 indicates that the process has terminated. Otherwise:

998 = process is still active

999 = status can't be obtained because the process did not start or it is not a 32-bit app (under Windows)

Note that a W/32 app will, by default, quit with the exit code set to 0. You can set the exit status by using the QUIT statement with a numeric expression between 0 and 98. Except for the following, values greater than 98 will normally indicate that an error was detected.

100 = process terminated by executing a CHAIN/LINK/RUN statement

101 = program name passed to CHAIN/LINK/RUN could not be found

If your program needs to wait for a process to complete, perform a SLEEP otherwise, the Window will appear to be hung. (Under Windows the whole system may seem to be hung.) For example:

```
REM Create a process to be run minimized and obtain its return code
call cCreateProcess( addrof(hdl%), "\ats\glpost.exe", 1 )
if hdl% = 0
      print "process not started"
      goto NoStatus
      ifend
stat% = 998
while stat% = 998
      SLEEP .25  \ REM Set to a value that will use up processing time just to
check on the process.
      call cProcessStat( hdl%, addrof(stat%) )
```

```
        wend
print "Process exit code: ";stat%
NoStatus: ...
```

Refer to the W32DEMOS sample menu program and PROGRESS, the sample program that displays a program bar, for examples of how this feature may be used.

cSaveBrowser

This routine may be used to display a File-Save As browser and return the selected item to the calling program. The syntax is:

call cSaveBrowser(addrof(name$), ext$, filter$)

where:

name

is the string used to return the full pathname of the selected file. This string should either be null or set to the name of the specific file to be used as the default value.

ext

is the default extension to be appended to the filename. Specify a null string if no extension is to be used.

filter

is the filter to be used to select the files which will appear on the list. Specifying a null string will display every file.

For example:

```
REM  Display File-Save As browser. If the user does not supply a file extension,
REM  .b will be used, by default. Do not restrict files.
call cSaveBrowser( addrof(name$), "b", "" )
```

The only difference between this function and cOpenBrowser is that this browser uses "Save As" for the title and an optional parameter, the default extension, which will be appended to the file name if one is not supplied by the user.

cSelectFont

This function allows you to change the default font used in a text-based application. The syntax is:

call cSelectFont(name$,height%, weight%, italic%, charset%, family%)

name

is the name of the font to be used.

height

indicates the height of the font (logical font size). Note that height is related to the font point size, but it is not the same value.

weight

indicates the weight or style of the font.

italic

specifies the italics flag.

charset

specifies the font characteristics.

family

specifies the font family.

For the parameters to use for a given font, refer to the description of cFontValues. For example:

```
REM displays the Terminal 9-point font in Regular style.
call cSelectFont( "Terminal", -12, 400, 0, 255, 49 )
```

cSetCursor

This function may be used to set the cursor position for the next console output. The syntax of the CALL statement is:

call cSetCursor(row%, column%)

row

specifies the new row position for the cursor.

column

specifies the new column position for the cursor.

cShowMainWin

The cShowMainWin routine may be used to control the visibility of the main application window. This feature is useful if you plan on moving the main window and want it to be invisible until it is in its final position (with another call to cShowMainWin). During that process, you would need to disable any mouse or keyboard input with a call to cEnableMainWin.

The syntax of the statement is:

call cShowMainWin(state%)

where:

state

is an integer value, either 1 to make the main window visible or 0 to make it invisible.

Note that when a form view is invoked (with a call to the subroutine), it will always be visible. A call to cShowMainWin(0) in the last line of the dialog routine (after the call to cShowForm and before the $EXIT) will allow you to go from one form to another without the text-based window appearing in between. If the first thing your main application does is call a form view, then you should add the statement "call cShowMainWin(0)" in the cetPreInit subroutine in the **cetwin.b** file.

The main window may also be hidden prior to displaying a dialog. In this case, the dialog will be visible and centered on the invisible main window.

cShowStatusBar

This function may be used to display or hide the status bar. When the status bar is visible, the window will automatically be resized to accommodate the display. The syntax of the CALL statement is:

call cShowStatusBar(state%)

where:

state

is an integer value, either 0 to hide the status bar or 1 to display it.

You may wish to use this feature in all of your application programs especially if you have used the cStatusText function to display program specific messages in the status bar. In that case, we recommend that you call this function from the **cetwin.b** file. This way you will only need to make a call to cShowStatusBar if you want to hide the display.

cShowToolBar

This function may be used to display or hide the toolbar. When the toolbar is visible, the window will automatically be resized to accommodate the display. The syntax of the CALL statement is:

```
call cShowToolBar(state%)
```

where:

state

is an integer value, either 0 to hide the toolbar or 1 to display it.

If you want the toolbar to be visible when you start your application, we recommend that you call this function from within the **cetwin.b** file you are using. If you have created a new toolbar with a call to cToolBar, it will be automatically displayed. Otherwise, the default toolbar will be shown.

cShowVersion

This routine allows you to display the version of the W/32 Application Builder runtime components in a message box. No arguments are required. The syntax of the statement is:

```
call cShowVersion
```

The version of each of the runtime components (**cetcab.lib, cetb32.lib, cetcmpt.lib** and **brtm**.**exe**) will be displayed in separate windows. The last one to appear may be hidden by the application window since Brtm is a separate task. If you cannot see the message, move or minimize your application window.

cSizeAppWindow

This function may be used to set the main application window to a specific size. The syntax of the CALL statement is:

```
call cSizeAppWindow( x%, y% )
```

where:

x,y

is the new width and height parameters in pixels.

The easiest way to find out what values should be used is to create a dialog with the desired size and then look at the width and height properties. Remember that

the maximum size of the default, text-based window is the size required to display 80x25 characters using the current font.

cStatusText

This function may be used to replace the default message on the status bar. The syntax of the statement is:

call cStatusText(msg$)

msg

is the message string to be displayed.

For example:

```
call cStatusText("Enter Client Name")
```

When the status bar is first displayed, "Ready" appears, by default. Messages for the various menu items and toolbar buttons will also appear on the status bar. When the mouse cursor leaves the menu or tool bar, the default message is redisplayed.

At any time, you can change the default message with a call to cStatusText so that your program always displays an appropriate message for the user.

cStringInputQ

This function may be used to place a string into the input buffer just as though the characters were input from the keyboard. The syntax of the statement is:

call cStringInputQ(string$)

string

is the string of characters to be placed in the input queue.

For example:

```
call cStringInputQ( "Hello World" )
```

Refer to cInputQ for a similar function that allows you to insert characters such as the ESC key into the input buffer.

cToolBar

The default toolbar that comes with the W/32 Application Builder has two buttons as shortcut keys for the Fonts-Select and Help-About menu commands. This toolbar is defined with the following line in the Bitmap section of the **cetuser.rc** file:

```
IDR_MAINFRAME     BITMAP MOVEABLE PURE     "CETTOOL.BMP"
```

The button images are stored in the **cettool.bmp** file in the **\cetlib\inc** directory if you used the default path during installation. You may **not** change the default toolbar, but you can add new toolbars and call cToolBar to change the toolbar from within your application.

To design your own toolbar, copy the default file into your local directory with a new name. For example:

```
copy \cetlib\inc\cettool.bmp \tst\mytool2.bmp
```

Now, use the IMAGEDIT program to modify the file **mytool2.bmp**. A toolbar bitmap may have up to twenty buttons. Each button in the bitmap is 15 pixels high and 16 pixels wide, therefore the bitmap for a toolbar with two buttons would be 15 pixels high by 32 wide. Three buttons would be 15x48 and so on.

When you are done modifying the bitmap file, add a line in **cetuser.rc** to use your new toolbar.

```
IDR_TOOLBAR2      BITMAP MOVEABLE PURE     "MYTOOL2.BMP"
```

You can have up to 50 toolbars using the predefined identifiers IDR_TOOLBAR2 through IDR_TOOLBAR50 in the **.rc** file. The identifier IDR_TOOLBAR1 is reserved for the default toolbar called IDR_MAINFRAME.

After the new toolbar has been defined in the **cetuser.rc** file (and recompiled), a call to the cToolBar routine may be made to load the toolbar and associate the menu items with the buttons. (This may be done in the initialization routine in your BASIC program or in the cetPreInit subroutine in your Windows Framework file.)

The syntax of the CALL statement is:

```
call cToolBar( bmId%, num%, mataddrof(btns%))
```

where:

bmId

is the ID number of the bitmap resource. The default toolbar defined as IDR_MAINFRAME has a menu ID number of 1 (BmId%=1) and is associated with the file **cettool.bmp**. Use BmId%=2 through BmId%=50 to reference the toolbars defined as IDR_TOOLBAR2 through IDR_TOOLBAR50.

num

is the number of toolbar buttons including the SEPARATORs. This argument defines the number of elements there are in the btns% array.

btns

is an integer array with up to 250 elements that represents the menu ID values (ID_WINSUB1 through ID_WINSUB250) that are associated with the buttons. The first element of the array is always 1. If OPTION BASE 1 is in effect, element 0 will be ignored.

Since buttons are shortcut keys for specific menu items, each button (in the btns array) must be associated with an identifier in the same way MENUITEM is used to define the individual items associated with a menu.

A specific menu ID is associated with a button by placing it in the integer array corresponding to the button position in the bitmap file. For example, if a toolbar has three buttons, dimension the array to a size of three. The first button could be associated with menu item ID_WINSUB12, the second with ID_WINSUB2 and the third with ID_WINSUB34 by assigning the following values:

```
btns%(1) = 12 \ btns%(2) = 2 \ btns%(3) = 34
```

There are six special ID values that you may use. They are:

0	a SEPARATOR used to add space between buttons.
500	ID associated with Font Select menu (ID_FONTS_SELECT).
501	ID associated with Help About menu (ID_APP_ABOUT).
502	ID associated with File Exit menu (ID_APP_EXIT).
503	ID associated with View Toolbar menu (ID_VIEW_TOOLBAR).
504	ID associated with View Status Bar menu (ID_VIEW_STATUS_BAR).

The SEPARATOR does not consume a button, but adds space between buttons so they can be grouped together by function. If you use a SEPARATOR, increase the number of buttons (num%) by the number of SEPARATORs that are used since this argument defines the size of the btns% array.

The sample code below sets the new toolbar to have three buttons with a separator between button two and three.

```
REM  Set the toolbar to have 3 buttons with a separator between button 2 and
REM  button 3. The bmId% argument 2 is associated with IDR_TOOLBAR2
REM  which you must add to the .rc file and associate it with a .bmp file that REM
represents your buttons.
DIM btns%(4)
btns%(1)=12 \ btns%(2)=2 \ btns%(3)=0 \ btns%(4)=34
call cToolBar(2,4,mataddrof(btns%))
```

Please refer to the *Windows Framework* chapter for information on how to have the toolbar automatically appear each time you run your program.

cWaitCursor

This routine may be used to display an hourglass cursor. This feature is useful when you want to tell users to wait while another operation is being performed. The syntax is:

call cWaitCursor(wait%)

where:

wait

is an integer value, either 1 to display an hourglass cursor or -1 to remove it.

cWinExec

This routine may be used to execute a Windows application. It is important to note that this function does not pass control to the specified program (like the CSI statement does). As soon as the call to cWinExec is performed, the next statement in your program will be executed. Refer to the cCreateProcess and cProcessStat functions for a way to start a process and then check its status to determine when another operation should be performed.

The syntax of the CALL statement is:

call cWinExec(pathname$, show%)

where:

pathname

specifies the complete path name of the executable file. This argument is available so you may override the default search that uses the PATH variable. The search sequence is as follows:

the directory where the calling program was executed

the current working directory
the \windows or \windows\system directory
the directories set in the PATH variable
the directories set in the B_LKPATH variable

Note that this argument only affects the executable. Windows programs will normally only search the current working directory for a file unless a complete pathname is specified.

show

specifies how the Windows application is to be displayed. . The valid codes are:

1 - as a normal window
2 - as a minimized (icon) window
3 - as a maximized (full screen) window

For example:

```
call cWinExec( "notepad",1)
```

Once the application is launched, control returns immediately to the calling program. There is no option to wait for the launched application to complete. Refer to cCreateProcess and cProcessStat if this feature is required.

Since the CSH/CSI statement is designed to execute MS-DOS commands, you cannot launch a Windows application using CSI. (Actually you can under Windows NT, but not under Windows.)

Also note that you may use cWinExec to launch DOS executables under Windows NT, but the show argument operates differently. If a **.pif** file has been created with the windowed property set, then the values are:

1 - as a normal window
2 - as a minimized (icon) window
3 - as a normal window

The show argument will have no effect on a DOS program that is run from a **.bat** file.

Refer to the *Implementation Notes* section in the Appendix for information on using the CSI statement.

Function List

The following chart lists the current W/32 functions that have been covered in this chapter. See the Appendix for a complete list of the available functions.

Function	Operation
cAppTitle	Adds a title to be displayed in the application window.
cAppWindowInfo	Gets the location and size of the application window.
cAppWindowMax	Maximizes the main application window.
cAppWindowMin	Minimizes the main application window.
cAppWindowNormal	Restores the main application window to its normal size.
cAppWindowTop	Moves the application window on top of the stack.
cCheckItem	Enters or removes a check mark for menu item.
cChoiceList	Displays a scrolling choice list of sorted items.
cChoiceListNS	Displays a scrolling choice list of non-sorted items.
cCloseDIBWindow	Closes a DIB window.
cCreateProcess	Starts a specific process.
cDIBInfo	Gets the height and width of a DIB image.
cDIBWindowInfo	Gets the location and size of DIB window.
cDIBWindowTop	Moves a DIB window on top of the stack.
cDisplayInfo	Gets the screen resolution and number of colors.
cEnableDocking	Enables a docking toolbar.
cEnableItem	Enables or disables a menu item.
cEnableMainWin	Controls the activation state of the application window.
cEnumPrinters	Gets information about available printers.
cFontValues	Returns the values for a selected font.
cForceExit	Exits a program without an OK message box.
cGetCursor	Gets the cursor's current column and row position.

cGetMouse	Gets the current mouse position.
cInputQ	Puts a keycode into the input queue.
cLogin	Displays a login dialog box.
cMaxSize	Resizes a window to maximum size.
cMessageBoxOkC	Displays an OK-CANCEL message box.
cMoveAppWindow	Moves the application window to a new location.
cMoveDIBWindow	Moves a DIB window to a new location.
cNewMenu	Displays a new menu.
cOnWin32s	Checks to see if running under Windows.
cOpenBrowser	Displays a File Open browser window.
cOpenDIBWindow	Opens a new window to display a bitmap file.
cProcessStat	Checks the status of a started process.
cSaveBrowser	Displays a File Save As browser window.
cSelectFont	Changes to the specified font.
cSetCursor	Sets the cursor for the next input operation.
cShowMainWin	Makes the main application window visible or invisible.
cShowStatusBar	Hides or displays the status bar.
cShowToolBar	Hides or displays the tool bar.
cShowVersion	Displays the version of all runtime components.
cSizeAppWindow	Sets the main application window to a specific size.
cStatusText	Displays text on the status bar.
cStringInputQ	Puts a string in the input buffer.
cToolBar	Loads a new toolbar bitmap.
cWaitCursor	Displays or removes an hourglass cursor.
cWinExec	Executes a Windows Application.

CHAPTER 9

BASIC Subroutines

Introduction

The W/32 Application Builder allows you to compile BASIC programs that call external BASIC subroutines. This feature may be used to integrate a number of BASIC programs into a single executable**.** The advantages of doing this are:

- Execution time is much faster, as all CHAIN and LINK operations are eliminated.
- The size of the resulting executable, though large, is much smaller than the executable BASIC programs that it replaces.
- Each BASIC subroutine may be compiled separately and later linked using OBWIN.
- Some developers have found that calling externally compiled subroutines is a more natural way to program. The use of subroutines (instead of CHAINs and LINKs) is similar to the way in which C, Pascal, COBOL and other applications are written.
- It is much easier to write structured code.
- Commonly used routines may be stored in external files, and then called as needed by entering the name of the subroutine in the main program.
- Maintenance is much easier. You make changes to one BASIC subroutine, then simply recompile all the programs that use it.

BASIC subroutines use a number of new statements. This chapter is intended to document these statements and point out how they affect the operation of a few of the conventional CET BASIC statements.

Compiling Programs that Call BASIC Subroutines

Programs that call BASIC subroutines are compiled with OBWIN using the normal syntax and options. The only difference is that you can simultaneously compile several modules with one command. For example:

```
obwin -o main main.b sub1.b sub2.b
```

Individual modules may also be compiled separately and linked together:

```
obwin -c sub1.b
obwin -c sub2.b
obwin -o main main.b sub1.obj sub2.obj
```

When you use the W/32 Integrated Development Environment to develop your application, W32APP automates the compilation process so that it is unnecessary to remember the correct syntax. The -c flag is used, by default.

The BASIC Subroutine Statements

These statements differ from conventional CET BASIC statements in two ways:

- Every statement begins with a dollar sign ($).
- With the exception of $CLEAR and $EXIT, the statements are not executable. Instead, they direct the compiler to treat the module and its constituent statements in a certain way.

The BASIC subroutine statements are:

$MAIN	Indicates that this module is the main routine in the program
$SUB	Indicates that this module is a subroutine
$GLOBAL	Provides a list of variables that are global to all routines
$CLEAR	Clears all variables and dimensioned arrays in the subroutine except for those defined with $GLOBAL
$EXIT	Exits from a BASIC subroutine

The $MAIN Statement

The $MAIN statement must be the first statement in the main module. The syntax is simply:

```
$MAIN
```

There can be only one module which uses the $MAIN statement. The main module is the program segment in which your application begins.

The $SUB Statement

A BASIC subroutine begins with a $SUB statement that defines the subroutine so that it can be called by a $MAIN program or another BASIC subroutine. The syntax is:

$SUB *module-name*

The names of BASIC subroutines are case sensitive, therefore TESTsub, testSUB and TestSub are all different. For example:

```
$SUB  mysub
$SUB  WriteRpt
```

BASIC subroutines are invoked with the CALL statement. The calling statements associated with the $SUB statements above are:

```
CALL  mysub       used to call $SUB mysub
CALL  WriteRpt    used to call $SUB WriteRpt
```

The $GLOBAL Statement

Data can be shared between BASIC subroutines using either of the following methods:

1. Any variable referenced within the same source module or **.b** file refers to the same variable. In other words, all variable names are global between BASIC subroutines that reside in the same source file.
2. Any variables that are used in separate source modules are distinct unless they are defined by a $GLOBAL statement. This is true even for variables with the same name.

This name management feature permits you to write and compile separate BASIC subroutines to be linked into an executable without worrying about conflicting variable names.

When the $GLOBAL statement is used, it must immediately follow the $MAIN in the main module and the $SUB in the BASIC subroutine. Otherwise, you will get an unresolved externals message during compilation.

The syntax for the statement is:

$GLOBAL *variable-list*

The *variable-list* is a list of names which refer to the same variable in all modules. The variables may be simple, scalar variables and/or matrix (array) variables preceded by the keyword MAT.

If multiple variables are to be shared, they may be specified on separate lines or all on the same line with commas separating the variable names. For example, a typical $GLOBAL statement might be:

```
$GLOBAL  GVAR, MAT MYARRAY, ANYSTG$, I%, MAT XSTG$
```

Note that an array must be dimensioned with either the DIM or COMMON statement before it is passed to a BASIC subroutine.

For example, consider the following main program and subroutine. There are three shared variables. The values for CNT% and the array ARRAY% are assigned in the main routine and passed to SumArray which will sum the CNT% number of elements and return the result in ANSWR%. These variables are exactly the same in both programs, and occupy the same address in memory.

```
$MAIN
$GLOBAL CNT%, MAT ARRAY%, ANSWR%
DIM ARRAY%(10)
FOR I% = 1 TO 10
      ARRAY%(i%) = I%
NEXT
CNT% = 10
CALL SumArray
PRINT "The answer is: "; ANSWR%
END
```

```
$SUB SumArray
$GLOBAL CNT%, MAT ARRAY%, ANSWR%
ANSWR% = 0
FOR I% = 1 TO CNT%
      ANSWR% = ANSWR% + ARRAY%(I%)
NEXT
$EXIT
```

To summarize, the important points to remember when using variables in programs that call BASIC subroutines are:

- Every variable defined in a $GLOBAL statement is the same variable in every module in which it is used.
- If a variable is not defined as a $GLOBAL (or COMMON) variable, it may be cleared upon entering or leaving a subroutine by using a $CLEAR statement.
- Each $GLOBAL variable used in a BASIC subroutine must also be declared in the main program.
- The $GLOBAL statement must immediately follow the $MAIN and $SUB statements.

