WINDOWS FOR C

by

Vince Taylor
Robert Otterberg

Reference Manual

Version 4.0

April 1986

Vermont
Creative
Software

DISCLAIMER OF WARRANTY

This manual and associated software are sold "as is" and without warranties as to performance or merchantability. The seller's salespersons may have made statements about this software. Any such statements do not constitute warranties.

This program is sold without any express or implied warranties whatsoever. No warranty of fitness for a particular purpose is offered. The user is advised to test the program thoroughly before relying on it. The user assumes the entire risk of using the program. Any liability of seller or manufacturer will be limited exclusively to replacement of diskettes defective in materials or workmanship.

Vermont Creative Software
21 Elm Ave
Richford, VT 05476
(802)848-7738

Version 4.0 Reference Manual

First Printing	October, 1985
Second Printing	November, 1985
Third Printing (revised)	February, 1986
Fourth Printing (revised)	April, 1986

PREFACE

This reference manual provides information necessary to use **Windows for C.**

Chapter 2 provides a guide to the organization of the manual and diskettes. For information on the contents of this reference manual, refer to Chapter 2.

AUTHORS' NOTE

The first version of **Windows for C** was conceived and written by one of us (V.T). The revisions that produced the present version were a joint effort.

Revising **Windows for C** while producing **Windows for Data** has not been easy. We would like to thank our wives for their understanding and forbearance at the long hours we've spent away from home. We also want to express our thanks for the support and assistance of Evelyn Guilmette, who has kept the business running smoothly while we were programming and writing.

Finally, thanks to the many customers who have expressed their appreciation of **Windows for C.** Without your support, this new version would not have happened.

Vince Taylor
Robert Otterberg

This page intentionally left blank.

WARNING TO ALL USERS

We are considering revising the order of elements in the **WINDOW** and **FREC** structures that carry the basic information about windows and memory files. No final decision on this revision has been made. We are providing this forewarning so that you can take steps to minimize the difficulties that such a revision would cause you.

To keep future problems to a minimum:

1) Use functions rather than external or static initializations to assign initial values to WINDOW and FREC structures. The functions **def_wn()**, **defs_wn()**, and **def_fr()** are provided for this purpose. You may wish to construct additional ones.

2) Place external initializations of structures in a single include file and #include it prior to main().

3) Avoid static initializations that are buried in subroutines.

We realize that any revision in the WINDOW and FREC structures will create problems for users, and we will not undertake it lightly. However, the computer world continues to change, and **Windows for C** must also change to maintain viability. As we continue to add to the capabilities of **Windows for C**, the original ordering becomes less and less logical. In new versions, some of the original elements may become archaic, and we would like to be able to place them at the bottom, where they would be available for those who continue to need them, but where users who did not need them would not be required to allocate memory for them.

Internal Subroutines

Windows for C contains a number of subroutines that are internal and not intended for user access. The names of internal **Window** subroutines begin with a prefixed underscore. Avoid names with this prefix to avoid possible conflict in names.

This page intentionally left blank.

TABLE OF CONTENTS

Preface and Authors' Note iii

Warning v

Changes from Prior Versions C-1

Chapter 1: Overview of Windows for C 1-1

Chapter 2: Getting Started 2-1

- Organization of Material 2-3
  - Organization and Contents of the Diskettes 2-3
  - Where to Find Information in this Manual. 2-4
- Compiling and Linking Programs 2-5
  - Include Files Required by Windows for C 2-5
  - Referencing the Include Files in Your Programs 2-5
  - Linking Window Libraries with Your Programs 2-5
  - A Batch File for Compiling and Linking 2-5
- Using Physical Attributes 2-6
- Some IBM Compatibles Are Incompatible with TopView and MS Windows . . . 2-6

Chapter 3: Window Basics 3-1

- Direct Display and Memory File Display 3-3
- The First Window Program: "Hello, World" 3-3
- Six Steps To Windows 3-4
  - Step 1: Include System Files 3-4
  - Step 2: Declare a Window 3-5
  - Step 3: Initialize Windows for C 3-5
  - Step 4: Define Initial Values of the Window 3-5
  - Step 5: Set the Window On The Screen 3-6
  - Step 6: Write To the Window 3-6
  - Recap 3-6
- Changing Defaults and Using Options 3-7
  - Definitions, Usage, and Abbreviations 3-7
  - Set-Window-Member Functions 3-8
  - Making Pop-up Windows 3-8
  - Naming a Window 3-9
  - Changing Window Margins 3-9
  - Changing Word-wrap, Auto-scroll and Cursor Placement Options . . . 3-9
  - Changing Special Options 3-10
  - Modifying Window Size and Location 3-10
- Controlling the Appearance of Output: Attributes 3-11
  - Physical Attributes 3-11
  - Logical Attributes 3-11
  - Using Physical Attributes 3-12
  - Physical and Logical Attributes Don't Mix 3-12
  - Changing the Attribute of Output 3-13
- Displaying and Removing Windows 3-14
  - Displaying a Window On the Screen 3-14
  - Removing Windows from the Screen 3-14
- Controlling the Location of Output 3-15
  - The Virtual Cursor 3-15
  - Controlling the Screen Cursor 3-16

TABLE OF CONTENTS

Writing To Windows 3-18
Basic String Output: v_st() 3-18
Full-string Output Function: v_fst() 3-18
Writing a String at a Specified Location: v_plst() 3-19
Centering, Left-Justifying and Right-Justifying Text 3-19
Formatted String Output: v_printf() 3-19
Writing Characters 3-19
Sounding the Bell (Beep) 3-20
Changing the Attribute of a Character 3-20
Scrolling Direct Display Windows 3-20
Reading the Contents of Windows 3-21
Reading a Character 3-21
Reading an Attribtute 3-21
Reading the Character Contents of Parts of the Window 3-21
Reading the Character-attribute Contents of Parts of the Window . 3-21
Reading the Keyboard 3-22
Keycode Conventions 3-22
The Read-Keystroke Function: ki() 3-22
Executing Subroutines While Waiting for Keystrokes 3-23
Creating a Pause in a Program 3-23
Checking the Keyboard Buffer: ki_chk() 3-23
Clearing the Screen and Windows 3-24
Clearing the Screen 3-24
Clearing Windows 3-24
Removing a Window from the Screen 3-24
Controlling The Color of the Screen Background 3-24
Using Window Functions on the Full Screen 3-25
Saving, Clearing, and Restoring the Original DOS Screen 3-25
Moving, Saving, and Restoring the Screen Cursor 3-25
Writing To the Full Screen 3-26
Another Version of "Hello, World" 3-26
Practical Examples of Windows 3-26
An Error Message Window 3-26
Establishing a Status Line 3-27

Chapter 4: Tutorial on Windows for C 4-1

Chapter 5: Controlling Color: Logical and Physcal Attributes 5-1

The Logical Attribute System 5-3
Why Logical Attributes? 5-3
Logical Attributes 5-3
Changing the Physical Attributes of Logical Attributes 5-4
Adding New Logical Attributes 5-5
Adding New Columns of Logical Attributes 5-6
Initializing the Logical Attribute Array 5-6
Constructing and Using Window-Specific Logical Attribute Tables . 5-7
Using Physical Attributes 5-8
Setting the System To Use Physical Attributes 5-8
Monochrome Attributes 5-9
Avoid Underline in System Logical Attributes 5-9
Managing Window Colors 5-9
Selecting the Screen Border Color 5-11
"Hello, World" with Physical Color Attributes 5-12
Determining and Changing Video Modes 5-12

TABLE OF CONTENTS

Using Graphics in Programs that Use Windows for C 5-12
Coding Example 5-13

Chapter 6: Creating and Viewing Memory Files 6-1

Creating Memory Files 6-3
Coding Example 6-3
Three Steps to Creating a Memory File 6-4
Reading a Memory File from Disk 6-5
Writing Lines Directly to a Memory File 6-5
Viewing Memory Files through Windows 6-7
Setting a Window to View a File 6-7
Viewing a File 6-7
Scrolling a File in a Window 6-8
Performing Operations on a File Displayed in a Window 6-9
Modifying Memory Files 6-9
Accessing Memory-File Lines 6-9
Modifying and Replacing Memory-File Lines 6-9
Scrolling The Contents of Memory Files 6-10
How Windows for C Functions Manage Memory Files 6-10
Clearing and Freeing a Memory File 6-11

Chapter 7: Help Files, Menus, and Off-Screen Buffers 7-1

Using a Pop-Up Help File 7-3
Preparing the Help File 7-3
Reading Help Files Into Memory 7-3
Displaying Files 7-3
Coding Example 7-3
Multiple Help Files 7-4
Creating and Displaying Pop-Up Menus 7-4
Preparing to Call a Menu 7-5
Calling the Menu 7-6
Coding Example of a Menu 7-7
Menu Demonstration Program 7-7
Managing an Off-Screen Display Buffer 7-7
Memory Files as an Alternative to Virtual Screens 7-7
Using a Memory File as an Off-Screen Buffer 7-8

Chapter 8: Advanced Topics, Utilities, and Demonstrations 8-1

Window Viewing of Multiple Files 8-3
Memory Requirements 8-3
Handling Multiple Files 8-3
Demonstration of Viewing Multiple Files 8-5
Moving Information from and to Windows 8-5
Moving the Character Contents of Windows 8-5
Using a Window as an Edit Buffer 8-7
Copying the Contents of a Window to a File 8-8
Moving Character and Attribute Contents of Windows 8-8
Moving, Saving, and Restoring Window Images 8-9
Moving Windows 8-9
Saving and Replacing Window Images 8-10
Character-Graphics Animation 8-10
Storing Window Images on Disk 8-11

TABLE OF CONTENTS

Highlighting and Changing Attributes 8-11
Highlighting a Specified Number of Characters 8-11
Formatting Text for Printing with Windows 8-12
A Demonstration of Printing Side-by-Side Labels 8-12
Graphing Functions 8-12
Using and Modifying System Globals 8-13
Using Alternative Display Adapters 8-13
String Utilities 8-13
Miscellaneous Utilities 8-14
Low-Level Character and String Functions: v_qch() and v_st_rw() . 8-14
Macros for Window Row and Column Quantities 8-15
Error Exit Function 8-15
Duplicating Window Structures 8-15
Developing Your Own Applications 8-15
User-Reserved Pointers 8-15
Building New Functions 8-16

Chapter 9: Microsoft Windows and TopView Compatibility 9-1

Video Mangement Under TopView and Microsoft Windows 9-3
How Windows for C Operates Under MSW/TV 9-3
Program Control of Screen Updates Under MSW/TV 9-4
Direct Control of Screen Updates Under MSW/TV 9-4
MSW/TV Program Information Files 9-5
Running Window Programs Under MSW/TV 9-5

Tables and Listings T-1

Table 3.1: Window Members, Default Settings, and Change Functions . T-2
Table 3.2: Logical Attribute Definitions T-4
Table 5.1: Physical Attribute Definitions T-5
Listing 5.1: Initializing the Logical Attribute Array T-6
Listing 5.2: Window-Specific Logical Attributes T-7
Listing 5.3: Hello World in Color Using Physical Attributes T-8
Table 6.1: File Viewing Key Assignments T-9
Listing 7.1: Displaying and Scrolling a Help File T-10
Listing 7.2: Demonstration of a Vertical-Format Pop-Up Menu T-11
Listing 7.3: Function for Writing a String to an Off-Screen Buffer . T-14

Appendix 1: #Include Files A1-1

Appendix 2: Windows for Data -- Library Functions A2-1

Appendix 3: Source Code Files A3-1

Appendix 4: Definitions and Abbreviations A4-1

Appendix 5: Window and Memory File Structures A5-1

Appendix 6: Error Handling and Error Codes A6-1

Appendix 7: System Diskette Files A7-1

Index I-1

CHANGES FROM PRIOR VERSIONS

CHANGES FROM VERSION 3.1 to 4.0

The major changes are:

* A logical attribute system allows you to write one program for both color and monochrome systems.

* Pop-up windows are now implemented by setting a switch in the window structure. Functions **set_wn()** and **unset_wn()** do the rest.

* The functions **di_f()** and **v_f()** have been replaced by an extensive set of new functions for creating, modifying, and displaying files in memory. Among the benefits:

  ** You can easily create memory files from in-line code.

  ** Information for display can be stored and updated off-screen.

* The menu function is much easier and less demanding.

* Formatted output, equivalent to **printf()**, is available.

* Windows can have names automatically displayed on the border.

* When the IBM Enhanced Graphics Adapter is active, the system takes advantage of its faster output.

* The manual has been extensively revised.

Many changes have been made to **Windows for C** to increase its power and improve ease of use. Even if you are very familiar with **Windows for C**, you should read the remainder of this section and review Chapters 3 through 7. There is much new material.

All of the changes made are summarized below, and you are referred to appropriate sections of the manual for further information.

CHANGES IN INCLUDE FILES

Files **bios.h** and **window.h** remain the two top-level include files and retain the same functions, but the internal organization has undergone major revision. We have separated the top-level files into more subsidiary files to try to bring together similar information. This should make it easier for you to find specific #defines, declarations, and constants once you are familiar with the organization. See Appendix 1 for more information.

LOGICAL ATTRIBUTES

A logical video attribute system has been created which permits programs to run correctly on both monochrome and color screens without special code.

Use of logical attributes is now the default. If you want to run your old programs, which use physical attributes, without change, you will need to disable logical attributes. For more information on logical and physical attributes refer to Chapters 3 and 5.

EXPLICIT INITIALIZATION OF WINDOWS FOR C RECOMMENDED

You should make the first statement in your main programs a call to a new initialization function:

```
init_wfc();
```

Your existing programs will run without the addition of this statement, but we recommend that you use it in all new programs.

In Version 3.0, we introduced an internal initialization routine that assigned values to global variables used by the system. This routine was called automatically by output functions. Even in Version 3.1 it was possible to reference the system globals before they were initialized. With the introduction of logical attributes, the possibility has become a likelihood. For safety's sake, explicitly initialize **Windows for C**.

If you have previously included explicit initialization by calling the internal routine **_v_init()**, you do not need to change the code. Calling this function is equivalent to calling **init_wfc()**.

Changing the Initialization Values

To permit users to adapt the initialization to different display boards, the initialization routine **init_wfc()** calls **u_init()** at the end. At present, **u_init()** is simply a dummy program that returns immediately, but you can add code to this to alter the values assigned to any of the global variables or to initialize additional variables of your choosing. Function **u_init()** is supplied as a source file.

DETECTION OF THE IBM ENHANCED GRAPHICS ADAPTER BOARD

The initialization routine checks to see if the Enhanced Graphics Adapter board is present and active. If it is, the global variables **_ibmega** and **no_retr** are set to one. When **no_retr** is set to one, screen output is made without waiting for video retrace (as is necessary in color modes with the IBM Color/Graphics Adapter).

If you are supporting a non-IBM board that does not require output during retrace, set **no_retr** to one in **u_init()**.

STRUCTURE CHANGES

WINDOW Structures

Five new members have been added to the **WINDOW** structure: **char *wname, char *larray, char *pu_storp, char bdratt,** and **char popup.** These new members have been added after the two user reserved pointers, **char *userp[2].**

If you have added your own members after **wn.userp[2]**, there will be a conflict and it will be necessary to move your own members after the new members that we added and then recompile all of your code.

For a description of all **WINDOW** structure members refer to Appendix 5.

FREC Structures

Four new members have been added at the end of the FREC structure: **FLINEPTR *far-ray, int fmaxline, int ftabq,** and **int fmaxcol.**

If you have added your own members to the **FREC** structure, there will be a conflict and it will be necessary to move your own members after the new members that we added and then recompile all of your code.

For a description of all **FREC** structure members refer to Appendix 5.

INITIALIZATION OF wn0

The initialization of wn0 is no longer done explicitly in window.h. The initialization has been moved to the initialization routine **init_wfc().**

USE OF TYPES OF NULL POINTERS

We have tried to consistently replace **NULL** in our functions by either **NULLP**, for null character pointers, or **NULLFP** for null pointers to functions. We have done this to avoid errors in compiler memory models that mix large and small data and function types (for example, the Lattice **data** model). See **wfc_defs.h** for definitions.