The $CLEAR Statement

Variables are not automatically cleared upon entry into a subroutine. $CLEAR may be used when it is necessary to clear all variables and dimensioned arrays that are not defined as $GLOBAL or COMMON. The syntax of the statement is:

```
$CLEAR
```

The reason for clearing all variables derives from the fact that this is what normally happens when a CET BASIC program is first executed. Moreover, programs rely upon the variables in separate programs being distinct (unless they are in COMMON), even if they happen to have identical names.

$CLEAR is provided so that a *new* program that uses BASIC subroutines to combine a number of separate programs which were formerly integrated using CHAIN, LINK or RUN statements will operate the same way as before.

Unless a variable is in a $GLOBAL list (or held in COMMON), $CLEAR ensures that the following actions occur upon entry into a subroutine:

- Scalar integer and real variables are set to zero.
- Scalar string variables are set to a null string.
- Matrices are cleared. Every matrix is initialized to a one-dimensional array with 10 elements, each of which is zero or null, depending upon the data type.

The $EXIT Statement

$EXIT is an executable statement. When $EXIT is encountered, control passes back to the calling module, to the statement following the CALL. Internally, this

statement operates like a RETURN. An $EXIT is implied at the end of a $SUB module if one does not exist.

The syntax of the statement is:

```
$EXIT
```

The END, QUIT and STOP statements may not be used instead of $EXIT. If one of these statements is detected in any module, the entire program will terminate.

Converting Existing CET BASIC Applications

BASIC subroutines are often used to convert CET BASIC applications which are composed of a number of separate programs. The steps involved include:

1. Determine what the execution *tree* should look like for your application. For example, suppose you have an application with a main menu program which CHAINs or LINKs to other submenu programs.
2. Add $MAIN on the first line of your main (menu) program which uses CALL statements to invoke the next level of (submenu) programs (now BASIC subroutines), depending upon the conditions for their execution (i.e. user response, a fixed sequence or some other method of selection).
3. Define all previously shared COMMON variables with a $GLOBAL statement. Remember to use the MAT keyword in front of any matrix (array) defined as global. (Also make sure that all $GLOBAL variables are dimensioned.)
4. Change all occurrences of CHAIN and LINK to CALL statements. The precise way in which you do this, and the placement of the CALL statement depends upon the structure of your *execution tree*.
5. Use $CLEAR to clear all variables and arrays not defined with $GLOBAL before entering (or exiting) the subroutine so they will not contain values loaded during a previous CALL statement.
6. Modify the subroutines to return to the calling program with a $EXIT. It is no longer necessary to remember the name of the program for statements such as CHAIN "menu". $EXIT will ensure that control is automatically passed back to the $MAIN program which issued the CALL statement.
7. Terminate all GOSUB statements with a RETURN before a $EXIT is executed. Otherwise, the program will transfer control to the last GOSUB statement performed.

8. It is possible to CHAIN, LINK or RUN among sets of main program modules. In these cases, you may still want to use COMMON to communicate variables and arrays between the programs. COMMON statements should only be used in the $MAIN module. To share variables (COMMON or not) with a BASIC subroutine, you must use a $GLOBAL statement.

A Sample Conversion of a CET BASIC Application

The following programs illustrate what needs to be done to convert a main menu program which CHAINs or LINKs to other submenu programs.

BASIC Program menu.b

```
COMMON CONAME$

PRINT "Select Program"
PRINT "1. Accounts Receivable"
PRINT "2. Accounts Payable"
PRINT "3. General Ledger"
PRINT "4. EXIT"
INPUT ANS%
SELECT ANS%
      CASE 1
           CHAIN "ar"
      CASE 2
           CHAIN "ap"
      CASE 3
           CHAIN "gl"
      CASE 4
           QUIT
CEND
```

Converted Program menu.b

```
$GLOBAL CONAME$
WHILE 1
     PRINT "Select Program"
PRINT "1. Accounts Receivable"
     PRINT "2. Accounts Payable"
     PRINT "3. General Ledger"
     PRINT "4. EXIT"
     INPUT ANS%
     SELECT ANS%
          CASE 1
               CALL ar
          CASE 2
               CALL ap
          CASE 3
               CALL gl
          CASE 4
               QUIT
     CEND
WEND
```

BASIC Program ar.b

```
COMMON CONAME$
<main body of ar>
CHAIN "menu"
```

Converted Program ar.b

```
$SUB     ar
$GLOBAL CONAME$ \  $CLEAR
<main body of ar>
     $EXIT
```

BASIC Program ap.b	**Converted Program ap.b**
	$SUB ap
COMMON CONAME$	$GLOBAL CONAME$ \ $CLEAR
<main body of ap>	<main body of ap>
CHAIN "menu"	$EXIT

BASIC Program gl.b	**Converted Program gl.b**
	$SUB gl
COMMON CONAME$	$GLOBAL CONAME$ \ $CLEAR
<main body of gl>	<main body of gl>
CHAIN "menu"	$EXIT

Very few modifications were made to convert the conventional CET BASIC application. None of the code in the <main body> of the programs (which may involve hundreds of lines) was modified. The CHAIN statements (back to the main menu) were replaced with $EXITs. COMMON statements were replaced with $GLOBALs, and $CLEAR was added to clear the other variables and arrays.

Differences from Conventional CET BASIC Programs

Error Handling

It is also important to note that each BASIC subroutine does its own error handling. Every subroutine where an error might occur should begin with an "ON ERROR GOTO" statement. This also applies to lock and interrupt key processing.

Also not that there are some CET BASIC statements that act differently than when compiled into separate program modules. The following sections address these differences:

RESTORE and READ with DATA Statements

Normally, the READ statement, when unaccompanied by a file number, will read through a list of data specified by one or more DATA statements. Each successive READ *consumes* the next item in the data list.

The *pointer* to the next data item may be changed by using RESTORE. This statement can *rewind* the pointer to the first DATA statement or change the current pointer to the DATA statement following a designated statement number.

When using BASIC subroutines, the following behavior should be noted:

- READ statements only read data items located in that module.
- RESTORE only refers to a DATA statement in the same module.

COMMON Statements

The COMMON statement is a powerful executable statement which adds a list of variables to the currently allocated common list. During a CHAIN or LINK, this common list is matched up with the COMMON statement(s) in the target program.

When using BASIC subroutines, it is not advisable to use the COMMON statement in any but the main ($MAIN) module. If access to common variables is required in other modules, a $GLOBAL statement should be used in each module in which you need to access the variables.

User Defined Functions

A user defined function, specified by the DEF statement, is only recognized in the module in which it is defined.

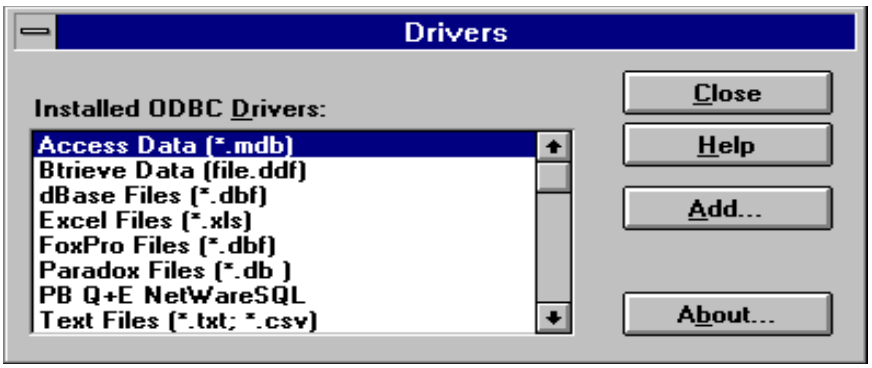
CHAPTER 10

ODBC Support

Introduction

The Open Database Connectivity (ODBC) facility is a standard created by Microsoft to provide transparent access by applications to a wide range of database and file management systems.

ODBC drivers exist for MS Windows file managers (e.g., Dbase, Btrieve), database managers (e.g., FoxPro, Paradox and MS Access) and SQL servers like Watcom SQL, and Gupta SQL servers. Full relational databases such as the MS SQL Server, Oracle, Informix and Sybase also offer ODBC drivers for desktop connectivity.



To use ODBC with a W/32 program, you must have the correct ODBC driver installed on your system. ODBC drivers are available from a number of sources. Often, they are included with the product (e.g. the ODBC driver for Microsoft Access is on the Access release disks).

Once you have the correct ODBC drivers installed and a data source configured according to the documentation for the driver, you can access that data source with a W/32 program.

The standard OPEN, CLOSE, READ, MAT READ, WRITE, MAT WRITE, READNEXT, MAT READNEXT, READPREV and MAT READPREV statements are used to access an ODBC data source. All values written to or read from the data source are strings. The W/32 Application Builder will convert all string values to and from the native data source type automatically.

Note that the environment variable B_ODBCERR must be set to one in your **w32app.w32** file to display messages for any ODBC errors that may occur.

Opening a File

To open an ODBC data source, use the OPEN statement with the INDEXED option, in either UPDATE or INPUT mode. The filename argument determines which ODBC data source is opened and may be indicated in one of two formats:

ODBC*xx*;

The first format brings up a dialog box so the user may select from any of the currently configured data sources. The *xx* in ODBC*xx* is either "SS" for a snapshot or "DS" for a dynaset. A snapshot contains records that are valid when selected the first time, while a dynaset maintains consistency with the actual data source by checking if the record is still current and getting a new record if it is not. A snapshot is quicker to use as it does not have to constantly re-query the data source.

ODBC*xx*;DSN="<*data source name*>;"

The second format opens a specific data source name that was set when you configured the data source through the ODBC administrator. For example, the following statement would open a data source named "Test Data" for reading and writing on channel #5 in snapshot mode:

```
OPEN #5:"ODBCSS;DSN=Test Data;", UPDATE INDEXED
```

When opening a data source using an explicit name, you must end the name with a semicolon. Otherwise, a "File not found" error is displayed.

Reading a File

Once the OPEN statement establishes a connection to the data source, the key field in the READ statement is used to obtain specific information about the data

source and access the records. The key may be set to one of the following reserved words, or it can be an SQL SELECT statement.

TABLES — Returns all names of type "table" that are found. Two values are returned for each table - name and type.

ALLTABLES — Returns all table names, including system tables. Two values are returned for each table - name and type.

TYPES — Returns information about the data types supported. The nine values returned are:

1. The native data base type
2. The associated SQL type
3. The column precision
4. Whether or not the column is case sensitive
5. Whether or not the column is searchable
6. Whether or not the column is an auto-increment column
7. The column prefix
8. The column suffix
9. The CREATE PARMS for the column

The information returned by TYPES is only required for advanced operations on the data source, such as adding new tables or columns.

The following example shows how to use the TYPES reserved word. Note that it is assumed that the data source supports a maximum of 30 types.

```
dim type$(30,9)
mat read #1, "TYPES": type$
print
print "Data Types:"
print
for I% = 1 to 30
    if type$(i%,1) = "" then goto endtype
    print   type$(i%,1)+"("+type$(i%,2)+")"+"("+type$(i%,3)
+")"+"("+type$(i%,4)+")"+"("+type$(i%,5)+")"
    print "("+type$(i%,6)+")"+"("+type$(i%,7)+")"+"("+
type$(i%,8)+")"+"("+type$(i%,9)+")"
    wait
next i%
endtype:
```

COLUMNS:*table* Returns all of the column names in the specified table. Three values are returned for each column: the name, field precision, and field type.

REQUERY Re-queries the data source to make sure that all records are current. No information is returned to the program.

An SQL SELECT statement may also be used to select a number of columns from a specific table. SQL SELECT statements may contain many options. (The maximum length is 2048 characters.) The simplest form is:

```
"SELECT <column names> FROM <table>": <numcol%>
```

where:

column names

is either a comma separated list of column names or "*" to select all columns.

table

is the name of the table to be accessed.

numcol

is the number of columns that were selected.

Do not use MAT READ for a SELECT statement. The correct usage is:

```
READ #5,"SELECT * FROM Authors": NUMCOLS%
```

A new keyword "RESULTCOLUMNS" has been added for READ statements. After a successful SELECT statement has been executed, this keyword may be used to retrieve a description of the columns that were selected. There are three values per column; the column name, precision and type. For example:

```
READ #1, "SELECT * FROM SECTION": NUMCOLS%
DIM col$( NUMCOLS%, 3)
MAT READ #1,"RESULTCOLUMNS": col$
```

After any SELECT statement, you are positioned at the beginning of the record set. Four new keywords have been added to READ to allow you to move through the records. They are:

MOVEFIRST Moves prior to the first record. (READNEXT returns the first one.)

MOVELAST Moves after the last record. (READPREV returns the last record.)

MOVE:num Moves relative to the current record. Negative values move back. Positive values move forward. Zero does nothing. Returns EOF

if moving backward and you have reached the beginning or if moving forward and you are at the end.

STATUS — Returns the zero-based index of the current record. The first record will be record 0. If at EOF, then -1 is returned. If the current index cannot be determined, then -2 is returned. .

Note that if you do a MOVELAST and then a READPREV or MOVE:-1, the last record will be current, but STATUS returns -2 because ODBC does not yet know the index of the last record.

The total number of records can not be determined without scrolling through all of the records or moving to the last record and doing STATUS. If you are performing READNEXT operations from the beginning, then STATUS can determine the current record number. For example:

```
READ #1,"any SELECT statement": numcols$
IF EOF(1) THEN .....   \ REM Always check for EOF on a SELECT
WHILE NOT EOF(1)
    READ #1,"MOVE:1":dummy$
WEND
READ #1,"STATUS": lastrec%
```

Updating a File

The WRITE statement is used to add, update or delete records. If the WRITE operation fails, an EOF condition will be generated.

To delete or update a record, first use READNEXT/READPREV statements to position the pointer at the correct record in the record set. The key field for the WRITE statement can have three values: "UPDATE", "DELETE" or "ADDNEW".

Updating a Column

To update the Comment column for Author "Joe Smith" with a new comment perform the following:

```
dim result$(2)
while 1
    mat readnext #5,"": result$
    if eof(5) then goto notfound
    if result$(1) = "Joe Smith"
        result$(2) = "a new comment"
        mat write #5,"update": result$
        goto done
        ifend
    wend
```

```
notfound:
    print "Couldn't find Joe Smith"
done:
```

Deleting a Record

To delete the entire record for "Joe Smith":

```
dim result$(2)
while 1
    mat readnext #5,"": result$
    if eof(5) then goto notfound
    if result$(1) = "Joe Smith"
        write #5,"delete": “
        goto done
        ifend
    wend
notfound:
    print "Couldn't find Joe Smith"
done:
```

An empty string is written. Although it is not used, the compiler will generate an error if a WRITE statement is encountered with no values to write.

A Sample Program

The following program shows how to browse a data source, select the table and columns to be retrieved, and display the selected data.

Note that the environment variable B_ODBCERR must be set to one in your **w32app.w32** file to display messages for any ODBC errors that may occur.

```
REM Allow up to 20 table and column names.
    dim tables$(20,2), col$(20,3), rslt$(20)
REM Open a snapshot for input.  Choose the data source at run-time.
    open #1: "ODBCSS;", input indexed
tables:
    REM Get the tables defined for this data source.
        mat read #1, "TABLES": tables$
    REM Check for EOF.  (EOF on TABLES is an error.)
        if eof(1)
            msgbox "No tables defined" \ quit
            ifend
    REM Display the tables in this data source.
        print \ print "Available Tables:" \ print
        for I% = 1 to 20
            if tables$(i%,1) = "" then BREAK
            print TAB(8);tables$(i%,1) + " (" + tables$(i%,2) + ")"
```

```
                next i%
        REM Let the user choose which table to browse.
            print \ print \ print "Select a Table name (or Q to quit):";
            linput tname$
            if tname$ = "Q" then quit
            mat read #1, "COLUMNS:"+tname$: col$
        REM Check for EOF.  (EOF on columns is an error.)
            if eof(1)
                msgbox "No columns in the selected table"
                goto tables
                ifend
        REM Display the columns in this table.
            print \ print \ print "Available Column names:" \ print
            for I% = 1 to 20
                if col$(i%,1) = "" then BREAK
                print  TAB(8);col$(i%,1) + " (" + col$(i%,2) + ")" + " (" +
    col$(i%,3) + ")"
                next i%
        REM Let the user choose which columns to retrieve.
            print \ print \ print "Select Column names (comma separated)
    for retrieval (or * for all):";
            linput cname$
            mat read #1, "SELECT "+cname$+" FROM "+tname$: a$
        REM Check for EOF.  (EOF on SELECT means no records were
    selected.)
            if eof(1)
                msgbox "EOF on SELECT" \ goto tables
                ifend
        REM Display the selected columns from the record set.
            while 1
                print \ print
                mat readnext #1, key$: rslt$
                if eof(1) then BREAK
                for I% = 1 to 10
                    if rslt$(i%) = "" then BREAK
                    print TAB(8); \ print rslt$(i%)
                    next i%
                wait
                wend
            print "End of Record Set Reached"
            wait
        REM Select another table.
            print CRT$("CLEAR");
goto tables
```

Another example has been stored in **cetlib\samples**, by default. There you will find an ODBC program that uses a special form designed to illustrate how you can read and write a file for which you have installed the ODBC drivers.

All the sample programs have also been uploaded to the WIN32 library on the Colorado BBS as **demoprj.zip**.

CHAPTER 11

Development Utilities

Introduction

The CET BASIC product includes a variety of utility programs that may be used in program development and data file recovery. These utilities are described in this chapter. To clarify their syntax, the following conventions have been used:

[ ]	Square brackets indicate an optional parameter.
|	A vertical bar indicates a choice between two parameters.
italics	Italics indicate a variable name to be supplied by the user.

Although Windows commands are case insensitive, all the names of the utilities except for CETLNK32 are shown in mixed-case letters. We recommend that you get in the habit of entering them this way in case you port your application to a UNIX environment where case mode is important.

It is important to note that these utilities are all 32-bit programs that should be stored in the directory with your W/32 executables, **\cetbin**, by default. If you have a copy of the CET BASIC DOS product on the same system, you will have 16-bit versions of the utilities with the same name stored in the **\bin** directory. Adjust your PATH so that the desired directory is specified first to avoid possible problems with trying to use the incorrect version.

On-line help is available for all the utilities by simply entering the command without any arguments. This information is stored in **util.msg** in the same directory with the executables.

The utilities are documented in alphabetical order for your convenience.

The Bcopy Command

This utility may be used to copy or merge all of the records or a specific group of records from one file to another file. The destination file may be of the same or a different type.

The command line syntax is as follows:

Bcopy [-f *fromkey*] [-t *tokey*] [-c *counter*] [-r] [-m] [-s] [-d *reclen*] [-i *reclen keylen*]<*from*><*to*>

where:

-f *fromkey*

Copies starting from the record with the *tokey*, a key in an indexed file or a record number in a direct file. The default is to start at the beginning.

-t *tokey*

Copies the *from* file up to (but not including) the record specified by the *tokey*. The default is to stop at the end of the file.

-c *counter*

Specifies the number of records to copy.

-r

Replaces the *to* file, if one exists. The default is to query the user if the existing file should be replaced.

-m

Merges records from the source *from* file into the destination *to* file. When this option is used and the *to* file does not exist, the program asks the user to specify which action to take "Create/Quit (C/Q)" and press Enter to continue.

Note that this option will not change the parameters of an existing file. This feature makes it possible to copy records to a file with a different key and/or record length.

-s

Creates a sequential output file.

-d *reclen*

Creates a direct output file.

-i *reclen keylen*

Creates an indexed output file.

Operation Notes:

1. When the -f (*fromkey*) and the -t (*tokey*) or -c (*counter*) options are used in the same command line, copying starts from the record specified with *fromkey* and ends when the counter value is exceeded or when the record specified with *tokey* is reached, whichever comes first.

2. When the -r (*replace*) or -m (*merge*) options are not specified in the command line and the *to* file already exists, the program asks the user to specify which action to take:

   ```
   Replace/Merge/Quit (R/M/Q):
   Press desired key followed with Enter.
   ```

3. When both -m and -r options are defined, -m will be ignored.

4. When the destination file does not exist and none of the -s, -d or -i options are defined, the file is created with same type, record length and key length as the source file.

   When more than one option is used to specify the *from* file type, the last option specified in the command line will be used.

5. When the -m (*merge*) option is used and the key or record length in the *from* file differs from that in the *to* file, Bcopy will use the specifications in the *to* file and truncate or expand the length(s) accordingly.

The Bcreate Command

The Bcreate utility may be used to create indexed and direct files or to clear the contents of an existing file. If you plan on creating files from within a W/32 program, we recommend that you use the Bcreate function (see the next chapter) to avoid having to exit the BASIC environment to perform the operation.

Note that this utility is an enhanced version of the CREATE command which you may have used under other platforms. If you like, you may rename this command to CREATE for compatibility purposes.

Bcreate recognizes a variety of command options. The minimum abbreviations have been underlined. Note that preceding the options with a left parenthesis is optional.

The syntax is:

Bcreate <*filename*> [( ] [*options*]

where:

filename

The name of the file in either an MS DOS or THEOS format. For example, **cp\data\custname** and **cp.data.custname** are considered the same.

Nodesize *value*

Specifies the node size to use when creating an indexed file. The default is 512. Although W/32 programs can read files with the default node size, we recommend that you use "n 1024" to be consistent with files you may create using the Bcreate function.

indexed

Creates an indexed file which is actually two files: a **.idx** file for the record's keys and a **.dat** file for the data.

direct

Creates a direct (relative access) file.

reclen *n*

Allocates a record length of *n*.

keylen *n*

Allocates a key length of *n*. This option is ignored for direct files.

clear

Clears the contents of an existing file. The key and record length will remain the same unless any of the keylen, reclen and/or nodesize options are specified in the command line.

Note that the file must be closed in order for this option to function successfully.

All CET BASIC files are dynamically allocated so any THEOS filesize option will be ignored. If the command is used to create a THEOS keyed file, the program will create an indexed file instead.

The following examples will create the indexed files **data\custname.idx** and **data\custname.dat** with a key length of 11 and a record length of 350:

```
Bcreate data\custname indexed reclen 350 keylen 11
Bcreate custname.data indexed reclen 350 keylen 11
```

The Blist Command

Blist may be used to display the contents of indexed, direct or sequential files. A banner will be displayed to identify the name and type of file being listed.

The syntax of the command is as follows. Entering the command without any arguments will display a help screen.

Blist [-h] [-f *fromkey*] [-t *tokey*] [-c *counter*] [-l *length*] [-p *page*] [-m *margin*] [*field-numbers*] [-v] [-u] <*filename*>

where:

filename

Specifies the name of the file in a DOS or THEOS format. For example, either of the following names may be used to list **cp\data\custname**:

.\cp\data\custname (or cp\data\custname) cp.data.custname

-h

Displays the contents of the file in a hex dump format.

-f *fromkey*

Lists starting from the record with the *tokey*, a key in an indexed file or a record number in a direct file. The default is to start at the beginning.

-t *tokey*

Lists the file up to (but not including) the record specified by the *tokey*. The default is to stop at the end of the file.

-c *counter*

Specifies the number of records to list.

-l *length*

Specifies the maximum length of the print line. Characters that do not fit on the line will wrap to the next line. The default line length is 79.

-p *page*

Specifies the output page length. The default is 24 rows. Use "-p 0" to turn off pagination.

-m *margin*

Specifies the number of characters to indent the second and all subsequent lines of a multi-line record. The default is 5.

field-numbers

Specifies the numbers of the fields to be displayed from the formatted file. The default is to display all the fields in the record.

The first data field starts with 1. The keys in an indexed file and the record numbers in a direct file are always displayed.

-v

Turns on *verbose* mode and displays the field number, field type and length of field before the contents of each data field. The field type will be either S for a string, F for a float or I for an integer. For example:

123-45-6789: (1S17)D.G.Widgets, Inc.,(2S11)Don Gillett,(3S17)…

-u

Specifies that the files are in UX-BASIC format.

The contents of indexed files will be displayed in alphabetical order by key. Direct files will be listed in ascending numerical order.

The Bpretty Command

The Bpretty utility may be used to rearrange the spacing and indentation of CET BASIC source files to reflect the program structure.

It is important to note that the syntax and operation of the 32-bit version Bpretty is different than the utilities that are available with the other CET BASIC products. Input from the keyboard is no longer accepted. The name of the source file must be entered on the command line. The specified file will be rewritten in place after storing the original code in a file with a **.bak** extension. By default, the backup file will have the same name as the source file. (File names must be in a DOS format.) The syntax is as follows.

Bpretty [-*n*] [-t] [-i] <*source*.b> [< *source*.bak>]

where:

-n

Indents *n* spaces. The default is 4.

-t

Replaces spaces with TAB characters wherever possible.

-i

Inserts text from #include files before performing any other operation.

The Brenum Command

Brenum may be used to renumber or unnumber BASIC source programs with up to 999,999 lines. Filenames must be in a DOS format.

It is important to note that the syntax and operation of the 32-bit version Brenum is different than the utilities that are available with the other CET BASIC products. Input from the keyboard is no longer accepted. The name of the source

file must be entered on the command line. The specified file will be rewritten in place after storing the original code in a file with a **.bak** extension. By default, the backup file will have the same name as the source file.

The syntax is:

Brenum [-a] [-n *initial*] [-i | + *increment*] [-b *begin*] [-e *end*] [-u]
<*source*.b> [< *source*.bak>]

where:

-a

Causes all lines within the given range to be renumbered. When omitted, the default is to renumber only the lines with numbers.

Note that the lines of code contained in any "#include" files will not be renumbered, even when the -a option is used.

-n *initial*

Starts renumbering with the number *initial*. The default is 10.

-i *increment*

Increments the new line numbers by *increment*. The default is 10.

+ *increment*

Works like the -i option to specify how the numbers are to be incremented.

-b *begin*

Begins renumbering at the *begin* line. When omitted, the default is to begin with the first line of the program.

-e *end*

Stops renumbering at the *end* line. When omitted, the default is to end with the last line of the program.

-u

Removes all line numbers from the *input* file.

Note that renumbering portions of a file may cause those lines to be moved to preserve line number order.

The flags may be arranged in any order provided that the parameters associated with the flags are entered in corresponding order. For example, the following command will renumber all lines in **source.b** after storing the original contents in a file with the default name **source.bak**.

```
Brenum -a source.b source.bak
```

The following examples will all produce the same results: **source.b** will be renumbered from line 200 to line 300. The new line numbers will begin with 600 and increase in increments of 5. The file **xsource.bak** will contain the original source code.

```
Brenum source.b xsource.bak -nibe 600 5 200 300
Brenum source.b -b 200 -e 300 -n 600 -i 5  xsource.bak
Brenum -n600 +5 -be 200 300 source.b xsource.bak
```

The CETLNK32 Command

The CETLNK32 program may be used to create a library of object files that can then be linked into a main program using the library flag. The syntax is:

cetlnk32 /lib /out:*library*.lib *file1*.obj... | *.obj

library

Specifies the name of the library file to be created.

file1

Specifies the name of the object file(s) to be included in the library.

***

Indicates that all of the object files in the current working directory should be included in the library.

Both of the following examples will create a library called **myobjlib.lib** in the current working directory:

```
cetlnk32 /lib /out:myobjlib.lib dg3btnmsg.obj dgerror.obj dgkey.obj

cetlnk32 /lib /out:myobjlib.lib *.obj
```

After creating a library, add its path to the LIB directory path in your **autoexec.bat** file. Then, you can link to it by compiling your main program with the library flag as in:

```
obwin -o mainpgm mainpgm.b -lmyobjlib
```

WARNING: Due to a problem in the Microsoft LIB utility, you will need to delete the current library before you re-invoke the CETLNK32 command to rebuild the library to include a new object.

The Dcheck Command

The W/32 product currently uses D-ISAM, an indexed file manager from Byte Designs. D-ISAM files are compatible with C-ISAM, the file structure which is

used in the other CET BASIC products. (Currently, a Windows version is not available for the C-ISAM product from Informix.)

Dcheck is a D-ISAM utility designed to check and fix indexed file inconsistencies. It is important to note the following:

- Dcheck only accepts DOS filenames.
- Dcheck may be used to view the file's structure, list the contents of the keys and the data file order and check file consistency without actually modifying the file.
- Dcheck is capable of building a full index from a skeleton and a data file. Deleted records are not removed so the size of the file remains the same.

The command line syntax is as follows. Note that the first parameter (optional) is a dash; followed by one or more of the listed letters. The second parameter is the full path name of the file to be checked. Enter the command without any arguments for on-line help.

Dcheck [-] [i] [l] [n] [y] [q] [b] [h] [x] [o] <*isamfile*>

where:

-i

Checks the index only. No data field checking is performed.

-l

Lists the contents of the indexes to the screen.

-n

Does not try to rebuild the index.

-y

Tries to rebuild the index if it is imperfect.

-q

Runs in quiet mode without displaying anything on the screen unnecessarily.

-b

Rebuilds the index from the data, even if the index is OK. To build an index for data from another source:

1. Organize the data into a file with each record separated by a newline character (ASCII 10).
2. Create an indexed file with a different name, with the desired key. You need not add the data.

3. Move your data file to the .**dat** file for the newly created indexed file with the command:

   dcheck -b isamfile.

This will cause your data file to be fully indexed as described when you built the ISAM file.

-h

Only prints out the file structure, with no checking. This option does not require exclusive access to the file.

-x

Outputs the contents of each record as hexadecimal.

-o

Prints a list of numbers which refer to the record number in the order that they are indexed. This feature may be used with the -q option to create a filter for quick printing.

If an error is encountered, or if the -b option is used, Dcheck disposes of the indexes, then rebuilds them from the data. The data is never destroyed in the repair process.

CHAPTER 12

Built-In C Language Routines

Introduction

The W/32 Application Builder provides you with a wide variety of C language routines that enable you to perform operations which would be difficult or inefficient to code using BASIC alone.

When using these functions, note that:

- √ Case mode is important. The name of the function must be entered in the same case as shown.
- √ Unless stated otherwise, the file names may be in either a DOS or THEOS format. CET treats a name that contains a period as a THEOS name and performs the necessary conversion. For example, the following file names are considered identical:

  ```
  cp.data.custname        cp\data\custname
  ```

  If the DOS file name contains a period, use the following format:

  ```
  .\trans.fil
  ```

This chapter describes some of these routines. Refer to the *CET BASIC Library Manual* for information on the additional functions that are available.

The Built-in C Routines

The currently supported routines have been documented in this chapter. As developers need additional functionality, new C routines are provided. Check with your CET distributor if you do not find the feature you need.

For your convenience, the routines have been grouped here by functionality.

File Related Functions

The following functions have been provided to eliminate the need to use CSI/CSH to perform a file-related operation. Failures will generate a trappable ON ERROR condition that is consistent with the errors found in **brun.err**.

Bcreate

Bcreate operates like the utility described in the previous chapter except that it is executed from within a BASIC program by using a CALL statement instead of a CSI. The syntax of the statement is:

CALL Bcreate("*create command line*")

Case mode is important. The name of the function must be entered in the same case as shown. The name of the file to be created or cleared may be in either a DOS or THEOS format. The following statements produce the same results.

```
CALL Bcreate("data\custname indexed reclen 350 keylen 11")
CALL Bcreate("custname.data indexed reclen 350 keylen 11")
```

Both of the commands above will create **data\custname** as an indexed file with a key length of 11 and a record length of 350.

The following example illustrates how to use Bcreate to clear the contents of a file. Note that the file must be closed before it can be cleared.

```
CALL Bcreate("cp\data\custname clear")
```

Berase

Berase is designed to erase a file and eliminate having to perform a CSI statement to execute the DOS DEL command. In either case, the file must be closed before it can be erased.

The syntax of the statement is:

CALL Berase("*filename*")

The *filename* may be in either a DOS or THEOS format. If the file is indexed, both the **.idx** and the associated **.dat** file will be erased.

Brename

The Brename function is designed to eliminate having to perform a CSI statement to execute the DOS RENAME command. In either case, the file must be closed before it can be renamed.

The syntax of the statement is:

CALL Brename("*original-filename new-filename*")

The *filename* may be in either a DOS or THEOS format. If the file is indexed, both the .**idx** and the associated .**dat** file will be renamed at the same time.

Bcopy

The Bcopy function may be used to copy a file without having to leave the BASIC environment to perform an operation with a CSI statement. File names may be in either a DOS or THEOS format. If the destination file does not exist, it will be created with the same specifications as the source file.

The syntax of the CALL statement is:

CALL Bcopy("*source-filename destination-filename*")

Note that this function does not provide you with all the features available with the Bcopy command described in the *Development Utilities* chapter.

Bfilestat

This function may be used to determine if a file or directory exists. Its syntax is:

CALL Bfilestat("*filename*",addrof(stat$))

where:

filename

The filename in a DOS or THEOS format. If the file is indexed, the extension (**.idx** or **.dat**) must be specified as in **data\test.idx**.

stat

A string that contains either a D for directory, R for a file, or a null if the *filename* does not exist.

The Bfilestat function has been available in the past as filestatus. Although this function is still supported, filestatus only recognizes DOS formatted names.

Bgetcwd and Bchdir

The Bgetcwd function may be used to get the current working directory (CWD). Bchdir may be used change the CWD. It is important to note that this is a permanent change. The syntax of the statements is:

CALL Bgetcwd(addrof(cwd$))
CALL Bchdir(cwd$)

Bindinfo

This function may be used to find out information about an open indexed file. The syntax is:

CALL Bindinfo(channel%,addrof(count),addrof(rec%),addrof(key%))

where:

channel

An integer variable or constant representing the channel on which the indexed file was opened.

count

A numeric value which will return the number of records in the file.

rec

The length of the data record. Note that this value has traditionally been the sum of the characters in the data record plus those in the key.

key

The return value is the length of the key.

Environment Related Functions

Bgetenv and Bputenv

The Bgetenv function may be used to determine the value of an environment variable that has been set in the **w32app.w32** file in use.

Bputenv may be used to set some of the variables to a new value only during the execution of the current program. (Check with your CET distributor if you have a question about which variables may be reset.) After the program ends, the original value will be restored.

The syntax of the statements is:

CALL Bgetenv(addrof(setting$),env$)

CALL Bputenv(envsetstrg$)

where:

setting

Returns the information stored in the environment variable or a null if the requested information is not available (i.e. the variable has not been set).

env

Specifies the environment variable.

envsetstrg

Specifies the command to set the environment variable to a new value.

The following code segment illustrates how these features may be used:

```
CALL Bgetenv(addrof(s$), aa$)
PRINT "Current value of ";aa$;" is <";s$;">"
REM Now change environment variable in form "<VAR> = <LIST>"
new$ = "B_USER=sk"
CALL Bputenv(new$)
```

Bsernum

Bsernum allows you to get the serial number from the CET KeyPlug so that it may be used to copy protect your application. Refer to the OPTION SERIAL statement in the *CET BASIC Language Reference Manual* for another method. The syntax of the CALL statement is:

CALL Bsernum(addrof(serno%))

where:

serno

The serial number programmed into the CET KeyPlug.

Program Related Functions

Becho and Bnoecho

Becho and Bnoecho may used to turn the echo of characters on or off during LINPUT and INPUT operations. No arguments are used. The syntax is:

CALL Becho
CALL Bnoecho

Bkbshift

The Bkbshift function may be used to retrieve the state of the shift keys on the PC keyboard. These keys include the Right Shift, Left Shift, Ctrl, NumLock, CapsLock, and Insert keys. The Alt/<key> sequence is not supported as this is a special state used by Windows. The syntax of the statement is:

CALL Bkbshift(addrof(key%))

where:

key

Returns a coded integer where each bit represents the state of the keys as:

Bit Set	Condition
0	Right Shift key entered
1	Left Shift key entered
2	Ctrl key entered
3	(unsupported)
4	Scroll Lock turned on
5	Num Lock turned on
6	Caps Lock turned on
7	Insert Mode turned on

For example:

```
PRINT crt("clear") \  PRINT "Hit Shift, Ctl  or other key [CR when done]:"
PRINT "Right   Left                      Scroll   Num    CAPS  Insert "
PRINT "Shift    Shift    CTRL    Lock    Lock     Lock   Mode   "
PRINT "_____  _____  _____  _____  _____  _____  _____  "
WHILE 1
      CALL Bkbshift(addrof(key%))
      if key% <> 0
            Rshift%       = key% and 01H
            Lshift%       = key% and 02H
            Ctrl%         = key% and 04H
            ScrollLock%   = key% and 10H
            NumLock%      = key% and 20H
            CapsLock%     = key% and 40H
            InsMode%      = key% and 80H
            PRINT at(1,7);
            IF(Rshift%          ) then PRINT tab( 3);crt("rvon")+" "+crt("rvoff");
            IF(LshIFt%          ) then PRINT tab(10);crt("rvon")+" "+crt("rvoff");
            IF(Ctrl%            ) then PRINT tab(17);crt("rvon")+" "+crt("rvoff");
            IF(ScrollLock%      ) then PRINT tab(24);crt("rvon")+" "+crt("rvoff");
            IF(NumLock%         ) then PRINT tab(31);crt("rvon")+" "+crt("rvoff");
            IF(CapsLock%        ) then PRINT tab(38);crt("rvon")+" "+crt("rvoff");
            IF(InsMode%) then PRINT tab(45);crt("rvon")+" "+crt("rvoff");
      IFEND
      GET a%
      IF a% <> 0 THEN BREAK
WEND
```

APPENDIX A

Implementation Notes

Introduction

This section is designed to cover information on a variety of miscellaneous topics to assist you in implementing your W/32 Windows application.

The Application Window Size and Location

When you execute a W/32 program that uses a text-based application window, the program will use the entire window, by default. The starting position is x=0 and y=50 (in pixel coordinates).

You may override the default position by passing the argument -wposX,Y on the command line as you execute a program. For example, the following command would open a window starting at position 100,50:

```
test -wpos100,50
```

The initial size can also be altered by using the command line option -wsizeX,Y. It is important to note that both of these arguments are case sensitive and must be entered in lower case letters to have any effect.

In order to think in terms of *pixels*, it may be helpful to remember that a character displayed in the default courier 9-point font is approximately 8 pixels wide and 13 pixels high.

The Default PIF File

Whenever you run an MS-DOS command from inside Windows, by default the command will run in DOS full-screen mode (although you can always switch to *windowed* by pressing Alt-Enter). This also applies to W/32 programs that CSI/CSH to MS-DOS commands.

Calling a DOS program normally causes a *flash* as the screen clears to display the program in a DOS window. When the program ends, you will see another flash as the screen clears and the window for your W/32 application is restored.

To avoid this flash, change the Windows default to automatically run MS-DOS programs *windowed* instead of *full-screen*. This is done by using the PIF Editor to modify the file **_default.pif**. If you change the Display Usage property from "Full Screen" to "Windowed", anytime you run an MS-DOS program, it will open and run inside a small DOS window on the Windows desktop (i.e. the screen does not clear). When you exit the program, the window will close.

If you have a specific DOS program that you want to run full-screen from inside Windows, we suggest that you create an icon for the program and use a **.pif** file in which you specify "Full Screen" for the Display Usage. For more information about program item icons and .**pif** files, refer to your Microsoft documentation.

The Current Working Directory

Any (data) file that is opened by a W/32 program must be specified as an absolute path or a path relative to the current working directory (CWD).

The CWD can be specified in two ways. When a program is executed from the File Manager, the CWD is the directory where the program is started. If the program is run from the Program Manager (after adding it to a Program Group), then the CWD is part of the Properties for the program. You can change or view the properties by selecting the icon and then “Properties” from the File Menu.

Two built-in C language routines, Bgetcwd and Bchdir, are provided for your use in determining the CWD and changing the CWD, respectively. Refer to the *Built-In C Language Routines* chapter for specific information.

The File Name

File names may be specified in either a DOS format or a THEOS format. CET treats a name that contains a period (but does not begin with a drive code, a slash (\) or a dot slash (.\)) as a THEOS name and performs the necessary conversion. The following file names are considered identical:

```
data\custname          custname.data
```

Operating system conventions should be followed to insure that file names are interpreted correctly, especially if the DOS file name contains a period as in:

```
.\trans.fil
```

A DOS format name will be recognized by a preceding "\", ".\", or drive code only. The complete path name may be specified as in **\pos\cp\data\custname** or relative to the current directory by **.\memo.txt** or **.\trans**. Any other naming convention will be interpreted as a THEOS file description.