NEW FUNCTIONS

```
csr_hide()  -- hides cursor off-screen
csr_show()  -- restores cursor to previous position from off-screen
csr_type()  -- changes shape of cursor
def_fr()    -- defines initial values for a FREC structure
di_file()   -- reads a file from disk into a memory file
dup_wn()    -- duplicates the member values of a WINDOW structure
file_lnp()  -- returns a pointer to the string associated with the i-th line in a
               memory file
free_file() -- frees memory allocated for a memory file
free_mem()  -- frees memory with error checking
get_mem()   -- allocates memory with error checking
init_wfc()  -- initialization function
lower_st()  -- converts a string to lower case
menu2()     -- improved pop-up menu function
mod_wn()    -- modifies coordinates of window
mv_cs()     -- moves virtual cursor within a window
pl_mfwn()   -- places window origin at specified location in memory file
scrl_file() -- scrolls a memory file
skip_wh()   -- skips the leading white space in a string
stblank()   -- allocates memory for a string;  initializes to blanks
sti_file()  -- stores string information in a memory file
strcpyp()   -- copies source string to destination string; returns pointer
strip_wh()  -- strips trailing white space from string
s_latt()    -- support function for logical video attributes
upper_st()  -- converts a string to upper case
vo_att()    -- reads an attribute from the window
vo_ch()     -- reads a character from the window
vs_file()   -- views a memory file and permits scrolling
v_att()     -- writes an attribute to a window
v_border()  -- displays a border around a window; writes window name in border
v_ch()      -- writes a character to a window
```

```
v_file()    -- views a memory file within a window
v_mova()    -- moves characters from a window to standard ASCII string and vice-
               versa
v_plst()    -- writes a string at the specified location in a window
v_printf()  -- formatted string output within a window
```

NEW MACROS

Many new macros have been added which simplify setting the individual members within WINDOW structures. See Chapter 3.

OBSOLETE FUNCTIONS

Several functions available in Version 3.1 are no longer used internally or have been replaced by new functions. We have eliminated these functions from the documentation. With the exception of **cs_add()**, which was really an internal function that has not been used for some time, the obsolete functions have been retained in the library.

Obsolete function	Replacement function
di_f()	di_file()
v_f()	v_file()
v_bdr()	v_border()
menu()	menu2()
cs_add()	-------
set_cwn()	set_wn()

Obsolete functions will be dropped from the library on the next major upgrade. If you wish to retain use of a function after is dropped from the current library, use a library manager to extract the function from the previous version of the library and insert it into the new version of the library.

We **strongly** recommend that you change to the new functions in new programs. They have major advantages over the functions they replace.

BUG FIXES

cls() attempted to clear a 80 column screen even in 40 column video modes.

ki_chk() did not properly detect the CTRL-BREAK key. This affected the operation of ki_cum().

k_vcom() did not properly place the cursor if the top-of-file or bottom-of-file message appeared in a 1 row window.

k_vcom() permitted the cursor to move beyond the end of the file if the file length was shorter than the display window.

k_vcom() always positioned the cursor to the first line when **<K_UP>** was pressed and the top-of-file message appeared in the window.

pl_wn() allowed the window to be placed 1 column too far to the right which resulted in the right most column being placed off-screen. This caused the column to be wrapped around to the left side of the screen. **v_st_nop()** improperly handled output if the virtual cursor was not in the first column in the window. The string would write beyond the right boundary of the window.

v_st_rw() improperly word wrapped a line if a newline was the first character beyond the word wrap boundary. This resulted in an extra blank line being displayed in the window.

CHANGES FROM VERSION 2.2 to 3.1

The major changes made between Versions 3.+ and Version 2.2 are related to improving the portability of **Windows for C.**

Most of the portability-related changes are in internal routines and do not involve changes in functions available to the user. Some changes, however, will be noticeable to the user and may require changes in previously coded user programs:

WINDOW STRUCTURES

Window coordinates and virtual cursor values have been changed from type **char** to type **int**. This change was made to permit use of **Windows for C** with screens greater than 128 columns and still allow error checking for negative window dimensions or virtual cursor locations. The change to type **int** will not require changes in any user C routines, but if assembler routines reference window-structure values, the offsets will need to be revised.

The WINDOW element wn.reserv1 is now utilized in internal routines, with the name wn.location. This element should always be assigned the value 0, as is now done by def_wn() and defs_wn(). WINDOW structures that are assigned initial values in the declaration statement must also give the value 0 to wn.location. In future multi-tasking versions, this parameter will be used in connection with providing off-screen window buffers. At present, the routines for this capability are not implemented. Attempts to write output with wn.location not equal to zero will cause the program to abort.

TOPVIEW COMPATIBILITY

Versions 3.1+ of **Windows for C** are fully compatible with TopView, IBM's multi-tasking operating programs. Programs that rely on **Windows for C** for screen output can operate in the background mode under TopView and can make use of TopView's windowing functions.

Screen compatibility with TopView is handled automatically by **Windows for C.** **Windows for C** detects operation in the TopView environment and adjusts its screen handling to conform to the requirements of TopView.

Although no special programming is required to make **Windows for C** compatible with TopView, the speed of screen updating under TopView can often be improved by tailoring the screen updating procedures to a specific application. **Windows for C** allows you to easily change the default video updating procedures built into library functions.

For a complete discussion of using **Windows for C** under TopView, see the new Chapter 9 in the Reference Manual entitled, **TopView Compatibility Features.**

FUNCTION CHANGES

New Options in the String Output Functions

v_st_rw(), v_st(), and v_fst() in prior versions always advanced the virtual cursor and cleared to the end of the window row, even if the string was shorter than the

remaining spaces in the window. In response to popular demand, switches are now available in the WINDOW structure to disable advance of the virtual cursor and to disable clearing to the end of the row in these functions.

Another switch is provided to enable automatic placement of the screen cursor at the location of the virtual cursor upon return from these functions. The ability to disable advance of the virtual cursor is useful primarily for writing output to status lines or comparable displays, where the string to be written is less than one window row in length.

Functions v_st() and v_fst() will not work properly with cursor advance disabled if the string will not fit on the present row; thus caution must be exercised in setting these switches.

See the revised reference page for v_st_rw() for details on implementing these switches.

New String Output Function

Checking for the various options now provided slows down v_st_rw() and the other string output functions. This is not noticeable except when viewing files, using v_f(). To improve speed of v_f(), a new string output function has been added: v_st_nop(), where "nop" stands for "no options". This function does just what is required in v_f() and no more. v_f() has been changed to call v_st_nop() and also to call v_qch() instead of v_rw() to write spaces to blank rows. The source for v_f() is provided.

Scrolling Changed

The way in which scrolling was implemented in demo_wn and dem_menu was changed. Formerly multiple keystrokes were collected from the buffer and implemented at once when scrolling did not keep up with the demand. Now identical keystrokes are removed from the buffer after each scroll operation, but only one of these is then implemented. This provides a smoother, visually more pleasant scroll. See the discussion of ki_cum() in demo_wn.

Macros Added

c_att() has been changed from a function to a macro (in bios.h).

Macros have been added to calculate the number of columns and rows in a window. See **col_qty(wnp)** and **row_qty(wnp)** near the end of wfc_defs.h. Note that these macros require the input variable to be of type WINDOWPTR.

BUG FIXES

itoa() in dem_menu returned the wrong number in early copies of Version 2.20. (Fixed late in July).

copy_wc() called a non-existent subroutine in early copies of Version 2.20. (Fixed late in July).

prt_labl had an error in early copies of Version 2.20. (Fixed late in July).

menu.c omitted from early copies of Version 2.20. (Corrected 2 August.)

pl_csr() and **mv_csr()** were in wrong order in DeSmet library (which requires that later functions not call earlier functions). (Corrected in Version 2.21.)

c_att() did not work. (Corrected in Version 2.21.)

copy_wc() did not copy correctly. (Corrected in Version 2.21.)

v_qch() did not write extended character set properly for CI C86. (Corrected Serial # 2C161.)