For example, the following statement indicates a THEOS file description and will cause the program to output the file **txt\invoice**.

```
OPEN #1: "invoice.txt", OUTPUT SEQUENTIAL
```

If the file is really **invoice.txt** relative to the current directory, the statement above should have been

```
OPEN #1: ".\invoice.txt", OUTPUT SEQUENTIAL
```

The Default Search Sequence

The W/32 Application Builder behaves like other Windows programs when searching for an executable. The search sequence is as follows:

the directory where the calling program was executed
the current working directory
the \windows or \windows\system directory
the directories set in the PATH variable
the directories set in the B_LKPATH variable

The Number of Open Files

Currently, an application may open up to 64 file channels at a time. Forty-eight of these channels may be used to open ISAM files. Be aware that the more files you have open, the more memory your program will require. Refer to the *W/32 Compiler* chapter if you have a question about memory requirements.

The CSI/CSH Statement

The CSI/CSH statement may only be used to execute a 16-bit program. These statements should be replaced with a call to cWinExec or cCreateProcess when performing a 32-bit operation.

Actually, we recommend that you eliminate leaving the BASIC environment to perform an operation with a CSI statement whenever possible. Special file-related functions have been provided for that purpose. (Refer to the *Built-in C Language Routines* chapter for information on the available functions.)

When you do use a CSI statement, the command to be executed is written to an MS-DOS batch file in the **\tmp** directory. Because this is a batch file, multiple commands can be entered by separating them with a <carriage-return>/<linefeed> produced by CHR$(13)+CHR$(10).

For example, to change to the **\tmp** directory and delete the file **tst.exe**, enter:

```
CMD$ = "CD \TMP"+CHR$(13)+CHR$(10)+"DEL TST.EXE"
CSI CMD$
```

Your program will be suspended until the last command that was passed to the CSI statement terminates. While the CSI command is executing, an MS-DOS icon will appear at the bottom of the screen. When the CSI command finishes, the icon will disappear and your program will resume execution.

Because some commands that are passed to CSI require keyboard input, an option has been added to the CSI statement. If the first seven characters of the passed string are "NOICON:", then the MS-DOS window will appear as another window (on the screen) that will accept user input. Your program will still be suspended until the command has completed.

For example, to format a floppy disk in drive A:

```
CMD$ = "NOICON:FORMAT A:"
CSI CMD$
```

At this point, an MS-DOS window would appear on the screen with the message from the FORMAT command: "Insert new disk for drive A: and press ENTER when ready...". The user would then press the ENTER key and wait while the disk is formatted. When the format command finishes, the MS-DOS window disappears and the program resumes execution.

Also note that if any command passed to CSI has output that must be read, you will need to use the NOICON option and possibly the multiple line option to insert the MS-DOS PAUSE command as the last line.

For example, if you issue the DIR command to CSI, by default, the window will disappear as soon as the command is finished. You would not have time to read the output. Issuing the following command would solve this problem:

```
CMD$ = "NOICON:DIR"+CHR$(13)+CHR$(10)+"PAUSE"
CSI CMD$
```

The MSGBOX Statement

Message boxes are often used in Windows programs. The message "Application Exit" displayed by the File-Exit command is an example of how this feature may be used. The message is displayed until the operator clicks on the OK button.

Message boxes may be used anywhere your program would use a PRINT statement. The syntax is:

```
MSGBOX string-expression
```

For example:

```
MSGBOX "Are the correct forms loaded on the printer?"
```

cMessageBoxOkC is a similar function that displays an OK-Cancel box so the user may choose to continue or cancel the operation.

Printing

When a W/32 program opens the file "PRINTER*n*" where *n* is 1 through 9, a spooled print file is automatically created in the **\tmp** directory. When the channel associated with the print file is closed, the file is sent to the printer.

The first phase of 'real' Windows printing has been implemented so that you can setup a specific printer to use or allow the user to select one from a Print dialog. In either case, you will still be printing ASCII text (with embedded control sequences). The next phase will allow you to send an image to the printer.

To use the new printer interface, set the following environment variable in the initialization file called **w32app.w32**, by default:

```
B_WINPRINT=1
```

Note that if B_WINPRINT is not set or is equal to a value other than one, then your W/32 applications will use the **.bat** file printing method documented in the following section.

When B_WINPRINT is used, the variable B_PRINTER*n* may also be set to specify the printer(s) in one of the following ways:

- If B_PRINTER*n* is not set or is equal to DEFAULT as in "B_PRINTER1=DEFAULT", then the system default printer specified in Print Manager is used. As soon as the printer channel is closed, the job is automatically turned over to the Print Manager.

- A specific printer may be defined by setting "B_PRINTER*n*=printer name". The Print Manager will handle the printing the way it does when you set the variable to DEFAULT.

  Under Windows NT and 95, this name must be the one assigned to the printer in Print Manager. Under Windows, you do not have the option to name a printer so use the name of the driver that was installed on the port (e.g. "HP LaserJet III").

  Note that this is the only case, where the printer number 1 to 9 in the variable name is associated with the LPT port for that printer. If the B_PRINTER*n* variable is set to DEFAULT or DIALOG, there is no association between the printer number and the LPT port under any environment.

  The cEnumPrinters function may be used to get information about the available printers. Normally, this feature is used from a BASIC program when you need to get information about printers in order to set the B_PRINTER*n* variable to a specific printer name with a call to the Bputenv function. Refer to the description in the *W/32 Functions* chapter for an example.

- If B_PRINTER*n* is equal to DIALOG, then the user is presented with a Print dialog that allows a printer to be selected from the list specified in Print Manager.

  Since you will still be printing ASCII text, there will be some options on the Print dialog such as "Copies" and selecting specific pages with "Page Range" that have been disabled. They will have no effect until Phase 2 is implemented.

  Also note that if you change the default printer with the "Print Setup" option in the Print dialog, the selected printer will appear only during the current run of the program. The change will have no effect in a chained or linked program. The "Network" option is available so you can connect to another printer on the network, and the above rule will still apply.

If the B_WINPRINT variable is not set, the mechanism for printing a file depends on the following:

- Which Windows operating system are you running under?
- Is the printer connected directly or is it a network printer?
- Is it a printer under Windows for Workgroups or another network?

Because there can be so many variables involved, W/32 will execute a batch file to send the spooled file to the printer. The batch file is called **cetprnt*x*.bat**. (in **\cetbin** by default) where the '*x*' corresponds to the printer number. For example, PRINTER and PRINTER1 will both execute **cetprnt1.bat,** PRINTER2 will execute **cetprnt2.bat,** and so on.

The **cetprnt1.bat** file that is included with the compiler only contains comments and will not print a file. To enable printing, you must insert the correct commands to print in your environment. The comments contain two examples of how this may be done, one for Windows and one for Windows NT.

To determine which command should be used, first make sure that you have a printer correctly installed. Then, determine the command required to print from the MS-DOS Command window. For example, if you are running under Windows, send the **cetprnt1.bat** file to the printer by entering:

```
COPY CETPRNT1.BAT PRN
```

Note that the DOS PRINT command should not be used under Windows (as it is a memory resident command), but it may be used under Windows NT.

Once you can print from the MS-DOS command window, edit the **cetprnt1.bat** file and insert the command you used. The sample line for Windows is:

```
REM COPY %1 PRN
```

Delete all of the comments in the file so that it contains the following line:

```
COPY %1 PRN
```

The name of the file to be printed will be passed as a parameter to the batch file (%1 will be replaced by the file name). After modifying the file **cetprnt1.bat**, test this procedure by printing any text file such as **config.sys** by entering:

```
CETPRNT1 \CONFIG.SYS
```

The last step in this process is to create a batch file for each of the available printers.

APPENDIX B

W/32 Function List

All of the C language functions that may be used in a W/32 application have been listed here. Note that the names of the functions that are available in all the CET products begin with the letter "b", while those that are only available with W/32 begin with "c". The chapter containing detailed information about the function has been indicated for your convenience.

Chapter Title

Std W/32 Window Functions
Dlg W/32 Dialog Editor
Misc Built-in C Language Routines

Function	Operation	Std	Dlg	Misc
Bchdir	Changes the current working directory.			√
Bcopy	Copies the contents of one file to another.			√
Bcreate	Creates (or clears) the specified data file.			√
Becho	Echoes the characters during LINPUT and INPUT functions.			√
Berase	Erases the specified data file.			√
Bfilestat	Determines if a file or directory exists.			√
Bgetcwd	Gets the current working directory.			√
Bgetenv	Gets the value of an environment variable set in the w32ap.w32 file.			√
Bkbshift	Gets the state of specific keys.			√
Bnoecho	Turns off the echo of characters during			√

	LINPUT and INPUT functions.			
Bputenv	Sets the value of an environment variable.			√
Brename	Renames the specified file.			√
Bsernum	Gets the serial number from the keyplug.			√
cAddLBoxContents	Sets the contents of the list box.		√	
cAddComboContents	Sets the current value of the combo box.		√	
cAppTitle	Adds a title for the application window.	√		
cAppWindowInfo	Gets the app window's size and location.	√		
cAppWindowMax	Maximizes the main application window.	√		
cAppWindowMin	Minimizes the main application window.	√		
cAppWindowNormal	Restores the main application window to its normal size.	√		
cAppWindowTop	Moves the application window on top.	√		
cCheckItem	Enters/removes check mark for menu item.	√		
cChoiceList	Displays a choice list of sorted items.	√		
cChoiceListNS	Displays a choice list of non-sorted items.	√		
cCloseDIBWindow	Closes a DIB window.	√		
cComboAddString	Adds a string in a combo box.		√	
cComboDelString	Deletes a string in a combo box.		√	
cComboInsString	Inserts a string in a combo box.		√	
cComboSetChoice	Sets the default selection for combo box.		√	
cCreateProcess	Starts a specific process.	√		
cDefaultBtn	Changes the default command button.		√	
cDIBInfo	Gets the height and width of DIB image.	√		

cDIBWindowInfo	Gets the location and size of DIB window.	√		
cDIBWindowTop	Moves a DIB window on top of stack.	√		
cDisplayInfo	Gets screen resolution & number of colors.	√		
cDlgTitle	Sets the title of the dialog.		√	
cEnableCtrl	Enables or disables a control.		√	
cEnableDocking	Enables a docking toolbar.	√		
cEnableItem	Enables or disables a menu item.	√		
cEnableMainWin	Controls activation state of main window.	√		
cEndDlg	Sets the CMDID% to end the dialog.		√	
cEnumPrinters	Gets information about available printers.	√		
cFontValues	Returns the values for a selected font.	√		
cForceExit	Exits a program without an OK message.	√		
cFormBtnClick	Sends an event to the active form.		√	
cGetCheck	Gets the state of a check box or radio button		√	
cGetCombo	Gets the selection from a combo box.		√	
cGetCtrlText	Gets the value of a static (text) control.		√	
cGetCtrlValue	Gets the contents of an interactive control.		√	
cGetCursor	Gets the cursor's column and row position.	√		
cGetEditText	Gets the value of an edit control.		√	
cGetLBoxSelection	Gets the selected ID value from a list box.		√	
cGetLBoxString	Gets the selected string from a list box.		√	
cGetMouse	Gets the current mouse position.	√		
cGetRadioGroup	Gets the selected button from a group.		√	
cGetScrollPos	Gets the current scroll bar position.		√	

cInputQ	Puts a keycode into the input queue.	√		
cLBoxAddString	Adds a string to the end of a list box.		√	
cLBoxDelString	Deletes a string from a list box.		√	
cLBoxInsString	Inserts a string into a list box.		√	
cLBoxSetSel	Sets the selected string in a list box.		√	
cLBoxTabStops	Sets the stops for tab expansion.		√	
cLBoxTopIndex	Sets the first item to appear in a list box.		√	
cLimitComboText	Limits the characters in a combo box.		√	
cLimitEditText	Limits the characters in an edit control.		√	
cLogin	Displays a login dialog box.	√		
cMaxSize	Resizes a window to maximum size.	√		
cMessageBoxOkC	Displays an OK-CANCEL message box.	√		
cMoveAppWindow	Moves the application window.	√		
cMoveDIBWindow	Moves a DIB window to a new location.	√		
cNewMenu	Displays a new menu.	√		
cOnWin32s	Checks to see if running under Windows.	√		
cOpenBrowser	Displays a File Open browser window.	√		
cOpenDIBWindow	Opens a window to display a bitmap file.	√		
cProcessStat	Checks the status of a started process.	√		
cSaveBrowser	Displays a File Save As browser window.	√		
cSelectFont	Changes to the specified font.	√		
cSendBtnClick	Sends a button click.		√	
cSetCheck	Sets the state of check box or radio button.		√	
cSetCtrlText	Sets the value of a static (text) control.		√	

cSetCtrlValue	Sets the contents of an interactive control.		√	
cSetCaret	Sets the cursor position in an edit control or combo box.		√	
cSetCursor	Sets the cursor for the next input operation.	√		
cSetEditText	Sets the contents of an edit control.		√	
cSetFocus	Sets the input focus to a control.		√	
cSetScrollPos	Sets the scroll bar position.		√	
cSetScrollPage	Sets the “page “ value of a scroll bar.		√	
cSetScrollRange	Sets the minimum/maximum values for bar.		√	
cShowCtrl	Hides or displays a control.		√	
cShowEditWrap	Set the inclusion of line-break characters in a multi-line edit control to on or off.		√	
cShowMainWin	Hides or displays the main app window.	√		
cShowStatusBar	Hides or displays the status bar.	√		
cShowToolBar	Hides or displays the tool bar.	√		
cShowVersion	Displays version of runtime components.	√		
cSizeAppWindow	Sets the app window to a specific size.	√		
cStatusText	Displays text on the status bar.	√		
cStringInputQ	Puts a string in the input buffer.	√		
cToolBar	Loads a new toolbar bitmap.	√		
cWaitCursor	Displays or removes an hourglass cursor.	√		
cWinExec	Executes a Windows program.	√		

APPENDIX C

Adding W/32 Features to Text-based Programs

Introduction

This section is designed to show how a text-based application can have the *look* of a Windows program. We purposely kept the number of modifications to the source code to a minimum. In fact, many of the W/32 features that are covered may be implemented by simply modifying the Windows Framework files.

With the exception of the floating edit control that is used to get the input from the operator and the white boxes that mark the inactive input fields, all the other features that are covered in this section will still apply when you decide to create a GUI interface for the program.

A special CLIENT program is used to illustrate how you can add:

- A custom title for the application window.
- A customized set of Windows Framework files.
- A Windows menu bar that may be used in addition to your standard menu.
- A custom tool bar with pop-up tooltips that describe each button.
- The status line to display prompting messages to assist the user in entering the data items.
- Windows message boxes to indicate an error condition.
- A Windows choice list to help the user find an existing record or the available entries for a particular field.
- A call to cWinExec to execute the Windows Notepad program so the user may write comments that will be linked to the current record.
- A box-drawing routine to give the data entry screen a 3-dimensional look.
- Floating edit controls to get input from the user and white boxes to mark the inactive fields.

When you start adapting W/32 features to your own application, you may also consider replacing any text-based windows you may have created (e.g. with Phase One's Window System) with dialog boxes for more of a Windows look.

Note that CLIENT was originally developed under the THEOS operating system using CONTROL 'Plus' from Phase One Systems. It has all of the features that are typicallly found in data entry programs. The source code is very highly structured and easy to read. Remarks have been added to document the features, especially those that pertain to W/32 programs.

This section will document each step that was used to create the CLIENT demo in order to illustrate how you might implement the features in your application.

Installation

The CLIENT program files have been stored in a **.zip** format. To install the files, move the **client.zip** file to the root directory and execute:

```
pkunzip -d client.zip
```

The files will be unzipped into the **\cetlib\samples\client** directory. The W/32 environment variables to be used will be stored as **\windows\client.w32**. If you are using Windows NT, the file **must** be moved into the appropriate **\winnt** directory. All of the other files are listed here for your convenience.

File name	Function
client.exe	Executable
client.doc	This documentation in a WORD format.
scrn\client	ASCII file that contains information regarding the data file to be maintained and the text fields used to display the data entry form.
data\client.idx data\client.dat	Indexed file used to store the client data.
help	Subdirectory containing the help file.
client	Subdirectory containing the description memos written using the Windows Notepad.
res	Subdirectory containing the new program icon and toolbar.
sales	Subdirectory containing the sales memo files written using the Windows Notepad.
cpwin.obj	Modified version of **cetwin.obj**.
cpwin.b	Source code for the **cpwin.obj** file.
cpuser.res	Modified version of **cetwin.res.**
cpuser.rc	Source code for the **cpuser.res** file.
client.b	Source code for the main program.
cpu	The subdirectory used to store the #include files.

Using the W/32 Features in CLIENT

We suggest that you execute the CLIENT program and use some of the features before looking at the source code.

After executing CLIENT, the first thing you will notice is the edit control used to get input from the operator. This is a special compiler feature that may be implemented using the -Zi and -32 flags. White boxes have been used to mark the inactive input fields. Special boxes have also been drawn to give the screen a 3-dimensional Windows look.

To see the other W/32 features that have been implemented, first create a record by entering your company name and pressing Return. A Windows message box will appear to indicate that this is a new record. This text used to be displayed on row 24.

Enter your address and press Return to move to the City field. The prompting message for each field is now displayed in the status bar instead of on row 23. (The ↓ and ↑ keys may also be used to move through the record.)

Also notice how the edit control moves to the next *input* statement. Although CLIENT uses LINPUT USING statements, it does not matter whether you use INPUT, LINPUT, LINPUT USING or GET, the statements will continue to operate as you expect and return the normal INP values.

Now, move the cursor to the Phone1 field, enter 10 digits and press Return. CLIENT uses a special CONTROL 'Plus' feature that automatically validates and formats data such as phone numbers. If you had entered 7 digits, the default area code would have been displayed.

Enter some alpha characters in the phone2 field. The program will detect an error and display the appropriate text in a Windows message box. This is done with a special user-defined function that may be adapted to your application.

Note that the original CLIENT program used Phase One's Memo Writer, a special full-screen editor that allows you to write up to 100 lines of text. The memo file would then be linked to a specific field in the data record.

The W/32 version of CLIENT has been modified to use the Windows Notepad instead of Memo Writer. To see how this feature works, move to the Description field and press Esc to edit the memo. After Notepad is loaded, enter some text, save the file and exit. An 'X' will appear to indicate that a memo has been written. The name of the text file will be stored in the field so that the program can find the memo later.

As you enter the record, notice how most of the toolbar buttons (and their corresponding menu commands) are grayed out. If you decide to add a custom menu and toolbar, you must disable any item that should not be selected during the data entry operation.

After you have entered all the input fields, the cursor will be positioned at the standard menu at the bottom of the screen. Now, all of the menu commands are available except for File-Open.

In the THEOS version, you would typically enter "FI" or press the ↓ key to file the record. For the purposes of this discussion, please select File to continue.

The program should now have cleared the screen and moved you to the Company field. This time, do as the prompt message indicates and press F4 to display a list of the existing company names. Select one of the records. (The same operation may be performed with the File-Open command.)

Before quitting the program, select the Help-On Client menu command to display a brief message about the program. Clicking the "?" button or entering a "?" in the Company field will display the same message.

At this point, we assume that you have finished looking at CLIENT and would like to exit the program. Select the File-Exit command.

The program has been modified to suppress the "Program Interrupted" and "Application Exit" message boxes that would normally have been displayed. The application window will close immediately and you will be returned to the Program Manager.

The CLIENT Environment Variables

Since CLIENT was created originally under the THEOS operating system, the B_EMULATE variable is used in addition to the following variables. All of these variables are set in the customized **w32app.w32** file called **\windows\client.w32**.

```
B_DFLTBGC = 7
B_DFLTFGC = 0
B_EMULATE=1
B_PRINTER1=DIALOG
B_WINPRINT=1
```

The default foreground and background colors have been set to black on white which appears as gray under Windows. It is important to note that the W/32 Windows Framework will use these colors, instead of the default white on black, when initializing the application window. (This happens before the CLIENT program paints the screen using the same colors).

The CLIENT Framework Files

CLIENT has been compiled using a modified copy of the **cetwin.b** file called **cpwin.b**. The BASIC subroutine cetPreInit is used to display a title for the application window and set certain other behavior as the program is initialized. Remarks have been added to indicate what has been done.

```
$SUB cetPreInit
CALL cSelectFont( "Terminal", -12, 400, 0, 255, 49 )
REM  Use Terminal as the default display font
CALL cEnableDocking()              \ REM Toolbar is 'dockable'
CALL cShowToolBar( 1)              \ REM Display toolbar
CALL cShowStatusBar( 1 )       \ REM Display status line
CALL cAppTitle("The Client Demo Program")
$EXIT
```

The cetWINSUB subroutine has also been modified to display additional items on the File and Help menus. When these items are selected, specific key values will be placed into the input queue or buffer just as though the keys were entered by the operator (into the *standard* menu line at the bottom of the data entry screen).

Note that the File-Exit command makes a call to cForceExit to suppress the default "Application Exit" message box. If you have created a Windows help file for your application, you should modify the lines in CASE 1 to display it.

The special variable MOD% is defined as global here and at the top of the $MAIN program (**client.b**) so that the menu selection can be passed back to CLIENT for processing.

```
$SUB cetWINSUB
$GLOBAL CmdID%, MOD%
SELECT CmdID%
    CASE 1              \REM Help
        REM  Use cWinExec to call Winhelp if you have created a
        REM  Windows help for your application.
        REM  CALL cWinExec("WinHelp help\client",1)
        CALL cStringInputQ("?")
        CALL cInputQ(13)
    CASE 2              \ REM New or Cancel
        IF MOD%     \ REM Entered from modification line
            CALL cStringInputQ("CA")
            CALL cInputQ(13)
        ELSE            \ REM Entered from any of the input fields
            CALL cInputQ(3)
            IFEND
    CASE 3              \ REM Open an existing record
```

```
        CALL cInputQ(4)
    CASE 4          \ REM Save (File)
        CALL cStringInputQ("FI")
        CALL cInputQ(13)
    CASE 5          \ REM Delete
        CALL cStringInputQ("DE")
        CALL cInputQ(13)
    CASE 6          \ REM Print
        CALL cStringInputQ("PR")
        CALL cInputQ(13)
    CASE 7          \ REM Exit
        CALL cForceExit
        END
    OTHERWISE
        MSGBOX "Unhandled Command ID"
    CEND
$EXIT
```

None of the BASIC subroutines used to process mouse events have been modified. Implementing mouse support would require more modifications than we felt were necessary for the purpose of this demo.

The CLIENT Resource File

The resource file **cetuser.rc** has been modified to define the custom menu bar and the icon and toolbar (both of which are stored in the **\res** directory). In addition, the text displayed by the Help-About command has been changed to provide information about the CLIENT program.

Note that the File-Exit command has been changed from ID_APP_EXIT in order to eliminate the default "Program Interrupted" message that appears when you exit before the *end* of the program. A call to cForceExit is also made to eliminate the default "Application Exit" message.

```
// Icon
IDR_MAINFRAME   ICON  DISCARDABLE    "RES\\POSICON.ICO"

// Bitmap
IDR_MAINFRAME  BITMAP  MOVEABLE PURE  "CETTOOL.BMP"
IDR_TOOLBAR2 BITMAP  MOVEABLE PURE  "RES\\POSTOOL.BMP"

// Menu
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
   POPUP "&File"
   BEGIN
```

```
      MENUITEM "&New    ",          ID_WINSUB2
      MENUITEM "&Open   ",          ID_WINSUB3
      MENUITEM "&Save   ",          ID_WINSUB4
      MENUITEM "&Delete ",          ID_WINSUB5
      MENUITEM SEPARATOR
      MENUITEM "&Print  ",          ID_WINSUB6
      MENUITEM SEPARATOR
      MENUITEM "E&xit",             ID_WINSUB7
    END
    POPUP "&View"
    BEGIN
      MENUITEM "&Toolbar",          ID_VIEW_TOOLBAR
      MENUITEM “&Status Bar”,       ID_VIEW_STATUS_BAR
    END
    POPUP “Fonts”
      BEGIN
      MENUITEM "Select",            ID_FONTS_SELECT
    END
    POPUP "&Help"
    BEGIN
      MENUITEM "&Help on Client",  ID_WINSUB1
      MENUITEM SEPARATOR
      MENUITEM "&About Client...", ID_APP_ABOUT
      END
END

// Dialog
IDD_ABOUTBOX DIALOG DISCARDABLE  34, 22, 203, 82
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_CAPTION  |
WS_SYSMENU
CAPTION "About Client Version 3.0"
FONT 8, "MS Sans Serif"
BEGIN
    ICON            IDR_MAINFRAME,IDC_STATIC,11,17,18,20
    LTEXT           "\nClient Demo Program Vers 3.0\n\nCopyright \251
Phase One Systems 1996.\nAll right reserved.",
                    IDC_STATIC,40,7,152,54
    DEFPUSHBUTTON       "OK",IDOK,99,64,32,14,WS_GROUP
END

//   String Table
//  The strings displayed for the Help-About, Exit, Toolbar, Status line
//  and Fonts commands aren’t included since they haven’t changed.
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME  ""
```

```
END
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   AFX_IDS_APP_TITLE   "Client Application"
END
// Add ID_WINSUB prompts here...
//
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
   ID_WINSUB1   “Help on Client\nHelp on Client"
   ID_WINSUB2   "Cancel current record\nClear screen for new record"
   ID_WINSUB3   “Open new record\nOpen record"
   ID_WINSUB4   "Save current record\nSave record"
   ID_WINSUB5   “Delete current record\nDelete record"
   ID_WINSUB6   "Print current record\nPrint record"
   ID_WINSUB7   "Quit the application\nExit"
END
```

The CLIENT Source Code

Although the W32APP program has been used to create all of the executables for the other W/32 sample programs, we chose not to use this feature in order to simplify the discussion.

CLIENT has been compiled with the following command. The -wr and -wo flags indicate that the modified resource files should be used during compilation. The -32 and Zi flags have been used to implement the floating edit control feature.

```
obwin -o client client.b -wr cpuser -wo cpwin -32 -Zi
```

If you print the source code **client.b**, note that:

- Changes made for W/32 are indicated by “rem *** pd” statements.
- Special subroutines have been added with #include statements. This code has been included in the manual.
- The lines in the source code are unnumbered so that it is easier to modify (and merge the #include files during compilation).

The CLIENT source code may be used as a guide to adding Windows features in your application. It is a highly structured program with all of the features you would find in a typical data entry program. It contains:

Main Routine	Function
INITIALIZE	Initializes the program variables. Defines the error handling routine, function/control key values, status line printing, 2 and 3-

	button dialog boxes, and the 3-D box display. It also defines the choice list displayed during record lookup.
INP.SCRN	Sets the name of the screen file.
READ.SCRN	Reads the screen file (**scrn\client**). Dimensions and loads the arrays used to display the data entry form, format, validate and store the data and draw the 3-D boxes.
OPEN.FILE	Opens the data file.
PAINT.SCRN	Paints the text and white boxes for the input fields.
EXECUTIVE	Calls the routines to read, delete, update and write the data records. Edits memos, if any.
EXIT	Processes any 'pending' memos, closes the data file and exits.

Although the modifications that apply to W/32 programs are emphasized, all of the routines are briefly described to give you a general understanding of how CLIENT works and make it easier to implement the new features.

Please contact Phase One Systems if you have any questions about CLIENT or any of the other W/32 sample programs that are included with the product. Phase One may be reached at 510-895-0832 (voice) or at 510-895-0828 (fax).

The Top of the Program

BASIC programs may be compiled with the default W/32 framework files without making any modifications to the source code unless you want to add more Windows features..

In the case of CLIENT, the menu commands in the BASIC subroutine called cetWINSUB (in **cpwin.b**) were modified to operate as though the user had entered an option on the menu line at the bottom of the data entry screen. The variable MOD% was defined as global in order to pass the user's selection back to the *main* program for processing.

A $MAIN statement is entered on the first unremarked line in **client.b** to indicate that it is the main source module. Next, a $GLOBAL statement is added to define the data (MOD%) that is to be shared between the modules.

```
$MAIN
$GLOBAL MOD%
```

You will also find statements to call the major program routines. Each of these routines will be documented in the following sections.

```
GOSUB INITIALIZE
GOSUB INP.SCRN
GOSUB READ.SCRN
GOSUB PAINT.SCRN
GOSUB OPEN.FILE
```

```
GOSUB CLEAN.SCRN
GOSUB EXECUTIVE
```

The Initialization Process

The top of the INITIALIZE routine defines the error handling routine. In a *real* application, this routine (or one defined with an ON LOCK statement) should be modified to process file/record lock conditions. (The variable B_THLOCK would need to be set off in order for record lock errors to be detected.)

```
ON ERROR GOTO ERROUTINE
```

This routine also initializes the program variables and defines the special pattern masks used to validate and format the phone number and zip code fields. None of this code was modified.

INITIALIZE contains a number of #include statements to define the files that contain the code created to implement the W/32 features used in CLIENT. The contents of the #include files are documented here for your convenience.

```
#include "cpu\cpstdbar.cpu "
#include "cpu\colors.cpu "
#include "cpu\fnbox.cpu "
#include "cpu\fnprompt.cpu "
#include "cpu\disperr.cpu "
#include "cpu\browseky.cpu "
```

cpstdbar.cpu

The cetPreInit subroutine in **cpwin.b** has been modified to display the toolbar ("call cShowToolBar(1)") when the program is initialized. The **cpuser.rc** file has also been modified to set IDR_TOOLBAR2 to the name of the custom toolbar which should be used (**res\postool.bmp**).

The #include "cpstdbar.cpu" statement inserts the code to dimension and load the BTN% array. After the CALL to cToolbar is performed, the new toolbar will be used instead of the default provided with the W/32 App Builder. Note the special button value of 0 is used as a SEPARATOR to add space between the buttons.

```
DIM BTN%(10)
BTN%(1)=7       \ rem Program exit
BTN%(2)=0
BTN%(3)=2       \ rem New (cancel) record.
BTN%(4)=3       \ rem Open record. A list of existing keys will appear.
BTN%(5)=4       \ rem Save Record
BTN%(6)=5       \ rem Delete Record
BTN%(7)=0
```

```
BTN%(8)=6          \ rem Print Record
BTN%(9)=0
BTN%(10)=1         \ rem Help
CALL cToolBar(2,10,MATADDROF(BTN%))
```

colors.cpu

This code defines the colors that may be used. Since CLIENT uses THEOS emulation, only 8 are available. The DefColor$ variable is also set so that it may be used later to display the default colors for the input fields.

```
BLACK% = 0  \  BLUE% = 1  \  GREEN% = 2  \  CYAN% = 3
RED% = 4  \  MAGENTA% = 5  \  YELLOW% = 6  \  WHITE% = 7
DefColor$ = crt$("BC"&str$(white%))&crt$("FC"&str$(black%))
```

fnbox.cpu

This file contains a special user-defined function created by Resolutions in Software. It is called (from PAINT.SCRN) to display the 2-color boxes that give the screen a 3-dimensional look. Please print the file if you wish to see this code.

fnprmpt.cpu

This file contains the user-defined function which is used to display the prompting messages for the individual data items on the status line. In CLIENT, these messages are read from an external screen file in the READ.SCRN routine.

```
DEF FNPRMPT.ERS$(MSG$)
    CALL cStatusText(MSG$)
    FNEND
```

disperr.cpu

This file contains the user-defined function which is used to display a Windows message box whenever an error is detected. Previously, error text was displayed on row 24.

```
DEF FNERR.ERS$(DISPERR$)
    PRINT CRT$("BELL");
    MSGBOX DISPERR$
    FNEND
```

browseky.cpu

This file contains a routine which displays a Windows choice list box with the keys to the existing records. The operator's selection is passed back to the program where it is used to read the desired record and display it.

The routine is called when the File-Open command is selected or when an F4 key has been entered in the key (Company) field. The name of the data file to be opened is read from an external SCRN file and assigned to the DATA.DESC$ variable.

```
BROWSE:
    OPEN #19: DATA.DESC$, INPUT INDEXED
    rem Determine record count to set size of choice list.
    call Bindinfo(19,addrof(rec.count),addrof(a%),addrof(b%))
    DIM in.keys$(rec.count)
    choice.count% = 0
    WHILE NOT EOF(19)
        READNEXT #19, temp$: dum$
        IF NOT EOF(19)
            choice.count% = choice.count% + 1
            in.key$(choice.count%) = temp$
            IFEND
        WEND
rem Array for choice list cannot be smaller than the number of items
rem to be displayed.
    DIM choice.list$(choice.count%)
    MAT choice.list$ = in.key$
    CLEAR in.key$
    selected$ = ""
    call cChoiceListNS("Select Record",mataddrof(choice.list$),
addrof(selected$))
    IF selected$ <> ""
        rem A selection has been made…
        rem Perform the operations required before a new record can
        rem be read and displayed. Since this code is unique to the
        rem CLIENT program, it has been omitted from this document.
.
.
.
    IFEND
CLEAR choice.list$,selected$
CLOSE #19
RETURN
```

The Screen Painting Routines

A brief description of the operations performed in these routines is provided to assist you in understanding where you would need to make changes to implement the new W/32 features. Normally, only the modified code has been included.

INP.SCRN

Data entry programs display an input screen designed to maintain one or more specific data files. CLIENT is a typical CONTROL 'Plus' program that does not contain any information about a specific data file or screen form. All of these specifications are read from an external ASCII screen (SCRN) file as the program is executed.

INP.SCRN handles any command line arguments that may have been used and sets the SCRN.DESC$ variable to the name of the file that contains the data file and screen specifications.

READ.SCRN

This routine reads the screen file assigned in INP.SCRN to get the name of the data file to be maintained and assign the variables used to store the number of fields for the key, data, display text and boxes. These variables are then used to dimension and load the appropriate arrays.

The only modification to this routine was to use the new FNERR.ERR$ function to display the error text in a Windows message box instead of on line 24.

```
IF EOF(1) THEN PRINT FNERR.ERS$("Corrupted screen file. Program
aborted."); GOTO EXIT
```

If you are interested in drawing 2-color boxes to give your screens a 3-dimensional look, note how this is done in CLIENT. The arrays that were dimensioned to the defined number of boxes (BOXCNT%) in INITIALIZE are now loaded from the values stored in the screen file. Later, in PAINT.SCRN, the FNBOX function will be used to draw the boxes.

```
DIM BTYPE%(boxcnt%),BULC%(boxcnt%),BULR%(boxcnt%),BLRC%
(boxcnt%),BLRR%(boxcnt%)
FOR I% = 1 TO BOXCNT%
    INPUT #1: BTYPE%(I%),BULC%(I%),BULR%(I%),BLRC%(I%),
BLRR%(I%)
    NEXT I%
```

CLIENT uses a number of arrays to determine how to read, validate, format and display the data fields. The values loaded for the Company field are included here only to make the CLIENT program more *readable*.

Function	Array Name	Value for Company
Field lenth	LEN%	30
Prompt the user for input	PRMPT$	Y
Modification symbol	MODC$	N/A in this version
Starting column position	COL%	22

Starting row position	ROW%	4
Output conversion/format	FMT$	A
Prompting text	TXT$	Enter the company name.
Case mode	CS$	U
Trim required	TRM$	Y
Required field	REQ$	Y
Default data	DFT$	(None)
Starting string	OFMT$	(None)
Pattern masks...		

When you used the CLIENT program earlier, you may have noticed the special validation and formating for the phone fields. These fields have a starting string of "(000) 000-0000". Built-in pattern masks are provided (in the INITIALIZE routine) so you can enter either 7 or 10 digits that will be mapped over the starting string and displayed in the correct format.

PAINT.SCRN

The original routine has been replaced with two #include statements. We suggest that you adapt this code to your application if you plan on using the floating edit control feature and display white boxes for the inactive input fields.

```
#include "cpu\paintscr.cpu"
#include "whitefld.cpu"
```

paintscr.cpu

This file contains the new PAINT.SCRN routine that sets the screen to the default colors, draws the 2-color boxes with the FNBOX function and displays all the text on the screen. At the end of the routine, a GOSUB WHITE.FIELDS is performed.

```
PAINT.SCRN:
    PRINT crt$("KOFF");DefColor$;CLS;
    FOR I% = 1 to boxcnt%
        PRINT fnbox$(BULC%(I%),BULR%(I%),BLRC%(I%),BULC%
(I%),BLRR%(I%),BULR%(I%)+1,NOT (BTYPE%(I%)*-1));
        NEXT I%
    FOR I% = 1 TO DISPFIELD%
        rem Display the text fields on the screen.
        PRINT AT$(DCOL%(I%),DROW%(I%));DTXT$(I%);
        NEXT I%
    GOSUB white.fields
    RETURN
```

whitefld.cpu

This file contains the code which is used to paint white boxes wherever an input field exists. These white boxes will be covered up later with a floating edit control when the cursor is positioned in the field.

```
WHITE.FIELDS:
    FOR I%=1 to totfield%
        PRINT at$(col%(I%),row%(I%),crt$("bc"&str$(black%));space$
(len%(I%))
        NEXT I%
    PRINT DefColors$;
    RETURN
```

In the **colors.cpu** file, the DefColors$ variable was defined as:

```
DefColor$ = crt$("BC"&str$(white%))&crt$("FC"&str$(black%))
```

The Open File Routine

The OPEN.FILE routine opens an indexed file with the name assigned in READ.SCRN and performs a RETURN.

The EXECUTIVE Routine

This is the main routine in the program. From here, calls are made to other routines to read, delete, update and write records in the data file.

Modifications have been made to the program so that a white box is painted *behind* the data after a Return or ↓ key has been entered to terminate the input. (The floating edit control has also moved to the new *input* statement.) In CLIENT, this was done by performing a GOSUB DISP.FIELD (in the code inserted with a #include statement at the end of the file.)

```
FOR FIELD% = SFIELD% TO TOTFIELD%
    IF PRMPT$(FIELD%)="Y"
        rem If the user must enter the data...
        GOSUB GET.FIELD
        rem User has moved to a previous field.
        rem ***pd Redisplay field after floating edit control passes it.
        GOSUB DISP.FIELD
```

If you look through the code, note that GOSUB GET.KEY gets the data in the key, attempts a trial read on the file and calls GET.REC (to assign the *key* variables and call READ.RAND). If a record exists, it is displayed in DISP.REC. Otherwise, a new record is created by calling GET.FIELD for each data field.

After all of the fields have been entered, a call is made to MODIFICATION where the appropriate action is taken to file, delete, exit, etc.

The CLEAN.SCRN Routine

This routine is called when a user decides to cancel the record that is being input and start entering a new one. Previously, the program cleared the data from the screen when "CA" had been entered on the menu line or Ctrl+C had been pressed. In the W/32 version, selecting the File-New command will have the same effect.

The original routine has been replaced with a GOSUB WHITE.FIELDS to repaint the white boxes on the screen. (This routine is stored in **whitefld.cpu**.)

The PRINT.SCRN Routine

This routine is called when the operator enters "PR" on the menu line or selects the File-Print command. In either case, the currently displayed record will be printed.

As soon as the printer channel is closed, a Print Dialog will appear so that the user may select from any printer on the list. This happens because the following variables were set in the **client.w32** file. No modifications to the code were necessary.

```
B_PRINTER1=DIALOG
B_WINPRINT=1
```

The GET.KEY Routine

This routine is called whenever a user attempts to enter the key to a record. If no record is currently displayed, an ↑ or ↓ key may be used to get the next or previous record in the file.

When a key value is entered, the program attempts to find a record with a matching key. CLIENT has been modified so that if one can not be found, "New Record" is displayed in a Windows message box instead of on line 24.

Additional modifications have been made so that users can click on the File-Open command or press F4 to display a Windows choice list of existing keys with a GOSUB BROWSE. (The BROWSE routine is found in **cpu\browseky.cpu**.)

It is important to notice that the File menu commands are disabled and/or enabled at the appropriate time so that the user can not perform an operation that could *damage* the data.

If you plan to use a memo field, a call should be made to MEMO.PEND to see if any special memo processing needs to be done before a new record is displayed.