Output to non-zero pages of the Color Graphics Adapter card was not implemented properly. (Corrected in Version 3.0.)

mv_wi() did not erase borders of previous image. (Corrected in Version 3.1)

Minor bugs in **v_natt()**, **v_rw()**, **mv_rws()**, and **v_st_rw()** were corrected in Version 3.0.

CHANGES FROM VERSION 2.1

Several functions available in Version 2.1 have been eliminated. In all cases, other functions can be substituted to accomplish the same purpose. If you wish to retain the function available in a previous version, use a library manager to move the function to the new **Window** library.

Eliminated functions	Replacement functions
new_att( )	v_natt( )
setcsr( )	pl_csr( )
v_att( )	v_natt( )
v_ch( )	v_qch( )
v_hr_ch( )	v_qch( )
v_mov_rw( )	v_mov( )

Bug Fixes

Bugs appearing in Versions 2.0+ and 2.1+ have been fixed. Functions affected:

mv_rws, vid_int, v_mov, menu, v_qch, di_f.

Some of these bugs are minor, but others involve hidden problems that could cause problems later. Recompilation with the new library functions is recommended.

Returns Changed from NULL to Zero

In previous versions, a number of functions that return integers (**adj_cs()**, **set_wn()**, **v_axes()**, **v_bar()**, **v_co()**, and **v_rw()**) were programmed to return NULL when an error occurred. This was at variance with standard C nomenclature, which uses NULL only for the special (zero) pointer. It also created some confusion with respect to the Lattice suggestion that it may be necessary to define NULL as a long integer to make code compatible with long integers. To do this would create no problems with the NULL-returning functions of **Windows for C**, but it might appear to do so.

In Version 2.2, the integer-returning functions that formerly returned NULL now return zero to indicate an error condition.

This page intentionally left blank.

Chapter 1

OVERVIEW OF WINDOWS FOR C

CONTENTS

This chapter describes the features and capabilities of **Windows for C**. The overview will help guide you through the rest of the reference manual. **Windows for C** has many options and capabilities, each of which needs to be covered one at a time in the manual. The overview will place the initial sections of the manual in a helpful context and also point you toward sections of particular interest.

This page intentionally left blank.

OVERVIEW OF WINDOWS FOR C

Windows for C provides an integrated set of window-based functions that simplify all common screen display tasks.

Capabilities include:

* Unlimited windows and files
* Horizontal and vertical scrolling
* Pop-up windows, menus, and help files
* Status line management
* Off-screen buffers
* Rapid screen changes
* Frugal memory use
* Microsoft Windows and TopView Compatibility
* Logical video attributes
* Color-control
* Highlighting
* Window names
* String output with word wrap and auto scroll
* Formatted string output
* Print windows
* Keyboard input
* Read screen attributes and characters
* Plus a library of over 80 building block subroutines

Managing windows: There is no limit to the number of windows that can be established. Each window is controlled by a C-language structure that contains all the information needed for managing the window. Reference to the window structure substitutes for long parameter lists in function calls. This simplifies coding, reduces errors, and saves time.

Displaying files: Routines are provided for reading ASCII files into memory and viewing them through windows. A cursor-pad interpreter permits scrolling through the file horizontally or vertically. These routines, for which source is provided, will be especially helpful for providing users with access to help files. Context-sensitive help systems are easily implemented.

Pop-up windows: Windows can either pop-up onto or overwrite the screen. Pop-up windows are implemented simply by setting a switch in the window structure. Underlying screen contents are automatically saved and restored for pop-up windows.

Pop-up menus: Ready-to-use routines are provided for pop-up menus. The menu can be larger than the window. The cursor keys are used to select an item. Selections are highlighted. Source is provided so that the routines can be modified and enhanced to meet individual needs.

Status-line management: Status lines are easily maintained as single-line windows. Special options in the string output functions simplify the task of updating information in the status line.

Off-screen buffers: Facilities are provided for managing and displaying memory files. These files, which can be of any size, can be updated dynamically. They provide a superior alternative to virtual screens for the capture and display of real-time information.

Display speed: Writing to the screen is done via our own interface routines that have been designed to achieve speed without sacrificing clarity.

Memory usage: The only memory initially required by a window is that required by the window structure -- about 50 bytes.

Memory is used only when needed. All functions are in separate modules; so only those used are linked into the executable program. Because of efficient design, the code is compact. Typically, programs that make extensive use of the **Windows** library will increase in size by 15 to 20 kilobytes.

Memory is allocated for window buffers only when off-screen storage of window contents is needed, such as when a pop-up window is displayed. Only the amount of memory needed at the moment is allocated. When no longer needed, buffer memory is released.

Microsoft Windows and TopView Compatibility: Windows for C (and **Windows for Data**) are fully compatible with **Microsoft Windows** and IBM's **TopView**. Programs that use Windows for C for screen output and keyboard input can operate within a window in MS Windows and TopView and run in the background.

Programs built using **VCS Windows** will automatically run under TopView and MS Windows. You do not need to buy Microsoft's or IBM's Programmer's Toolkits or incorporate any special code in your programs. Compatibility is handled automatically by **Windows for C**, which detects the presence of MS Windows or TopView and adjusts its screen handling to conform to their requirements.

Logical video attributes: Professional programmers will appreciate the availability of logical video attributes in **Windows for C.** A single program can be written that uses the full color capabilities of the IBM Enhanced Graphics Adapter and the more limited colors available with the standard Color/Graphics adapter, while still being completely legible on monochrome displays.

Color-control: Functions provide complete control over color capabilities of the IBM PC family. The colors of window borders and contents can be set individually. A Window will automatically be cleared to the background attribute specified for that window.

Highlighting: Highlighting is easily accomplished using a function that changes the attribute without affecting the character contents of specified portions of a window.

Writing to windows: A variety of string and character output functions are available. Switches are provided to control word wrap, auto scrolling on full screen, automatic updating of the cursor, screen cursor placement, and auto clearing to the end of a row.

Formatted output: Formatted string output, equivalent to **printf()**, is provided. Screen output is much faster than that provided by compiler-supplied printf() routines, which utilize DOS function calls to write to the screen.

Printing windows: Contents of windows can be copied to standard ASCII files or to the printer. The print-window function simplifies printing text in unusual formats.

Reading the keyboard: A function is provided that reads the keyboard and returns the entire IBM Extended ASCII key set with a single call. The keyboard buffer can also be checked for an available keystroke.

Reading the screen: Characters and attributes can be read singly, or the character-attribute contents of specified parts of a window can be read in a single call.

IBM Enhanced Graphics Adapter support: A global variable is set when the IBM EGA is the active display adapter. When the EGA is active, output to the screen in color modes is as fast as with the Monochrome Display Adapter.

Building blocks: A library of over 80 building-block subroutines allows you to modify the supplied modules and to construct new integrated routines to suit your needs.

PRACTICAL ROUTINES AND LEARNING AIDS

In addition to tutorials and code examples included in the reference manual, several integrated routines are provided that can be used without modification to accomplish common tasks of window management. These routines are incorporated in demonstration programs that illustrate their use and serve as teaching tools for the use of **Window** functions. C source to the library level is provided for these programs.

* **demo_wn** reads multiple ASCII files into memory, displays them in multiple windows, and provides the user with cursor-pad control for viewing the files. Complete C source is provided for the file-input and cursor-control routines.

* **dem_menu** reads an ASCII file into memory and provides the user with the ability to call up a menu and select an item. Cursor pad commands are used to move through the menu, which can be larger than the menu-display window. Menu items pointed to are highlighted with reverse video. After selection, the menu is automatically removed and the original screen contents replaced. The item selected is reported on a status line.

* **dem_cmov** demonstrates the color management and window storage and movement capabilities of **Windows for C.** Colored windows are moved rapidly around the screen. The movement of images is useful for games, but the rapid speed with which windows can be stored and restored is also valuable for business applications.

* **prt_labl** reads addresses stored sequentially in a file and places them on the screen in side-by-side format. A print-window function is then used to print the windows in this format. This program illustrates how **Window** functions can be use to simplify printing in unusual formats.