Code that does not apply to the W/32 version has been included. Please refer to **client.b** if you wish to see all of the operations that are performed.

```
GET.KEY:
    rem ***pd
    CALL cEnableItem(1,1)  \  rem Enable Help
    CALL cEnableItem(3,1)  \  rem Enable Open
    CALL cEnableItem(2,0)  \  rem Disable New
    GOSUB MEMO.PEND
    rem  Perform operations to assign the values for the key to the last
    rem  record displayed, process the entry of a control key (e.g a ↓ or
    rem  ↑ or Ctrl+C key)
    .
    .
    .
    FOR FIELD%=1 TO KEYFIELD%
        GOSUB GET.FIELD
        rem Get the input. Then, validate and format the entry.
        .
        .
        .
    IF LOOKUP%      \rem Lookup (open) an existing record
        GOSUB BROWSE
        LOOKUP%=0
        IF FOUND%<>0 THEN FOUND%0 \ GOTO GET.KEY.RET
        IFEND
    NEXT FIELD%
    .
    .
    .
```

The GET.FIELD Routine

This routine is called whenever a data field is to be entered or modified. Modifications have been made to support the Windows menu bar by disabling all the File commands that should not be available while the operator is entering data. Note that the File-New command is enabled. Selecting this item would be the same as canceling the current record by entering "CA" at the menu line.

The IF-IFEND structure will disable or enable specific menu commands if the field is not a key. The second group of menu commands will always be disabled.

```
GET.FIELD:
    get.field%=1
```

```
IF field%>keyfield%
    CALL cEnableItem(1,0)  \  rem Disable Help
    CALL cEnableItem(3,0)  \  rem Disable Open
    CALL cEnableItem(2,1)  \  rem Enable New
    IFEND
CALL cEnableItem(4,0)  \  rem Disable Save
CALL cEnableItem(5,0)  \  rem Disable Delete
CALL cEnableItem(6,0)  \  rem Disable Print
.
.
.
```

The GET.FIELD routine was also modified to display the prompt message for the individual fields in the status bar instead of on line 23.

```
PRINT FNPRMPT.ERS$(TXT$(FIELD%));
```

Errors detected while entering the data will now be displayed in a Windows message box.

```
IF err.flg% THEN PRINT fnerr.ers$(err.msg$); \ GOTO GET.FIELD
```

If you are interested in adding memo fields to your application, note that we test the field format for an "M" to determine whether a memo field is to be input. If it is, the input is processed and conditional calls are made to the appropriate memo routines.

CLIENT always stores the initial field value in the NDATA$ array no matter what format the data field has. The temporary variable AN$ is formatted and tested before it is loaded back into the NDATA$ array at the end of GET.FIELD.

The FIELD.FMT Routine

This routine is called to format the input variable AN$ according to the output format (date, alphanumeric, numeric, dollar, or memo) and load DISP.FIELD$. A SELECT-CASE structure is used to perform extensive data validation tests and format the field using a pattern mask, if any.

No special changes have been made to implement a Windows feature.

The MODIFICATION Routine

This routine is called to process the menu options. In the THEOS version, the operator would enter a 2-character code on the menu line at the bottom of the screen to indicate which action should be performed. For example, entering "DE" would delete the record. A SELECT-CASE structure conditionally performs the specified operation on the currently displayed record.

The W/32 version of CLIENT not only allows the operator to continue using the menu line, but also provides commands on the Windows menu bar as an alternative method. In either case, the user's selection is assigned to the variable MOD%. (Remember that this variable has been defined as $GLOBAL so that a File menu selection will be passed back to CLIENT for processing.)

CALL statements were added to the top of the routine to enable and disable the File menu commands before the menu line is printed on the screen. The rest of the code is not related to a specific W/32 feature so it has not been included.

```
MODIFICATION:
    MOD% = -1
    WHILE MOD%
        CALL cEnableItem(1,1)  \  rem Enable Help
        CALL cEnableItem(2,1)  \  rem Enable New
        CALL cEnableItem(4,1)  \  rem Enable Save
        CALL cEnableItem(5,1)  \  rem Enable Delete
        CALL cEnableItem(6,1)  \  rem Enable Print
        CALL cEnableItem(3,0)  \  rem Disable Open
        ERR.FLG% = 0 \ UNMOD% = 0
        PRINT FNPRMPT.ERS$("");
        PRINT PRMPT.ERS$;"<mod symbol>, 'CA'ncel, 'De'lete, 'FI'le,
'PR'int or 'QU'it";at$(63,PAGE(0));
```

The FIELD.MOD Routine

This routine is used to process the entry of a *mod symbol* on the menu line. If the entry is invalid, the following line will be executed to display the specified text in a Windows message box:

```
PRINT FNERR.ERS$("Invalid modification Entry.");
```

If you are using the floating edit control feature with white boxes to mark the inactive fields, it is important to note that after performing a GOSUB GET.FIELD (to modify the specified field), a GOSUB DISP.FIELD is performed to redisplay the data in a white box:

```
GOSUB GET.FIELD
rem Redisplay field after floating edit has passed it.
GOSUB DISP.FIELD
```

The DISP.FIELD routine is found in the **cpu\dispfld.cpu** file that is defined with a #include statement at the bottom of the program. Note that most of the lines that are unique to the CLIENT program have not been included here.

```
DISP.FIELD:
    .
    .
```

```
        .
        GOSUB FIELD.FMT \ rem to validate/format the input
        IF ERR.FLG% THEN DISP.FIELD$ = NDATA$(FIELD%)
        PRINT AT$(col%(field%),row%(filed%));crt$("bc"&str$(white%));
crt$("fc"&str$(black%));rpad$(disp.field$,len%(field%));
        PRINT defcolors$;
        FIELD% = FD%
        RETURN
```

The DISP.REC Routine

This routine is called when a record with a key that matches the user's entry is found. The data fields are displayed exactly as if they were just entered at the keyboard. That means that each field value is temporarily assigned to AN$ and passed to FIELD.FMT for format and pattern match checking. The resulting variable DISP.FIELD$ is then displayed.

Note that modifications have been made to use the new window colors.

```
DISP.REC:
    FOR FIELD% = 1 TO TOTFIELD%
        AN$ = NDATA$(FIELD%)
        ERR.FLG% = 0
        GOSUB FIELD.FMT
        IF ERR.FLG% THEN DISP.FIELD$ = NDATA$(FIELD%)
        rem Paint white input fields as record is displayed
        PRINT AT$(COL%(FIELD%),ROW%(FIELD%));CRT$
("BC"&STR$(WHITE%));CRT$("FC"&STR$(BLACK%));RPAD$(DISP.FI
ELD$,LEN%(FIELD%));
    NEXT FIELD%
    PRINT defcolors$ \ rem Reset default colors.
RETURN
```

Except for using FNERR.ERS$ to display errors in a Windows message box, the only other modications made to CLIENT are in the memo routines. The rest of the routines have been described here only to make the program more readable.

The READ.RAND Routine

READ.RAND reads a record from a random access file and loads the fields into an array called NDATA$. Most of this code involves handling the READNEXT and READPREV conditions.

The SHUFFLE Routine

This routine loads the array used to store the data (NDATA$) with the fields that were read from the data file. Special code is included to align the data elements

when a key is made up of separate fields concatenated with an asterisk character. (This is a feature used in CONTROL 'Plus' programs.)

The CHECK.KEY Routine

This routine is called before writing or deleting a record in order to determine whether or not the key value has been changed. An error message box is now displayed when an invalid key was detected, but no other modifications have been made to implement a W/32 feature.

The WRITE.REC Routine

WRITE.REC first checks to see if the current key value has changed. If so, a call is made to DELETE.REC to delete the old record before the new record is written. No special modifications have been made to the W/32 version of CLIENT.

The DELETE.REC Routine

This routine deletes a record. If the record contains a memo field, a call is made to the appropriate memo routine to delete the associated memo file.

The ERROUTINE Routine

This routine currently prints the error message and performs a RESUME 0. In a *real* application, we recommend that you define special processing to be used when a record lock condition is detected. (Note that locked records may only be detected if B_EMULATE or B_THLOCK is not set.)

The following is a sample general-purpose routine. ON LOCK and ON INTERRUPT may be used if you wish to define separate error handling routines.

```
ERROUTINE:
    SELECT err
        CASE 1
            rem Process the system interrupt key
        CASE 48
            rem locked record
            call cMessageBoxOkC(" Record in use. OK to Retry ?",
addrof(rc%))
            IF rc% = 0
                THEN SLEEP 1 \ RESUME
            ELSE
                eof.flag% = -1
                RESUME executive
            IFEND
    CEND
```

```
ON ERROR GOTO 0
RESUME
```

The HELP Routine

This routine displays a help message when "HELP" (or a ?) is entered as a command line argument or when the Help-On Client command is selected from the menu or toolbar.

The MVUP Routine

This routine is called whenever an ↑ key is entered in a data field. When detected, the cursor will be repositioned in the previous field.

The MEMO Routines

The THEOS version of CLIENT uses a special feature called Memo Writer to allow users to take up to 100 lines of notes without leaving the data entry program. Under THEOS, a full-screen editor is provided along with special code used to file the memo with a unique name. This name is stored in the data record so that the memo could be called up later and modified.

The W/32 version of CLIENT has been modified to use the Windows Notepad program. All of the special code to process the memo fields is stored in an external file which is included at the bottom of the program with the statement:

```
#include cpu\cetmemo.cpu
```

Contact Phase One Systems if you have any questions about implementing this feature in your application. Phase One may be reached by calling (510) 895-0832 or by fax at (510) 895-0828.

The EXIT Routine

The code for the EXIT routine is actually located near the top of the program. It has not been modified.

```
EXIT:
    PRINT CRT$("c");
rem Check for an action to be performed on a memo file before exiting.
    GOSUB MEMO.PEND \ CLOSE #1
    END
```

APPENDIX D

Converting a Text-Based Program

Introduction

In the previous section, we used the CLIENT demo to show how W/32 features may be added to a text-based program to give it a Windows look without making major modifications to the BASIC source code.

Now, we will create a GUI interface for the demo program called CLIENT32. We assume that you are already familiar with the features covered in the *W/32 Dialog Editor* chapter. While it is not imperative to read about what was done to create the CLIENT demo, it may be easier to understand the changes we have made in CLIENT32 if you glance through the description of the original source.

The major emphasis in this section is on converting a *linear*, text-based program to an *event-driven* program with a GUI interface that could be used as a *model* to simplify the conversion of the rest of your application.

In this section, we will discuss how to

- Modify the Windows Framework files that were created for the CLIENT demo in the previous section.
- Use the Import Dialog Template feature to convert the CLIENT input screen so that it can be loaded into the dialog editor.
- Modify the controls to take advantage of the W/32 features.
- Modify the source code to include the standard data entry operations such as record lookup and data validation/formatting that would be used throughout an application.

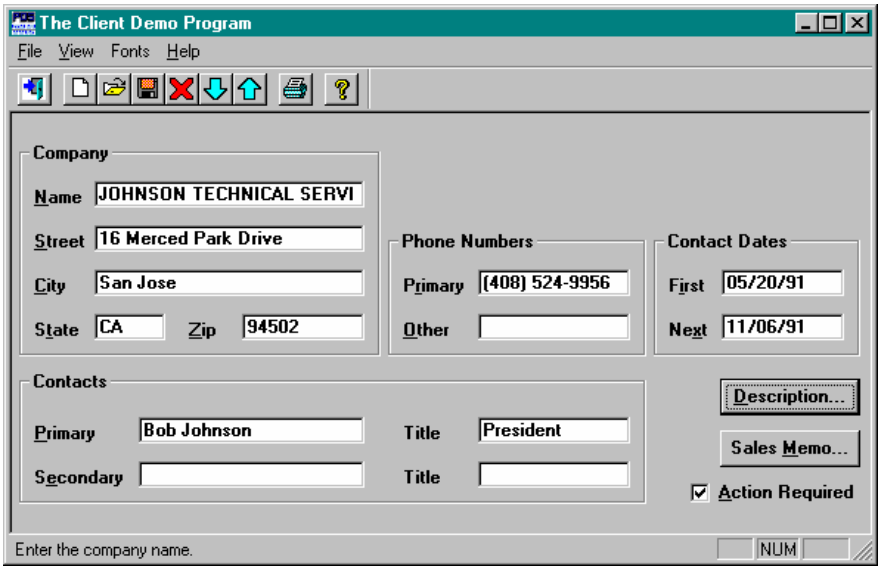
Once you have created your own model program, it should be mainly a matter of cutting and pasting code (or inserting #include statements) from your model form to your new one. This process has been simplified since multiple source files may be open at the same time.

The advantages of using this procedure are two-fold. Afterwards, you will have

- ✓ A GUI interface that looks and operates the way users expect.
- ✓ A highly structured, event-driven program that is easy to maintain.

The CLIENT32 Demo Program

In this section, we will document the procedure we used to convert the text-based CLIENT program to use the GUI input screen shown below.



Note that all file operations are handled from the File menu items and their corresponding toolbar buttons. This procedure lends itself to the *model program* approach because the same Windows Framework files (and BASIC source routines) may be used for a number of different programs (e.g. all those in a particular module or the main program in every module or …).

On the other hand, if you create individual button controls to handle the file operations like we did in the INVOICE demo, they will have to be reentered in each of your programs.

Modifying the Windows Framework

The procedure for creating customized Windows Framework files is covered in detail in the last section. Here, we will only cover the minor modifications that needed to be made to the files used for the CLIENT demo.

First, we copied the **cpwin.b** file to **cpwin32.b**. We then removed all of the code for the menu items that handle file operations in the $cetWINSUB subroutine. Previously, W/32 functions were used to place characters into the input buffer just as if the operator had entered the characters into the menu displayed at the bottom of the text-based screen.

The subroutine was also modified to execute the Winhelp program to display a help message.

```
$SUB cetWINSUB
$GLOBAL CmdID%, cp.flag%
    SELECT CmdID%
    CASE 1 \ rem Help
        call cWinExec("winhelp help\client",1)
    CASE 7 \ rem Exit
        call cForceExit
        END
    OTHERWISE
        cp.flag% = CmdID%
        call cFormBtnClick(cp.flag%)
        REM Pass the toolbar button or menu item click
    CEND
$EXIT
```

If a menu item or toolbar button (other than Help or Exit) is selected, the variable CP.FLAG% is set to the value of CmdID% and defined as a $GLOBAL variable here and in the $MAIN program. In a text-based application such as CLIENT, that is all you have to do in order to be able to process the user's selection in the main program.

In a program that uses a form view, you also have to define the $GLOBAL variable in the form and make a call to cFormBtnClick (or cSendBtnClick) to send the button click (to the form). (If you remark out the call statement and recompile, you will find that nothing happens when you click on an item.)

A new resource file **cpuser32.rc** was made by modifying a copy of the **cpuser.rc** file. Two new items were added to the File menu to display the next and previous record. The name of the new toolbar was also specified to display the corresponding ↓ and ↑ toolbar buttons. (In the text-based version of CLIENT,

these operations were performed by entering the arrow keys in the key field or at the menu line.)

```
// Bitmap
//
IDR_MAINFRAME      BITMAP  MOVEABLE PURE   "CETTOOL.BMP"
IDR_TOOLBAR2 BITMAP MOVEABLE PURE "RES\\POSTOOL2.BMP"
//////////////////////////////////////////////////////////////////////////
//
// Menu
//
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New",                ID_WINSUB2
        MENUITEM "&Open",               ID_WINSUB3
        MENUITEM "&Save",               ID_WINSUB4
        MENUITEM "&Delete",             ID_WINSUB5
        MENUITEM "Ne&xt",               ID_WINSUB8
        MENUITEM "P&rior",              ID_WINSUB9
        MENUITEM SEPARATOR
        MENUITEM "&Print",              ID_WINSUB6
        MENUITEM SEPARATOR
        MENUITEM "E&xit",               ID_WINSUB7
    END
```

Converting a Text-based Input Screen Form

The File-Import Dialog Template command from within the W32APP program may be used to convert your input screens so that they can be loaded into the dialog editor.

This is a special feature that allows you to build dialog *templates* from images of your text-based, data entry screens so you do not have to start from scratch. All text fields will be converted into text controls followed by an optional edit control. At that point, you can rearrange and modify the controls to take advantage of the other features that are available.

Please refer to the *W/32 Dialog Editor* chapter for details on using this feature. In this section, we will only cover the procedure we used to convert the CLIENT input screen.

Creating the dialog template was a simple 2-step process.

1. First, we created an ASCII text file to store the screen image to be converted. This was done by executing the CLIENT program with a special argument:

   client -cetsd

   As soon as the input screen was displayed, we pressed the Alt+PageDown keys (at the same time) to write the image to the file **cetsd001.txt** in the current working directory. We did not bother to edit this file since we felt that the changes we wanted to make to the text fields could just as easily be done after the template was created.

2. Next, we executed W32APP and ran the File-Import Dialog Template command. After selecting **cetsd001.txt**, the program converted and loaded the file with the default dialog properties. All the text items were converted to text controls.

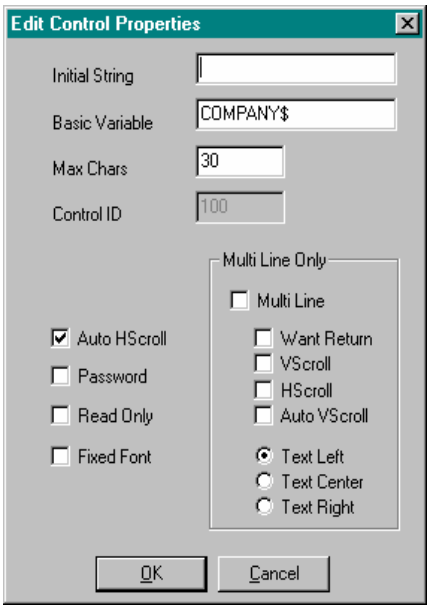
Modifying the CLIENT32 Form

After a screen is converted into a dialog you can easily rearrange the controls on the form and modify them to take advantage of the new control properties. To finish the CLIENT form, we:

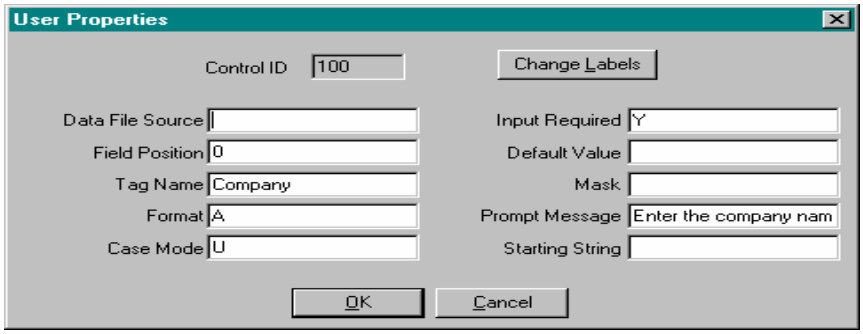
1. Modified the text controls by removing the field number and adding an ampersand character before the letter to be used as the shortcut key. This is an optional feature that makes it easy to move to the associated input control from anywhere on the screen.

2. Added the edit controls. We chose not to convert the input fields with the File-Import Dialog Template command since we planned on using the Edit-Copy and Paste commands instead.

   We started with the edit control associated with "Name". This control was defined with the properties shown here.



The user-defined properties defined for Name are shown below. The labels for each property have been modified to meet our particular needs. They may be changed to anything you wish.



Each interactive control will have a set of user-defined properties. In CLIENT32, most of the properties are used to validate and format the input fields. One of them defines the prompt message which will be displayed on the status line.

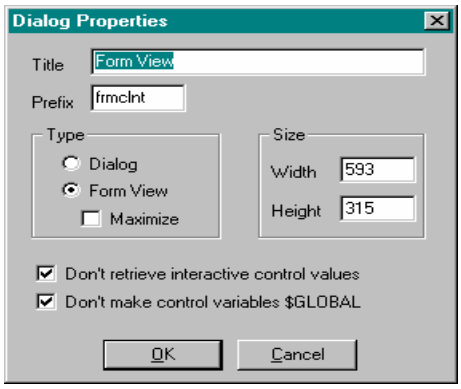
After defining the edit control for "Name", we used the Edit-Copy and Paste commands to create the rest of the edit controls we needed. Run W32APP and load the dialog file **frmclnt.idh** if you wish to look at the definitions for the other controls that were used.

In the case of CLIENT32, we decided to create command buttons to call up Notepad in order to write a memo regarding the (company) Description and the sales activity. A check box control was used to indicate whether any action is required.

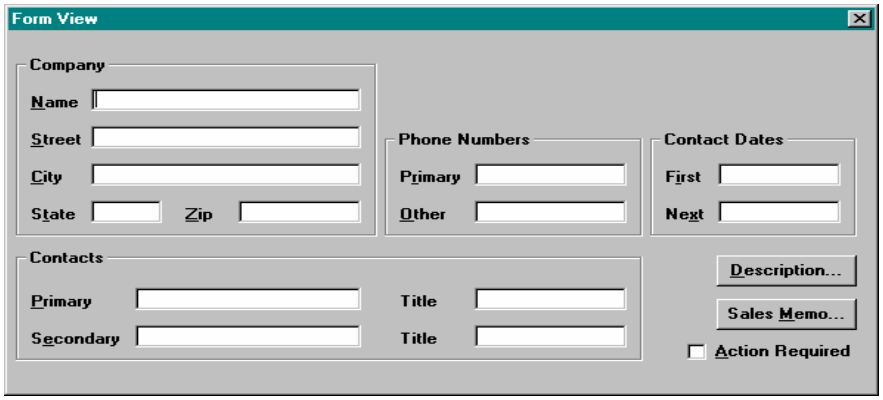
3. After the controls were created, the alignment commands on the Layout menu were used to line everything up properly.

4. Group controls were added to create the same effect as the 3-D boxes in the text-based version.

5. The default properties for the form were changed to the ones shown here.

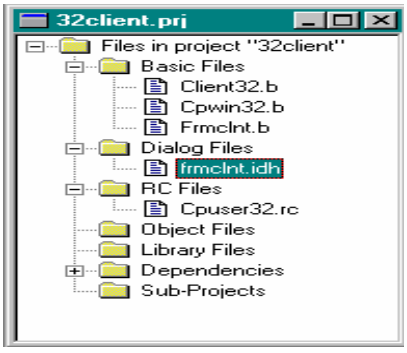


The final form view is shown below.



Creating a Project for CLIENT32

W32APP was used to generate the CLIENT32 program. An empty project was created, then all the files shown in the tree view to the right were added manually using the Add menu.



When the project was built the first time, the files inserted with #include statements in the **frmclnt.b** file were added as dependencies. These files have not been shown here as the list is quite long.

Creating the $MAIN Program

The $MAIN program simply defines the $GLOBAL variable, adds the title to be displayed and calls the form view.

```
$MAIN
$GLOBAL cp.flag%
call cAppTitle("The Client Demo Program")
CALL frmclnt
END
```

Modifying the Source Code for the Form View

The next step is to convert the *linear*, text-based program to an *event-driven* program which may be used as a model to simplify the conversion of the rest of your application.

All the useable code has been pulled out of the original source file and modified, when necessary. If the code is unique to CLIENT32, it has been hard-coded into the BASIC program (**frmclnt.b**). Code that is common to the rest of the application programs has been stored in external files and inserted with #include statements. Making these changes has been simplified since W32APP allows you to have multiple files open at once.

Important Note: Although your programs will look a lot different than the original CLIENT, each subroutine in the source code has been documented to explain why we made the change so that you can adapt the procedure to meet your own needs.

Top of the Program

The variable CP.FLAG% has been defined as global in the first dialog save area so that the form program can detect when a menu item or toolbar button has been selected from the cetWINSUB subroutine found in the **cpwin32.b** file.

```
REM@# Dialog frmclnt V2.3
REM@# 1ststart
$GLOBAL cp.flag%
REM@# 1stend
#include "frmclnt.IDH"
```

The Main Subroutine

The main subroutine contains the code that is normally found in an initialization routine. In the text-based CLIENT version, the screen and data file specifications are read from an external SCRN file. The following modifications have been made in order to create a model program that does not require an external file. These changes not only allow us to use other code from the original file, but they make it so our model can more easily be adapted to other form views.

1. The number of key fields (keyfield%), data fields (dtafield%) and total number of fields (totfield%) is normally read from a SCRN file and used to dimension the appropriate arrays. These variables are now set to the appropriate value in the form view so that the same INITIALIZE routine may be used throughout the application.

2. Variables are set to the number of each type of interactive control that is available. This procedure makes the code that processes these controls *generic*!

3. A GOSUB INITIALIZE is performed to dimension the arrays and define the special pattern masks that are used. This routine is stored in the **cpu\init.cpu** file and inserted with a #include statement in the Init subroutine. Turn to the end of this section if you want to look at the actual code that is used.

4. The variable DATA.DESC$ is set to the name of the data file to be accessed. A GOSUB OPEN.FILE calls the original code (in **cpu\openfile.cpu**) that opens the file.

```
$SUB frmclnt
REM@# dlgstart
keyfield% = 1 \ dtafield% = 15 \ totfield% = 16
buttons% = 2 \ checks% = 1 \ radios% = 0 \edits% = 13
lists% = 0 \ texts% = 12 \ scrolls% = 0 \ combos% = 0
data.desc$ = ".\cp\data\client"
gosub initialize
gosub open.file
REM@# dlgend
GOSUB frmclnt.CREATE
$EXIT
```

The Button Click Subroutine

This subroutine has been modified to test the value of CP.FLAG% and process a menu selection or toolbar button click that has been passed from the cetWINSUB subroutine in the **cpwin32.b** file. The code that it common to *all* programs is inserted with a #include “cpu\toolclck.cpu”. (Refer to the last section if you want to see the actual code that was used.)

Code has also been included to process the Description and Sales Memo button clicks. The routine called by the GOSUB ED.MEMO statement (and all the rest of the memo code) is inserted in the Init subroutine with a #include “cpu\cetmemo32.cpu” statement.

Note that a user-defined function FN.FINDIDX% has been created to find the index value (FIELD%) for a specific CmdID%. (In our programs, FIELD% is the same as the tab order.) In the case of these buttons, the promt message will be displayed when the control receives the focus.

Also note that a CASE 1 statement has been entered to detect when the operator has clicked on the System-Close button or entered Alt-F4.

```
$SUB frmclntBtnClick
```

```
REM@# btnstart
#include "cpu\toolclck.cpu"
SELECT CMDID%
    CASE 1
        msgbox “You may not quit the program at this time.”
        Resume 0
    CASE 200 \ rem Description memo
        field% = fn.FindIdx%(200)
        gosub ed.memo
    CASE 201 \ rem Sales memo
        field% = fn.FindIdx%(201)
        gosub ed.memo
    CEND
REM@# btnend
```

The Event Subroutine

Except for button clicks, all other types of events that may occur are processed in this subroutine. In CLIENT32, three event types are detected and processed.

1. When an edit control has changed (CASE 1), the program checks to determine if it is the key field (CMDID%=100). If so, a call is made to SET.FIELD which disables all menu items (and the corresponding toolbar buttons) that should not be available when entering a new record.

   Regardless of which edit control has changed, the EDIT.CHANGED% flag is set to trigger the special validation and formatting routines.

2. When a control receives the input focus (CASE 6), the appropriate prompt message (in the control properties) is displayed in the status bar. The FNPRMPT.ERS$ function is inserted with #include”cpu\fnprmpt.cpu” at the end of the program.

3. When the control has lost focus (CASE 7) and the EDIT.CHANGED% flag has been set, the user’s input will be validated and formatted according to the values specified in the user-defined properties for the control. (The code the performs the operations is inserted with the #include “cpu\fieldfmt.cpu” statement in the usrstart save area.)

```
$SUB frmclntEvent
SELECT CET_EVTYPE%
REM@# evtstart
CASE 1
    if CMDID%=100 then gosub set.field
    edit.changed% = -1
CASE 6
```

```
        rem Print prompt messages.
        field% = fn.FindIdx%(CMDID%)
        print fnprmpt.ers$(txt$(field%))
    CASE 7
        if edit.changed%<>0
            fld% = CMDID%
            field% = fn.FindIdx%(fld%)
            select
            case fmt$(field%)="D"
                gosub date.format
                call cSetEditText( HDLG%, fld%, date.fld$)
            case fmt$(field%)[1:1]="N"
                gosub number.format
                rem Not done yet.
            case fmt$(field%)[1:1]="$"
                gosub dollar.format
                rem Not done yet.
            case ucase$(msk$(field%))="PHONE"
                gosub phone.format
                call cSetEditText( HDLG%, fld%, phone.fld$)
            case ucase$(msk$(field%))="ZIP"
                gosub zip.format
                call cSetEditText( HDLG%, fld%, zip.fld$)
            case ucase$(msk$(field%))="SOCSEC"
                gosub ss.format
                call cSetEditText( HDLG%, fld%, ss.fld$)
            cend
            edit.changed% = 0 \ fld% = 0
            ifend
        if cmdid%=100 and cet_evparm1%>100 and cet_evparm1%<=800
        rem Only if we just left the key to move to one of the data fields:
            call cGetCtrlValue( HDLG%, CMDID%, ADDROF(key$))
            gosub get.rec
            if eof.flg%
                gosub set.key
                call cSetFocus( HDLG%, 100)
            else
                if oldrec%=-1 then gosub disp.rec
                ifend
            ifend
    REM@# evtend
    CEND
    $EXIT
```

Note that a test is made to determine if the control that just changed was the key field (CMDID%=100) and the ID of the next control is within the valid range

(CET_EVPARM1% > 100 and CET_EVPARM1 < 800). This range test was necessary since moving anywhere (even outside the form via the minimize button) will generate a lost focus event #7.

If the operator has gone from the key to another data field in the form, then a call is made to cGetCtrlValue to get the data that was input and use it to search for an existing record (GOSUB GET.REC).

If a record is not found, the GOSUB SET.KEY (inserted with #include "cpu\set.cpu") will enable the rest of the interactive controls (and turn them white) and disable/enable the appropriate menu items and toolbar buttons. Afterwards the focus is set back to the key field (call cSetFocus(hdlg%,100) so that the "New Record" prompt message could be displayed.

If a record is found, a GOSUB DISP.REC is performed to display the record. This routine has been hard-coded into the usrstart save area.

The Init Subroutine

This subroutine contains the code to load the new toolbar and access the user-defined properties.

Note that we set variables such as CTRLS.CNT% to the ones generated by the dialog editor (and stored in the **.idh** file) so we could refer to the generic names in the rest of our code. These variables must be here. Their values are only known after the form is created in the Main subroutine.

The code in the **cpu\parseprp.cpu** file parses out the control arrays so that the user-defined properties may be accessed in tab order. This code is included in the last section for your convenience.

The last step is to perform a GOSUB SET.KEY which sets up the conditions for entering the key field:

1. The appropriate menu items and toolbar buttons are disabled or enabled.
2. The focus is set to the key field. (It is active and appears in white.)
3. All other interactive controls are disabled and grayed-out.

```
$SUB frmclntInit
REM@# inistart
rem Load toolbar.
#include "cpu\cpstdb32.cpu"

rem Set generic variables that are used in the #include files.
hdlg% = frmclntHDLG%
```

```
ctrls.cnt% = frmclntCTRLS.CNT%
dim ctrls%(ctrls.cnt%), props$(ctrls.cnt%,10)
mat ctrls% = frmclntCTRLS%
mat props$ = frmclntPROPS$

rem Code to pull out user-defined properties.
#include "cpu\parseprp.cpu"
gosub set.key
REM@# iniend
$EXIT
```

All of our routines and user-defined functions are stored in the usrstart save area. The *generic* routines that may be used in other programs are inserted with #include statements. The code in these files is included at the end of this section for your convenience.

The routines that are unique to CLIENT32 are hard-coded into the program:

1. When an existing record is found, DISP.REC will display the values for the edit controls and set the check box (the action required field) if the field contains a Y.
2. The CLEAN.SCRN routine is used to reset the controls to their initial state.
3. PRINT.SCRN will print the data on the currently displayed form. (We just bought a $29 utility from JE Software that enables the print screen key. If you are interested, fax 914-699-6969.)
4. The RETRIEVE.CTRLS routine is called from WRITE.REC to retrieve the control values before writing the record.

Note that these routines have been written so that very few changes need to be made when they are pasted into another form program.

```
REM@# usrstart

rem Misc Subroutines
#include "cpu\init.cpu"
#include "cpu\openfile.cpu"
#include "cpu\browseky.cpu"
#include "cpu\cetmem32.cpu"
#include "cpu\fileio.cpu"
#include "cpu\set.cpu"
#include "cpu\nexttab.cpu"
#include "cpu\fieldfmt.cpu"
#include "cpu\erroutin.cpu"

disp.rec:
```

```
        rem Display the edit controls.
        call cSetCtrlValue( frmclntHDLG%, 100, key$)
        i% = 3 \ rem Skip memo field - start with ndata$(3)
        for ctrl% = 101 to 112
            call cSetCtrlValue( frmclntHDLG%, ctrl%, ndata$(i%))
            i% = i%+1
            next ctrl%
        rem Set the check box if action req field contains "Y".
        if ndata$(16)="Y" then call cSetCheck( frmclntHDLG%, 800, 1)
        return

    clean.scrn:
        rem Clear the edit controls.
        for ctrl% = 100 to 112
            call cSetCtrlValue( frmclntHDLG%, ctrl%, "")
            next ctrl%
        rem Restore check box to unchecked.
        call cSetCheck( frmclntHDLG%, 800, 0)
        return

    print.scrn:
        open #2: "PRINTER1", output sequential
        rem Retrieve the control values.
        call cGetCtrlValue( frmclntHDLG%, 100, addrof(prdata$(1)))
        prdata$(2) = ndata$(2) \ rem Description memo
        i% = 3
        for ctrl% = 101 to 112
            call cGetCtrlValue( frmclntHDLG%, ctrl%, addrof(prdata$(i%)))
            i% = i%+1
            next ctrl%
        prdata$(15) = ndata$(15) \ rem Sales memo
        rem Set the action req field to "Y" if control is checked.
        call cGetCheck( frmclntHDLG%, 800, addrof(state%))
        if state%=1 then prdata$(16)="Y" else prdata$(16) = "N"
        for field% = 1 to totfield%
            print #2: tag.name$(field%)&":  "&prdata$(field%)
            next field%
        close #2
        mat prdata$=("")
        return

    retrieve.ctrls:
        rem Retrieve the control values.
        call cGetCtrlValue( frmclntHDLG%, 100, addrof(key$))
        i% = 3 \ rem Skip memo field - start with ndata$(3)
        for ctrl% = 101 to 112
            call cGetCtrlValue( frmclntHDLG%, ctrl%, addrof(ndata$(i%)))
            i% = i%+1
```

```
        next ctrl%
    rem Set the action req field to "Y" if control is checked.
    call cGetCheck( frmclntHDLG%, 800, addrof(state%))
    if state%=1 then ndata$(16)="Y" else ndata$(16) = "N"
    return

rem Misc User-defined functions
#include "cpu\fnprmpt.cpu"
#include "cpu\disperr.cpu"
#include "cpu\findidx.cpu"
REM@# usrend
```

The #Include Files

CLIENT32 uses a number of #include files. They have been documented here in alphabetical order for your convenience.

cpu\browseky.cpu

This file is the same one that was used in CLIENT. It contains a routine which displays a Windows choice list box with the keys to the existing records. The operator's selection is passed back to the program where it is used to read the desired record and display it.

The routine is called when the File-Open command is selected. The name of the data file to be opened is assigned to DATA.DESC$ in the Main subroutine.

```
BROWSE:
    OPEN #19: DATA.DESC$, INPUT INDEXED
    rem Determine record count to set size of choice list.
    call Bindinfo(19,addrof(rec.count),addrof(a%),addrof(b%))
    DIM in.keys$(rec.count)
    choice.count% = 0
    WHILE NOT EOF(19)
        READNEXT #19, temp$: dum$
        IF NOT EOF(19)
            choice.count% = choice.count% + 1
            in.key$(choice.count%) = temp$
            IFEND
        WEND
rem Array for choice list cannot be smaller than the number of items
rem to be displayed.
    DIM choice.list$(choice.count%)
    MAT choice.list$ = in.key$
    CLEAR in.key$
    selected$ = ""
    call cChoiceListNS("Select Record",mataddrof(choice.list$),
```

```
addrof(selected$))
    IF selected$ <> ""
        rem A selection has been made…
        rem Perform the operations required before a new record can
        rem be read and displayed. Since this code is unique to the
        rem CLIENT program, it has been omitted from this document.
.
.
.
    IFEND
CLEAR choice.list$,selected$
CLOSE #19
RETURN
```

cpu\cpstdb32.cpu

The cetPreInit subroutine in **cpwin32.b** has been modified to display the toolbar ("call cShowToolBar(1)") when the program is initialized. The **cpuser32.rc** file has also been modified to set IDR_TOOLBAR2 to the name of the custom toolbar which should be used (**res\postool2.bmp**).

The #include "cpstdb32.cpu" statement inserts the code to dimension and load the BTN% array. After the CALL to cToolbar is performed, the new toolbar will be used instead of the default provided with the W/32 App Builder.

Note that the toolbar created for CLIENT has been modified to display two new buttons in the CLIENT32 .version. The ↓ button may be used to display the next record in the data file. The ↑ button will display the previous record. In the text-based version of CLIENT, these operations were performed by entering the arrow keys in the key field or at the menu line. (Note that the Image Editor is provided so that you can create and modify toolbars.)

```
DIM BTN%(12)
BTN%(1)=7       \ rem Program exit
BTN%(2)=0
BTN%(3)=2       \ rem New (cancel) record.
BTN%(4)=3       \ rem Open record. A list of existing keys will appear.
BTN%(5)=4       \ rem Save Record
BTN%(6)=5       \ rem Delete Record
BTN%(7)=8       \ rem Next Record
BTN%(8)=9       \ rem Previous Record
BTN%(9)=0
BTN%(10)=6      \ rem Print Record
BTN%(11)=0
BTN%(12)=1      \ rem Help
CALL cToolBar(2,12,MATADDROF(BTN%))
```

cpu\cetmem32.cpu

The THEOS version of CLIENT uses a special feature called Memo Writer to allow users to take up to 100 lines of notes without leaving the data entry program. Under THEOS, a full-screen editor is provided along with special code used to file the memo with a unique name. This name is stored in the data record so that the memo could be called up later and modified.

CLIENT32 has been modified to use the Windows Notepad program. All of the special code to process the memo fields is stored in an external file which is included in the usrstart area with the statement:

```
#include cpu\cetmem32.cpu
```

The CLIENT demo in the last section used a modified version of this code that displayed an "X" in the field when a memo had been written.

Contact Phase One Systems if you have any questions about implementing this feature in your application. Phase One may be reached by calling (510) 895-0832 or by fax at (510) 895-0828.

cpu\disperr.cpu

This file contains the user-defined function which is used to display miscellaneous messages such as "New Record" in a Windows message box. In the original program, this text was displayed on row 24. (The same #include file was used in CLIENT.)

```
DEF FNERR.ERS$(DISPERR$)
    PRINT CRT$("BELL");
    MSGBOX DISPERR$
    FNEND
```

cpu\errroutin.cpu

This is a general purpose error handling routine that displays error text in message boxes.

```
erroutine:
    select err
    case 30
        msgbox "File on channel "&str$(erf)&" not found."
    case 31
        msgbox "Disk is full."
    case 48
        msgbox "Record is locked on channel "&str$(erf)&"."
    case 49
```

```
        msgbox "File is locked on channel "&str$(erf)&"."
    otherwise
        msgbox "Error "&str$(err)&" at line "&str$(erl)
    cend
    resume 0
```

An ON LOCK statement may also be used to call specific file and record lock conditions. In any case, note that if B_THLOCK and/or B_EMULATE are not set and if:

1. ON ERROR or ON LOCK are not set, the program will abort with a lock error (#48/49).
2. ON ERROR or ON LOCK are set, the program traps the lock error and sets ERR.
3. ON LOCK and ON ERROR are set, a lock error causes the ON LOCK trap to occur.