* **dem_grph** illustrates the use of the graphing functions included in **Windows for C.** Vertical and horizontal bar graphs are drawn.

This page intentionally left blank.

Chapter 2

GETTING STARTED

CONTENTS

ORGANIZATION OF MATERIAL
- Organization and Contents of the Diskettes
  - The System Diskette
  - Library Files
- Where to Find Information in This Manual

COMPILING AND LINKING YOUR PROGRAMS
- Include Files Required by Windows for C
- Referencing the #include Files in Your Programs
- Linking Window Libraries with Your Programs
- A Batch File for Compiling and Linking

USING PHYSICAL ATTRIBUTES

SOME IBM COMPATIBLES ARE INCOMPATIBLE WITH TOPVIEW AND MS WINDOWS

This page intentionally left blank.

GETTING STARTED

This chapter provides essential information on using **Windows for C** and provides a guide to finding information in this manual. Everyone should review this chapter.

ORGANIZATION OF MATERIAL

ORGANIZATION AND CONTENTS OF THE DISKETTES

Windows for C comes on either one or two diskettes:

* A **system diskette** for **Windows for C** and, in some cases,
* A **library diskette** for **Windows for C**

If you have a version of **Windows for C** that has libraries for only one or two compiler memory models, the libraries may be included on the system diskette, in which case you will have one diskette for **Windows for C.**

The original diskettes should be copied and stored in a safe place. Use the copy for working purposes.

The System Diskette

The system diskette for **Windows for C** contains:

* **read.aaa**, which reports the changes that have occurred since this manual was published
* the include files listed in Appendix 1
* source code and ASCII files for the library functions and tutorials listed in Appendix 3.
* source code for the demonstration programs listed in Chapter 8.

For a complete listing of the contents of the system diskette, see Appendix 6.

Library Files

The library files may be either on your system diskette or a separate library diskette.

Separate libraries are provided for each memory model supported by your compiler.

The initial letters of the libraries for **Windows for C** are **wn**.

When there are different memory models, the initial letters will be followed by the letter used by the compiler authors to identify the model. For example, for the large-data model of Lattice, the appropriate **Window** library **wnd.lib**. For the C86 big-memory model, the **Window** library is **wnb.lib**. This method of identification allows you to set up standard batch files for processing the different memory models.

WHERE TO FIND INFORMATION IN THIS MANUAL

Chapter 1 provides an overview of **Windows for C.**

Chapter 2 provides the essential information that you need to incorporate the facilities of **Windows for C** in your programs. This chapter is essential reading for everyone.

Chapter 3 covers all of the basics of **Windows for C.** Everyone, including those who are already familiar with **Windows for C** should read this chapter. Many new features and capabilities of Version 4.0 are introduced here. This is the place to start when looking a for a way to do something with **Windows for C.**

Chapter 4 provides a tutorial on using the functions introduced in Chapter 3.

Chapter 5 explains how to control color. The **logical attribute** system of **Windows for C** is explained. Logical attributes allow you to write one program that will use color on color systems and monochrome attributes on black and white systems. This chapter also describes the facilities provided for controlling color through physical attributes.

Chapter 6 explains how to build and open windows on **memory files.** With memory files, you can retrieve and display text files stored on disk, and you can capture real-time data for later display. Memory files can be created and edited internally to programs. Windows on memory files can be scrolled either horizontally or vertically.

Chapter 7 illustrates the use of memory files to provide pop-up help files and menus and to manage off-screen buffers for storage of real-time information.

Chapter 8 covers advanced topics, utilities, and demonstration programs

Chapter 9 explains the IBM TopView and Microsoft Windows compatibility features.

Tables and Listings contains all of the tables and listing referred to in the text chapters.

Appendix 1 explains the structure of the include files and lists the most important of them.

Appendix 2 provides reference pages for the functions of **Windows for C.** The reference pages are in alphabetical order by name of function, except for string utilities, which are listed under **stringf.** Listings of the functions both alphabetically and by category of use are provided in an initial section of Appendix 2.

Appendix 3 lists the source files provided on the system diskette. These include tutorials, demonstration programs, and library functions. Source is provided for functions involved with file management so they can be modified for your own information structures.

Appendix 4 provides definitions of the variables and mnemonic abbreviations used in calling **Window** functions.

Appendix 5 provides a detailed guide to the structures that are used to manage windows and memory files within **Windows for C.**

Appendix 6 explains the error handling system of **Windows for C.**

Appendix 7 lists and briefly describes the files included on the system diskette.

COMPILING AND LINKING YOUR PROGRAMS

INCLUDE FILES REQUIRED BY WINDOWS FOR C

All of the **include** files (**.h** files) on the **Windows for C** system diskette need to be accessible to your compiler when you compile programs that reference the functions of **Windows for C.**

REFERENCING the #INCLUDE FILES IN YOUR PROGRAMS

Two top-level **include files** reference all definitions and global-variable declarations required by **Windows for C.** The rules for **#including** these files in the programs that you write are:

File **bios.h** must be **#included** in all modules that call **Window** library functions. It provides typedef definitions of the data structures used by **Window** library functions and **#defines** parameters that are useful in calling the library functions.

File **window.h** must be **#included** in the main program only, following **bios.h**. It contains all of the globally defined variables and structures used by **Window** functions. **Extern** declarations for these global variables are contained in **vextern.h**.

Users of Windows for Data, take note: the top-level include files provided with **Windows for Data** reference the include files of **Windows for C.** File **wfd.h** includes **bios.h**, and **wfd_glob.h** includes **window.h**; thus you **should not** place separate include statements for **bios.h** or **window.h** in data-entry functions or main programs.

LINKING WINDOW LIBRARIES WITH YOUR PROGRAMS

The libraries of **Windows for C** have the same format as the libraries supplied with the supported compiler version. To include them in the linking process, simply add the appropriate **Window** library names to the list of libraries in the link statement.

CAUTION: the Windows for C library must precede the standard compiler library in the linking statement.

For example, with the Microsoft linker, the following will link "filename" with the small-memory-model libraries of **Windows** and Lattice C:

```
(d:)link (d:)cs + (d:)filename, (d:)filename, , (d:)wns+ (d:)lcs
```

where **(d:)** stands for the drives (and paths) where the linker, compiler, object modules, and libraries are located. The values of **(d:)** may be different for the various components of the link statement.

WARNING: If you do not place the libraries in the correct order, you will get the message: UNRESOLVED EXTERNALS.

A BATCH FILE FOR COMPILING AND LINKING YOUR PROGRAMS

The system diskette provides a sample batch file, **CL.BAT**, specific to your compiler, for compiling and linking a program. This batch file assumes that everything is on the default drive. Modify it by assigning the proper drive and path designations.

Use this batch file to run the demonstration programs and tutorials included on the system diskette (see Appendix 3 for a listing).

USING PHYSICAL ATTRIBUTES

Windows for C is set up to use logical attributes as the default. If you have programs that you coded previously with **Windows for C**, they assume use of physical attributes. See Chapter 3 for instructions on enabling physical attributes.

SOME IBM COMPATIBLES ARE INCOMPATIBLE WITH TOPVIEW AND MS WINDOWS

Windows for C contains code to make it completely compatible with **TopView** and **Microsoft Windows**. This can create problems with some IBM compatibles, which use for other purposes an interrupt number that TopView and MS Windows use. If this is a problem for you, see Chapter 9 on how to disable TopView and MS Windows compatibility.

See the **READ.AAA** file on the system diskette for the names of computers for which we have had reports of problems.