If B_THLOCK and/or B_EMULATE are set and if:

1. ON ERROR or ON LOCK are not set, a lock error causes the program to suspend until the lock condition disappears.
2. ON ERROR is set, a lock error suspends the program while the lock condition is in effect.
3. ON LOCK and ON ERROR are set, a lock error causes the ON LOCK trap to occur.

cpu\fieldfmt.cpu

This is a modified version of the original FIELDFMT routine that formats the various types of data entries. In all cases, a call is made to cGetEditText to get the data to be formatted. The code to do special formatting such as phone numbers and zip codes is taken directly from the original **client.b** file.

```
date.format:
    date.fld$ = ""
    call cGetEditText( HDLG%, fld%, addrof(date.fld$))
    date.fld$ = dte$(date.fld$)
    return

number.format:
    rem Format numeric fields.
    return

dollar.format:
    rem Format dollar fields.
    return
```

```
phone.format:
    phone.fld$ = ""
    call cGetEditText( HDLG%, fld%, addrof(phone.fld$))
    patt$ = "" \ pmap$ = ""
    for i% = 1 to 4
        for j% = 1 to 6
            if match(phone.fld$,phone.msk$(j%))
                patt$ = phone.msk$(j%) \ pmap$ = phone.map$(j%)
                j% = 6 \ i% = 4
                ifend
            next j%
        next i%
    if patt$<>""
        temp$ = ofmt$(field%)
        if patt$="#####" and pmap$[5:5]="%" then temp$=temp$[1:5]
        for i% = 1 to len(patt$)
            chr.pos% = asc(pmap$[i%:i%])-32
            temp$[chr.pos%:chr.pos%] = phone.fld$[i%:i%]
            next i%
        phone.fld$ = temp$
        ifend
    return

zip.format:
    REM Format zip code fields.
    zip.fld$ = ""
    call cGetEditText( HDLG%, fld%, addrof(zip.fld$))
    patt$ = "" \ pmap$ = ""
    for i% = 1 to 4
        for j% = 1 to 3
            if match(zip.fld$,zip.msk$(j%))
                patt$ = zip.msk$(j%) \ pmap$ = zip.map$(j%)
                j% = 3 \ i% = 4
                ifend
            next j%
        next i%
    if patt$<>""
        temp$ = ofmt$(field%)
        if patt$="#####" and pmap$[5:5]="%" then temp$=temp$[1:5]
        for i% = 1 to len(patt$)
            chr.pos% = asc(pmap$[i%:i%])-32
            temp$[chr.pos%:chr.pos%] = zip.fld$[i%:i%]
            next i%
        zip.fld$ = temp$
        ifend
    return
```

```
ss.format:
    REM Format social security# fields.
    ss.fld$ = ""
    call cGetEditText( HDLG%, fld%, addrof(ss.fld$))
    patt$ = "" \ pmap$ = ""
    for i% = 1 to 4
        for j% = 1 to 2
            if match(ss.fld$,ss.msk$(j%))
                patt$ = ss.msk$(j%) \ pmap$ = ss.map$(j%)
                j% = 2 \ i% = 4
                ifend
            next j%
        next i%
    if patt$<>""
        temp$ = ofmt$(field%)
        if patt$="#####" and pmap$[5:5]="%" then temp$=temp$[1:5]
        for i% = 1 to len(patt$)
            chr.pos% = asc(pmap$[i%:i%])-32
            temp$[chr.pos%:chr.pos%] = ss.fld$[i%:i%]
            next i%
        ss.fld$ = temp$
        ifend
    return
```

cpu\fileio.cpu

This routine reads, writes and deletes records. It has been extracted from the original **client.b** and modified where necessary (in WRITE.REC and READ.RAND). All modifications have been entered in lower case letters. See the rem** lines for information on the few changes that were made.

```
CHECK.KEY:
    TEMP$=""
    FOR FIELD% = 1 TO KEYFIELD%
        TEMP$ = TEMP$&"*"&NDATA$(FIELD%)
    NEXT FIELD%
    TEMP$ = RIGHT$(TEMP$,2)
    IF ERR.FLG%
        PRINT FNERR.ERS$("Invalid record number.");
    ELSE KEY.CHG% = 0
        IF TEMP$<>KEY$ THEN KEY.CHG%= -1
    IFEND
    RETURN

WRITE.REC:
    NEW.STK%=0\OLD.STK%=0
```

```
      REM if key has been modified on old record, delete old one before
writing new one
      IF OLDREC% AND KEY.CHG% THEN GOSUB DELETE.REC
      IF KEY.CHG%
      KEY$ = TEMP$
      XFIELD% = FIELD%
      FOR FIELD% = 2 TO TOTFIELD%
            IF FMT$(FIELD%)="M" AND NDATA$(FIELD%)<>""
                  GOSUB GET.PARAMS
rem** Don't need now...
rem               MEMO.FILE$ = MEMO.LIB$&"."&MEMB$
rem               GET.F% = 0
rem               GOSUB READ.MEMO
rem               PARAM%(6) = I%-1
rem               GOSUB WRITE.MEMO
                  IFEND
            NEXT FIELD% \ FIELD% = XFIELD%
            ELSE KEY$ = TEMP$
      IFEND
      rem** Retrieve the values of the controls.
      gosub retrieve.ctrls
      FOR R% = 1 TO DTAFIELD%
            WDATA$(R%)= NDATA$(R%+KEYFIELD%)
            NEXT R%
      MAT WRITE #1,KEY$: WDATA$
      RETURN

DELETE.REC:
      NEW.STK%=0\OLD.STK%=0
      DELETE #1,KEY$
      IF KEY.CHG%=0
      XFIELD% = FIELD%
      FOR FIELD% = 2 TO TOTFIELD%
            IF  FMT$(FIELD%)="M"  AND  NDATA$(FIELD%)<>""  THEN
      GOSUB GET.PARAMS \ GOSUB DEL.MEMO
            NEXT FIELD%
            FIELD% = XFIELD%
            IFEND
      RETURN

GET.REC:
      IF RDPRV%
            IF LKEY>0 THEN KEY$ = LKEY$ \ KEY = LKEY ELSE KEY =
1 \ KEY$ = "1"
            IFEND
      EOF.FLG% = 0
```

```
        GOSUB READ.RAND
        RETURN

    READ.RAND:
        IF NOT RDNXT% AND NOT RDPRV%
            IF KEY$<>"" THEN MAT READ #1,KEY$: INREC$
            IF EOF(1)
                PRINT FNERR.ERS$("New Record.");
                rem Set defaults.
                for ctrl% = 100 to 99+edits%
                    field% = fn.FindIdx%(ctrl%)
                    if dft$(field%)<>""
                        select
                        case ucase$(dft$(field%))="@DATE"
                            call cSetEditText( HDLG%, ctrl%, date$(0))
                        otherwise
                            call cSetEditText(hdlg%, ctrl%, dft$
(field%))
                        cend
                    ifend
                next ctrl%
            ELSE OLDREC% = -1
                GOSUB SHUFFLE
            IFEND
        ELSE
            IF RDNXT%
                MAT READNEXT #1,KEY$: INREC$
            ELSE
                MAT READPREV #1,KEY$: INREC$
                IFEND
            IF EOF(1) OR D.FLG% \ REM 'top of file' flag for direct file
                IF RDNXT%
                    PRINT FNERR.ERS$("End of file. File pointer is reset
to tof..."); \ rem **
                    CLOSE #1 \ GOSUB OPEN.FILE \ EOF.FLG% = -1
                ELSE
                    PRINT FNERR.ERS$("Top of file. File pointer is reset
to bof..."); \ rem **
                    READ #1,CHR$(255): X$
                    EOF.FLG% = -1
                    IFEND
            ELSE OLDREC% = -1
                TEMP$ = KEY$ \ REM separate key into fields
                FOR FIELD% = 1 TO KEYFIELD%-1
                    NDATA$(field%)=LEFT$(temp$,SCH(1,temp$,"*")-1)
                    TEMP$ = RIGHT$(TEMP$,SCH(1,TEMP$,"*")+1)
```

```
            NEXT FIELD%
        NDATA$(KEYFIELD%) = TEMP$
        GOSUB SHUFFLE
        IFEND
    IFEND
RETURN

SHUFFLE:
    REM transfer input fields to DATA array
    FOR FIELD% = KEYFIELD%+1 TO TOTFIELD%
        NDATA$(FIELD%) = INREC$(FIELD%-KEYFIELD%)
    NEXT FIELD%
    MAT INREC$ = ("")
    RETURN
```

cpu\findid.cpu

This file contains a user-defined function that is called whenever we need to find FIELD% (e.g. the index into the array that holds the data record (NDATA$)).

```
def fn.FindIdx%( idx% )
    REM find the index of the control with ID=idx%
    for i% = 1 to CTRLS.CNT%
        if CTRLS%(i%) = idx%
            fn.FindIdx% = i%
            goto fn1end
            ifend
        next i%
    msgbox "Control "+STR$(idx%)+" was not found!"
    fn.FindIdx% = 0
fn1end:
    fnend
```

cpu\fnprmpt.cpu

Defines the FNPRMPT.ERS$ function that is used to print the prompting messages for the individual data items on the status line. Note that these messages are read from an external file in the original version of CLIENT(under DOS). Now the text is stored in one of the user-defined control properties and gets displayed when the control receives the focus.

```
DEF FNPRMPT.ERS$(MSG$)
    CALL cStatusText(MSG$)
    FNEND
```

cpu\init.cpu

This routine is inserted in the usrstart save area. The code was extracted from the original **client.b** file and modified (in lower case letters) to include the arrays for the new properties:

DFS$	Data File Source
TAG.NAME$	Tag Name

The PRDATA$ array is used to hold the values of the interactive controls for printing. (See the PRINT.SCRN subroutine.)

Instead of hard-coding the number of elements in the arrays, they are now dimensioned to the value of TOTFIELD% which is set to the number of interactive controls in the Main subroutine.

```
rem Initializes variables, dimensions arrays, loads in special field format
rem masks.
INITIALIZE:
     ON ERROR GOTO ERROUTINE
REM data field specs
     DIM FMT$(totfield%),TXT$(totfield%)
     DIM CS$(totfield%),REQ$(totfield%),DFT$(totfield%),OFMT$
(totfield%),MSK$(totfield%)
     dim dfs$(totfield%),tag.name$(totfield%)
REM data storage
     DIM NDATA$(totfield%),OLDATA$(totfield%),INREC$(totfield%),
WDATA$(dtafield%)
     dim prdata$(totfield%)
REM Control variables
     OPTION PROMPT "", CASE "M"
REM Special pattern masks
     DIM PHONE.MSK$(6),PHONE.MAP$(6),SS.MSK$(2),SS.MAP$(2),
ZIP.MSK$(3),ZIP.MAP$(3)
     FOR I% = 1 TO 6
          READ PHONE.MSK$(I%),PHONE.MAP$(I%)
     NEXT I%
     DATA "(###) ###-####","!""#$%&'()*+,-."
     DATA "(   ) ###-####","!""#$%&'()*+,-."
     DATA "##########","""#$'()+,-."
     DATA "### ###-####","""#$&'()*+,-."
     DATA "#######","'()+,-."
     DATA "###-####","'()*+,-."
     FOR I% = 1 TO 2
          READ SS.MSK$(I%),SS.MAP$(I%)
     NEXT I%
     DATA "###-##-####","!""#$%&'()*+"
     DATA "#########","!""#%&()*+"
```

```
FOR I% = 1 TO 3
    READ ZIP.MSK$(I%),ZIP.MAP$(I%)
NEXT I%
DATA "#####-####","!""#$%&'()*"
DATA "#########","!""#$%'()*"
DATA "#####","!""#$%"
RETURN
```

cpu\nexttab.cpu

This routine is called after DISP.REC to set the focus to the control that comes next in the tab order after the key field(s). (The FN.FINDINDX% function actually determines which control is next.)

The code was made *generic* so that it could be used in all programs.

```
set.next.tab:
    if edits%<>0
        for ctrl% = 99+keyfield% to 99+edits%
            if fn.FindIdx%(ctrl%)=keyfield%+1 then goto next.tab
            next ctrl%
        ifend
    if buttons%<>0
        for ctrl% = 200 to 199+buttons%
            if fn.FindIdx%(ctrl%)=keyfield%+1 then goto next.tab
            next ctrl%
        ifend
    if lists%<>0
        for ctrl% = 300 to 299+lists%
            if fn.FindIdx%(ctrl%)=keyfield%+1 then goto next.tab
            next ctrl%
        ifend
    if scrolls%<>0
        for ctrl% = 400 to 399+scrolls%
            if fn.FindIdx%(ctrl%)=keyfield%+1 then goto next.tab
            next ctrl%
    ifend
    rem Skip text (400's) and group (500's) controls.
    if combos%<>0
        for ctrl% = 700 to 699+combos%
            if fn.FindIdx%(ctrl%)=keyfield%+1 then goto next.tab
            next ctrl%
        ifend
    if checks%<>0
        for ctrl% = 800 to 799+checks%
            if fn.FindIdx%(ctrl%)=keyfield%+1 then goto next.tab
            next ctrl%
```

```
            ifend
        if radios%<>0
            for ctrl% = 900 to 899+checks%
                if fn.FindIdx%(ctrl%)=keyfield%+1 then goto next.tab
                next ctrl%
            ifend
next.tab:
    call cSetFocus( HDLG%, ctrl%)
    rem Set focus to next field in tab order (Description button).
    return
```

cpu\openfile.cpu

This routine has been taken from the original **client.b** file.

```
OPEN.FILE:
    OPEN #1: DATA.DESC$, UPDATE INDEXED
    RETURN
```

cpu\parseprp.cpu

In the form view, the array that is used to format and validate the user's input is loaded from the user-defined properties. In the text-based application, these values are read from an external (SCRN) file.

This code parses out the PROPS$ array. Note that the interactive controls come first in the tab order.

```
for i% = 1 to totfield%
    dfs$(i%) = PROPS$(i%,1)
    rem Don't need field position property ( PROPS$(i%,2) ).
    tag.name$(i%) = PROPS$(i%,3)
    fmt$(i%) = PROPS$(i%,4)
    cs$(i%) = PROPS$(i%,5)
    req$(i%) = PROPS$(i%,6)
    dft$(i%) = PROPS$(i%,7)
    msk$(i%) = PROPS$(i%,8)
    txt$(i%) = PROPS$(i%,9)
    ofmt$(i%) = PROPS$(i%,10)
    next i%
```

cpu\set.cpu

This file includes two new routines SET.KEY and SET.FIELD. These routines were made *generic* by using the variables set in the Main subroutine (e.g. EDIT%, BUTTONS% etc.).

SET.KEY sets the form to a *ready* state waiting to do input on the key fields; other date fields (controls) are disabled as well as some menu items and toolbar buttons. This routine is called in the Init subroutine and after the file I/O operations to file, delete and cancel (the New command) records.

SET.FIELD is called after the key has been input or after a record has been opened so that other data or menu items may be entered or selected. This is done by disabling/enabling the appropriate items.

```
set.key:
    rem Disable controls other than key fields.
    if edits%>keyfield%
        for ctrl% = 100+keyfield% to 99+edits%
            call cEnableCtrl( HDLG%, ctrl%, 0)
            next ctrl%
        ifend
    if buttons%<>0
        for ctrl% = 200 to 199+buttons%
            call cEnableCtrl( HDLG%, ctrl%, 0)
            next ctrl%
        ifend
    if lists%<>0
        for ctrl% = 300 to 299+lists%
            call cEnableCtrl( HDLG%, ctrl%, 0)
            next ctrl%
        ifend
    if scrolls%<>0
        for ctrl% = 400 to 399+scrolls%
            call cEnableCtrl( HDLG%, ctrl%, 0)
            next ctrl%
        ifend
    rem Skip text (500's) and group (600's) controls.
    if combos%<>0
        for ctrl% = 700 to 699+combos%
            call cEnableCtrl( HDLG%, ctrl%, 0)
            next ctrl%
        ifend
    if checks%<>0
        for ctrl% = 800 to 799+checks%
            call cEnableCtrl( HDLG%, ctrl%, 0)
            next ctrl%
        ifend
    if radios%<>0
        for ctrl% = 900 to 899+radios%
            call cEnableCtrl( HDLG%, ctrl%, 0)
            next ctrl%
```

```
        ifend
    rem Disable some menu items.
    call cEnableItem( 2, 0) \ rem Disable New
    call cEnableItem( 4, 0) \ rem Disable Save
    call cEnableItem( 5, 0) \ rem Disable Delete
    call cEnableItem( 6, 0) \ rem Disable Print
    rem Print prompt message for 1st key field.
    print fnprmpt.ers$(txt$(1))
    rem Original GET.KEY code.
    LKEY$ = KEY$ \ LKEY = KEY
    KEY$ = "" \ KEY = 0
    MAT NDATA$=("") \ RDNXT% = 0 \ RDPRV% = 0 \ ERR.FLG% = 0
    return

set.field:
    rem Enable other controls that were disabled during key field entry
    if edits%>keyfield%
        for ctrl% = 100+keyfield% to 99+edits%
            call cEnableCtrl( HDLG%, ctrl%, 1)
            next ctrl%
        ifend
    if buttons%<>0
        for ctrl% = 200 to 199+buttons%
            call cEnableCtrl( HDLG%, ctrl%, 1)
            next ctrl%
        ifend
    if lists%<>0
        for ctrl% = 300 to 299+lists%
            call cEnableCtrl( HDLG%, ctrl%, 1)
            next ctrl%
        ifend
    if scrolls%<>0
        for ctrl% = 400 to 399+scrolls%
            call cEnableCtrl( HDLG%, ctrl%, 1)
            next ctrl%
        ifend
    rem Skip text (500's) and group (600's) controls.
    if combos%<>0
        for ctrl% = 700 to 699+combos%
            call cEnableCtrl( HDLG%, ctrl%, 1)
            next ctrl%
        ifend
    if checks%<>0
        for ctrl% = 800 to 799+checks%
            call cEnableCtrl( HDLG%, ctrl%, 1)
            next ctrl%
        ifend
```

```
if radios%<>0
    for ctrl% = 900 to 899+radios%
        call cEnableCtrl( HDLG%, ctrl%, 1)
        next ctrl%
    ifend
rem Enable menu items
call cEnableItem( 2, 1) \ rem Enable New
call cEnableItem( 4, 1) \ rem Enable Save
call cEnableItem( 5, 1) \ rem Enable Delete
call cEnableItem( 6, 1) \ rem Enable Print
return
```

cpu\toolclck.cpu

This code is called from the BtnClick subroutine. The SELECT-CASE statements detect when a menu item or a toolbar button click has been entered and execute the appropriate operations.

```
if cp.flag%<>0 \ rem cp.flag% is passed from cpwin32 file.
    select cp.flag%
    case 2 \ rem New (cancel)
        gosub clean.scrn
        gosub set.key
        call cSetFocus( HDLG%, 100) \ rem Set focus to 1st key field.
    case 3 \ rem Open
        gosub browse
        if found%<>0
            found% = 0
            gosub set.field
            gosub disp.rec
            gosub set.next.tab
            ifend
    case 4 \ rem Save
        gosub check.key
        gosub write.rec
        gosub clean.scrn
        gosub set.key
        call cSetFocus( HDLG%, 100) \ rem Set focus to 1st key field.
    case 5 \ rem Delete
        gosub delete.rec
        gosub clean.scrn
        gosub set.key
        call cSetFocus( HDLG%, 100) \ rem Set focus to 1st key field.
    case 6 \ rem Print
        gosub print.scrn
    case 8 \ rem Next
        rdnxt% = -1
```

```
        gosub get.rec
        if not eof.flg%
            gosub set.field
            gosub disp.rec
            gosub set.next.tab
        else
            gosub clean.scrn
            gosub set.key
            call cSetFocus(hdlg%,100)\ rem Set focus to 1st key field.
            ifend
        rdnxt% = 0
    case 9 \ rem Prior
        rdprv% = -1
        gosub get.rec
        if not eof.flg%
            gosub set.field
            gosub disp.rec
            gosub set.next.tab
        else
            gosub clean.scrn
            gosub set.key
            call cSetFocus(hdlg%,100)\ rem Set focus to 1st key field.
            ifend
        rdprv% = 0
    cend
    cp.flag% = 0
    ifend
```

Converting the Rest of Your Application

Once you are satisfied with the operation of your model program, create (or convert) your next input form. From this point on, it should be mainly a matter of cutting and pasting code (or inserting #include statements) from your model form to your new one. The more generic you have made the model program, the fewer changes will have to be made. This process has been simplified since multiple files may be open at the same time.

The advantages of using this procedure are two-fold. Afterwards, you will have

- ✓ A GUI interface that looks and operates the way users expect.
- ✓ A highly structured, event-driven program that is easy to maintain.