Chapter 3

WINDOW BASICS

CONTENTS

DIRECT DISPLAY AND MEMORY FILE DISPLAY

THE FIRST WINDOW PROGRAM: "HELLO, WORLD"

SIX STEPS TO WINDOWS
- Step 1: Include System Files
- Step 2: Declare a Window
- Step 3: Initialize Windows for C
- Step 4: Define Initial Values of the Window
  - Border Types
  - Default Window Settings
- Step 5: Set the Window On The Screen
- Step 6: Write To the Window
  - String Output: v_st()
- Recap

CHANGING DEFAULTS AND USING OPTIONS
- Definitions, Usage, and Abbreviations
- Set-Window-Member Functions
  - Direct Assignment and Macros
  - Window Initialization Required
- Making Pop-up Windows
- Naming a Window
  - Removing a Window Name
- Changing Window Margins
- Changing Word-wrap, Auto-scroll and Cursor Placement Options
- Changing Special Options
- Modifying Window Size and Location

CONTROLLING THE APPEARANCE OF OUTPUT: ATTRIBUTES
- Physical Attributes
  - Monochrome Attributes
  - Color Attributes
  - Problems Created by Physical Attributes
- Logical Attributes
- Using Physical Attributes
- Physical and Logical Attributes Don't Mix
- Changing the Attribute of Output
  - Changing Window Attributes
  - Changing Border Attributes

DISPLAYING AND REMOVING WINDOWS
- Displaying a Window On the Screen
- Removing Windows from the Screen

CONTROLLING THE LOCATION OF OUTPUT
The Virtual Cursor
Working Dimensions and the Origin for Measuring the Virtual Cursor
Moving the Virtual Cursor
Direct Assignment of the Virtual Cursor Location
Macros for Row and Column Size of Windows
Controlling the Screen Cursor
Automatic Placement of the Screen Cursor
Direct Placement of the Screen Cursor
Reading the Location of the Screen Cursor
Hiding and Restoring the Screen Cursor

WRITING TO WINDOWS
Basic String Output: v_st()
Full-string Output Function: v_fst()
Writing a String at a Specified Location: v_plst()
Centering, Left-Justifying and Right-Justifying Text
Formatted String Output: v_printf()
Writing Characters
Single Characters
Rows and Columns of Characters
Filling a Window with a Specified Character
Sounding the Bell (Beep)
Changing the Attribute of a Character
Scrolling Direct Display Windows

READING THE CONTENTS OF WINDOWS
Reading a Character
Reading an Attribute
Reading the Character Contents of Parts of the Window
Reading the Character-attribute Contents of Parts of the Window

READING THE KEYBOARD
Keycode Conventions
The Read-Keystroke Function: ki()
Executing Subroutines While Waiting for Keystrokes
Creating a Pause in a Program
Checking the Keyboard Buffer: ki_chk()

CLEARING THE SCREEN AND WINDOWS
Clearing the Screen
Clearing Windows
Removing a Window from the Screen
Controlling The Color of the Screen Background

USING WINDOW FUNCTIONS ON THE FULL SCREEN
Saving, Clearing, and Restoring the Original DOS Screen
Moving, Saving, and Restoring the Screen Cursor
Moving the Screen Cursor
Saving and Restoring the Screen Cursor
Writing To the Full Screen

ANOTHER VERSION OF "HELLO, WORLD"

PRACTICAL EXAMPLES OF WINDOWS
An Error Message Window
Establishing a Status Line

WINDOW BASICS

This chapter explains how to establish, display, write to, and modify windows. It contains the basic information that you will need to use **Windows for C** and to understand the remaining chapters.

DIRECT DISPLAY AND MEMORY FILE DISPLAY

Windows for C has two approaches to displaying output in windows: 1) writing directly to a window on the screen and 2) writing information to a **memory file** and then opening a window onto the file. **Direct display windows** have the virtues of simplicity, speed, and minimum memory usage. Advantages of **memory file windows** are: text files stored on disk can be displayed; information for display can be stored and updated off-screen; the underlying file can be of any size, and the window can be scrolled horizontally or vertically over the file. Both approaches use the same windows, but different functions are used to display output within the windows.

Both direct display windows and memory-file windows can be either **pop-up windows** or **overwrite windows**. Pop-up windows preserve the underlying screen and restore it when removed. The memory required to save the underlying image is released when the window is removed.

This chapter deals only with **direct display windows**, that is windows in which writing is done directly to screen. This allows us to concentrate on explaining the basic features of the window system. These basic features will apply, in large part, whether you are writing directly to windows or displaying memory files in windows. Chapters 5 and 6 explain how to manage memory files and view them through windows.

THE FIRST WINDOW PROGRAM: "HELLO, WORLD"

Kernighan and Ritchie note at the beginning of "The C Programming Language:"

> The only way to learn a new programming language is by writing programs in it. The first program to write is the same in all languages:
>
> Print the words
>
> **Hello, world**

They note that once the mechanical details of doing this are mastered, "everything else is comparatively easy." Lets proceed to clear this basic hurdle right away.

The simplest program that writes "Hello, world" in a window -- call it hello_wn.c -- is:

```
#include <bios.h>
#include <window.h>

main()
{
     WINDOW wn;

     init_wfc();
     defs_wn(&wn, 10, 30, 8, 25, BDR_DLNP);
     set_wn(&wn);
     v_st("Hello, world\n", &wn);
}
```

The program illustrates all of the six **Window** steps. Try it on your system. You will find it helpful to run the program before reading the following explanation of the program.

SIX STEPS TO WINDOWS

The "Hello, world" program illustrates the six simple steps to introducing windows into your programs:

1. Include the system files
2. Declare a window
3. Initialize **Windows for C**
4. Define initial values of the window
5. Set the window on the screen
6. Write to the window

STEP 1: INCLUDE SYSTEM FILES

The program begins with two **#include** statements. The rules for including these files are:

> File **bios.h** must be **#included** in all modules that call **Window** library functions. It provides **typedef definitions** of the data structures used by **Window** library functions, **#defines** parameters that are useful in calling the library functions, and provides **extern declarations** for the system global variables.
>
> File **window.h** must be **#included** in the main program only, following **bios.h**. It contains all of the globally defined variables and structures used by the system.

Users of Windows for Data, take note: The two top-level include files provided with **Windows for Data**, **wfd.h** and **wfd_glob.h**, reference **bios.h** and **window.h** within them. You **should not** place separate include statements for **bios.h** or **window.h** in data-entry functions or main programs. The initial lines of **hello_wn** would read:

```
#include <wfd.h>
#include <wfd_glob.h>
```

STEP 2: DECLARE A WINDOW

A data structure is used to contain all of the information needed to manage a window. Declaring the variables used to represent these structures is no more difficult than declaring a variable to be of type **int**. To declare "wn" to be a window structure, use the declaration:

```
WINDOW wn;
```

WINDOW is a typedef specifier for window structures. It is not necessary to know the details of this structure to use most **window** functions.

STEP 3: INITIALIZE WINDOWS FOR C

The first statement in your main program, after the declarations, should always be:

```
init_wfc();
```

This function call initializes all of the global variables used by the window system. These include variables for the size of the screen, the video mode, and the logical attribute system.

If you do not call this function explicitly, **Windows for C** will call it internally when you call an output function, but you could easily attempt to use the system globals before this, creating hard-to-detect errors.

For safety's sake, always make **init_wfc()** the first statement in your main program.

STEP 4: DEFINE INITIAL VALUES OF THE WINDOW

The initial values of a window can be defined most simply by using a **window initialization function**. Two of these are provided in the window library: **def_wn() and defs_wn()**. "Hello, world" used **defs_wn()**.

```
defs_wn(&wn, row_beg, col_beg, row_q, col_q, BORDER_TYPEP);
```

The arguments passed to this function are:

1) A pointer to a WINDOW structure ('&' signifies a pointer)
2) The beginning screen row of the window (0 is at the top)
3) The beginning screen column of the window (0 is first column)
4) The quantity of rows in the window
5) The quantity of columns in the window
6) BORDER_TYPEP is one of the pre-defined border pointers

Border Types

Borders are controlled by special structures that contain the characters used to draw the borders. Pointers to the following border types are #defined:

Pointer Name	Border Type
BDR_LNP	Single line border
BDR_DLNP	Double line border
BDR_ØP	No (zero) border
BDR_REVP	Reverse border

Default Window Settings

There are other elements in the WINDOW structure that are not specified by the parameters in **defs_wn()**. Values not assigned directly by an initialization function are assigned **default values.** You will learn about these values and how to change them in following sections.

STEP 5: SET THE WINDOW ON THE SCREEN

After a window is declared and defined, it is placed on the screen and made ready to receive output by the statement:

```
set_wn(&wn);
```

The function set_wn() clears the area of the screen where the window will appear, draws the border, and reduces the "working dimensions" of the screen by the width of the borders and by the width of the margins specified in the WINDOW structure. Function **defs_wn()** sets the margins between the text and borders at 1 column on both the left and right sides of the window.

Window output functions operate within the working dimensions of a window. Thus after a window is "set," text will automatically be written within the margins established for a window.

STEP 6: WRITE TO THE WINDOW

String Output: v_st()

The function, **v_st()**, which was used to write the "Hello" message, is the basic **Window** function for writing a string to window. Underneath this simple function lies the output control system of **Windows for C.** Output can be done simply, as in "Hello, world," but many options and variations exist for controlling output. These are described in following sections.

RECAP

The "Hello, World" program consisted of six steps:

1. Including the system files **bios.h** and **window.h.**
2. Declaring a window to be of type **WINDOW.**
3. Initializing **Windows for C** by calling **init_wfc().**
4. Defining the initial values of the window using **defs_wn().**
5. Setting the window on the screen by calling **set_wn().**
6. Writing to the window with **v_st().**

The window in this program overwrites the contents of the screen, destroying what is underneath. In following sections, you will learn how to designate a window as a pop-up window and to change the other default settings used in this first window program.

CHANGING DEFAULTS AND USING OPTIONS

The operation of **Window** library functions are controlled by parameter values and variables stored as members of a WINDOW structure. The standard window initialization function, **defs_wn()**, allows you to specify the size, location, and border for a window. The alternative initialization function, **def_wn()**, additionally allows you to specify left and right margins for the window. The rest of the members in a window are set to default values.

This section introduces and explains the use of functions that are provided for changing the values of the window members.

DEFINITIONS, USAGE, AND ABBREVIATIONS

Variables and functions in **Windows for C** use a consistent set of mnemonic abbreviations. The most commonly used ones are described here. A comprehensive alphabetical listing of abbreviations is in Appendix 4.

v = video. **'v'** generally refers to output to a window on the video screen. For example, the function **v_st()** refers to output of a string to a video window.

wn = a type **WINDOW** data structure that holds the information for managing a window. In library functions and in text discussion, **wn** is generally used to refer to a window data structure.

Members of the **WINDOW** structure are addressed in the form **wn.x**, where **x** is one of the members. For example, **wn.rb** refers to the **screen** row-number of the top ("beginning") row of window **wn**. The top row of the screen is row zero.

wnp = pointer to a type WINDOW data structure. **wnp** can be declared using the typedef type-specifier **WINDOWPTR** (defined in **wfc_defs.h**). In a function call, **&wn** is equivalent to **wnp**.

In general, **p** is appended to a variable to refer to a pointer to the variable in question.

cs = the virtual cursor in a window. The virtual cursor is the position in a window at which the library functions will begin writing output. It differs from the screen cursor (abbreviated csr, see below) that appears on the screen. The screen cursor need not be located at the virtual cursor position to write at that point.

r = the row position of the virtual cursor, that is **wn.r** is the window-row number of the virtual cursor. The window row-number is measured relative to the top of the window, not from the top of the screen. Zero is the row-number of the first row of a window available for output to or input from a window.

c = the column position of the virtual cursor, that is **wn.c** is the window column number of the virtual cursor. Window column-numbers begin at the left-hand side of the window. Zero is the column-number of the first column available for output to or input from a window.

csr = the screen cursor, the blinking cursor that appears on the screen. The screen cursor is distinct from the virtual cursor of a window (see above).

bdr = refers to a type **BORDER** data structure. The typedef type-specifier **BORDER** is defined in **wfc_stru.h**. **bdr** contains the information necessary for drawing a border on a window.

bdrp = a pointer to a type **BORDER** data structure.

SET-WINDOW-MEMBER FUNCTIONS

A set of functions is provided for changing the values of the **WINDOW** structure members. The functions have the general form: **sw_member()**, where **sw** is a mnemonic for "set window" and **member** is a mnemonic for the window member (or members) affected.

Table 3.1 lists the **set-window-member** functions, the names of the related **WINDOW** structure members, when appropriate, and the default values assigned by **defs_wn()**. **Tables are in Tables and Listings, a separate section which follows the text chapters and precedes the appendices.** The interpretation and use of these functions is explained in the remainder of this section, with the following exceptions: the two functions that allow you to change **attributes** are covered in the following section; the function for referencing a memory file in a window is covered in Chapter 5.

Direct Assignment and Macros

You can change the values of the window members by direct assignment rather than using the **set-window_member** functions. For example:

```
wn.rb = 0;
```

will set the beginning row of window **wn** to 0.

Note that the set-window-member functions are implemented as macros. Direct assignment will produce the same code as use of the macros because the macros will be expanded into in-line assignment statements by the preprocessor.

Window Initialization Required

The following discussion of the functions for changing window values assumes that the window has already been initialized by a call to **defs_wn()**. You must **always** initialize a window after declaring it and before using it, else you will write output to unknown locations in memory.

Windows that have the default values assigned by **defs_wn()** will be called **default windows.**

MAKING POP-UP WINDOWS

Default windows will overwrite the screen, destroying permanently what was there before. Windows can be made pop-up windows by the call:

```
sw_popup(ON, &wn);
```

The window management functions of **Windows for C** will automatically save the underlying image before displaying a pop-up window on the screen and will restore that image after removing the window. The memory allocated to store the underlying image will be released as soon as the image is restored.

NAMING A WINDOW

A window can have a name automatically printed in the border, starting at the top left-hand corner. To assign a name to a window:

```
char *name = "Window 1";
sw_name(name, &wn);
```

Or, more directly, you can write:

```
sw_name("Window 1", &wn);
```

Function **set_wn()** will automatically write the assigned name on the window border, starting at the top left-hand corner. If the window has no border (the border pointer is BDR_OP), the assigned name will not be written.

Removing a Window Name

If a name has been assigned and you want to remove it from the structure, use:

```
sw_name(NULLP, &wn);
```

NULLP is a #defined null pointer. When the **wname** member of the window structure is a null pointer, no name will be written on the border.

The default name value is a null pointer; so no name will appears on a window until you assign it a name.

CHANGING WINDOW MARGINS

The left and right margins have default values of 1 space from each side of the border. To change these margins:

```
sw_margin(left_margin, right_margin, &wn)
int left_margin;
int right_margin;
WINDOW wn;
```

Note that contrary to typewriter convention, margins are measured in spaces from the nearest border. For example,

```
sw_margin(0, 0, &wn);
```

would allow text to be written in all spaces between the borders of a window.

CHANGING WORD-WRAP, AUTO-SCROLL AND CURSOR PLACEMENT OPTIONS

Default windows have word-wrap and auto-scroll ON and cursor placement OFF. With these settings:

1) string output functions will always break on a even word;

2) if an attempt is made to write to a full window, the window will automatically be scrolled up one line;

3) the position of the screen cursor will not affected by string output functions.

To change these options, use:

```
sw_wwrap(OFF, &wn);          /*will do char. wrap, not word wrap */
sw_scroll(OFF, &wn);         /*will not write when window full   */
sw_plcsr(ON, &wn);           /*screen cursor at virtual cursor   */
```

When screen cursor placement is turned on, using **sw_plcsr()**, the blinking screen cursor will automatically be placed at the location of the window **virtual cursor.**

The virtual cursor does not appear on the screen. It is the location where the next output to the window will be placed. After a write function is called, the virtual cursor will be just after the last character written. The location of the virtual cursor is kept in the window structure.

CHANGING SPECIAL OPTIONS

Two special window options are provided to control the behavior of the string output functions: auto-clear to the end of row, and automatic advance of the virtual cursor.

The functions for initializing windows set auto-clear and virtual-cursor advance **ON.** When **v_st()** or another string-output function is called, the row on which output ends will automatically be cleared with spaces from the end of output to the end of the row. The virtual cursor will be placed just after the last character written. These settings are appropriate for most applications.

The special-purpose option settings are for special applications, such as maintaining a status line, where output is written to the same place repetitively. To set these options, use:

```
sw_cleor(OFF, &wn);          /*disable auto-clear to end of row  */
sw_csadv(OFF, &wn);          /*disable virtual cursor advance    */
```

Warning: Disabling these options can cause problems in many normal output functions. Exercise care.

For an example on using these functions, see the section later in this chapter on setting up and maintaining a status line.

MODIFYING WINDOW SIZE AND LOCATION

To change the size and location of a window that has been initialized, use

```
mod_wn(row_beg, col_beg, row_q, col_q, &wn);
int row_beg;                 /*beginning row       */
int col_beg;                 /*beginning col       */
int row_q;                   /*number of rows      */
int col_q;                   /*number of columns   */
WINDOW wn;                   /*structure           */
```

This function will leave all elements of a window unchanged except the origin and size of the window.

Note: This is implemented as a function call and not a macro.

Warning: Do not call this function when a window has been set on the screen but not yet removed. This call will not affect the size of the window on the screen, but will create errors when a call is made to remove it.

CONTROLLING THE APPEARANCE OF OUTPUT: ATTRIBUTES

Attribute is the technical term for the color or monochrome appearance of text on the screen. Monochrome and color attributes are controlled by the value of an **attribute byte** that accompanies each character written to the screen.

Windows for C uses an attribute value stored in the WINDOW structure to determine the attribute of text written within a window. A separate attribute is stored in the structure for the border. Attributes can be changed at any point in a program. Output written prior to the change will keep the old attribute, and output after the change will have the new attribute.

PHYSICAL ATTRIBUTES

In **Windows for C**, each character displayed on the screen has associated with it an **attribute byte** that determines its display appearance. The attribute-byte values are termed **physical attributes.**

Monochrome Attributes

The IBM Monochrome Display Adapter has three "base" attribute states: normal, reverse, and underline. Values are #defined for these attributes: NORMAL, REVERSE, and UNDERLINE. These base states can be modified by adding to them the values #defined for HIGH_INT (high intensity) and BLINK.

See, "Changing the Attribute of Output" later in this chapter for an example of how to use the #defined attribute values to set the attribute for a window.

Color Attributes

The IBM Color/Graphics and Enhanced Graphics Adapters have other sets of physical attribute values that control the color possibilities of these adapters. Color attributes are discussed in Chapter 5.

Problems Created by Physical Attributes

Use of physical attributes in programs creates problems. Not all color attributes will display properly on black and white monitors. If you use physical attributes in your programs, you must either restrict the color choices or write separate code for color and black and white video modes.

LOGICAL ATTRIBUTES

To eliminate the problems caused by physical attributes, **Windows for C** optionally implements a system of **logical attributes.** Logical attributes allow you to write one program that utilizes the full color capabilities of color monitors and is fully legible on monochrome monitors.

Logical attributes are categorized by function or purpose. For example, some of the pre-defined logical attributes used in **Windows for Data** are: LNORMAL, LHIGHLITE, LERROR, LMESSAGE. Each of these logical attributes has a color attribute and a monochrome attribute defined for it. When a computer is in a color mode, the

display functions will use the color attribute. When in a monochrome mode, the monochrome attribute will be used.

Table 3.2 lists the pre-defined logical attributes, together with associated monochrome and color attributes.

For a full explanation of logical attributes, see Chapter 5, Controlling Color with Logical and Physical Attributes. There you can find out how to change the physical attributes associated with given logical attributes and how to add logical attributes. The use of set-window-member function **sw_latt()** is also explained there.

USING PHYSICAL ATTRIBUTES

As you receive **Windows for C**, it is set up to use logical attributes rather than physical attributes in window structures and function calls. If you use physical attributes in the system as configured, you will create errors.

You can easily change to using physical attributes. Logical attributes are implemented in the system by a #define statement near the top of **bios.h**:

```
#define ATT_LOGIC              /*use logical  attributes          */
```

To use physical attributes, all you need to do is to comment out or remove this statement:

```
/*  #define ATT_LOGIC          use physical attributes          */
```

Alternatively, you can make a specific program use physical attributes by **undefining ATT_LOGIC** at the start of the program. Use the following sequence of statements:

```
#include <bios.h>
#undef ATT_LOGIC               /*set for physical attributes      */
#include <window.h>
```

Warning: The **#undef** statement must come between the two **#includes.**

PHYSICAL AND LOGICAL ATTRIBUTES DON'T MIX

Be aware that once you set the type of attributes that the system is to use, you must be consistent throughout the program. You cannot mix physical and logical attributes in a single program.

You should make your subroutines use physical or logical attributes conditionally, if you are going to make some programs that use physical and some that use logical attributes. When and only when logical attributes are being used, the global variable **_lattsw** will be 1. Use this in the conditional test, as in the following code fragment:

```
if(_lattsw)
      sw_att(LNORMAL, &wn);
else
      sw_att(NORMAL, &wn);
```

CHANGING THE ATTRIBUTE OF OUTPUT

Changing Window Attributes

The window-initialization functions set the window attribute to LNORMAL (or NORMAL, if physical attributes are specified for the system). All output to a window will have this attribute.

If you do not want the default attribute, you can change it with the "set window attribute" function:

```
sw_att(attribute, &wn) int attribute; WINDOW wn;
```

For example, if you are working with physical attributes, you could set the window to write normal text in high intensity with blink by the statement:

```
sw_att(NORMAL + HIGH_INT + BLINK, &wn);
```

You can use the "set window attribute" function to change logical as well as physical attributes. For example, this code fragment sets up a one-row "window" for error messages:

```
WINDOW wn_err;

defs_wn(&wn, 24, 0, 1, 80, BDR_0P);
sw_att(LERROR, &wn_err);
```

The window has no borders. It is the full width of the screen and appears on the bottom row of the IBM PC. Output to this window will have the attribute **LERROR** (see Table 3.2 for the physical attributes associated with **LERROR**).

Changing Border Attributes

A separate member in the window structure specifies the attribute of window borders. The default value is LNORMAL (or NORMAL, if physical attributes are specified for the system). To change the value, use the "set window border-attribute" function:

```
sw_bdratt(attribute, &wn);
```

For example, to make the border **LRED** (which will show up as **red** on **blue** in color monitors and **reverse** in monochrome modes):

```
sw_bdratt(LRED, &wn);
```

You must change the border attribute prior to calling a function that draws the border on the screen. If the window is already on the screen, changing the border attribute will have no effect until the border is redrawn.

In general, you will want to use **sw_bdratt()** before you call function **set_wn()**, which sets a window on the screen and draws the border. If the window is already on the screen and you want to change the border (either its type or its attribute), issue a call to:

```
v_border(&wn, bdrp)
WINDOW wn;                  window structure
BORDERPTR bdrp;             pointer to a border structure
```

For example, suppose window **wn** has been initialized with border attribute **LNORMAL** and a single line border. If after this window has been displayed on the screen, you want to give it a double line border with attribute LMESSAGE, you would use the following code:

```
sw_bdratt(LMESSAGE, &wn);
v_border(&wn, BDR_DLNP);
```

DISPLAYING AND REMOVING WINDOWS

DISPLAYING A WINDOW ON THE SCREEN

The same function is used to place either pop-up or overwrite windows on the screen:

```
set_wn(&wn);
```

This function clears the area of the screen where the window will appear, draws the border, and reduces the "working dimensions" of the screen by the width of the borders and by the margins specified in the WINDOW structure. Function **defs_wn()** sets the margins between the text and borders at 1 column on both the left and right sides of the window.

The output functions of **Windows for C** operate within the working dimensions. You can change the working dimensions to include the border if desired. See the reference page for **dim_wn()** in Appendix 2.

The virtual cursor is set to location 0, 0.

If a **window name** is specified, **set_wn()** will write the name over the top border, beginning at the left corner. For pop-up windows the underlying image is saved.

Function **set_wn()** sets window member **wn.setsw** to 1. You can test the value of **wn.setsw** to find out whether the window has been set on the screen.

REMOVING WINDOWS FROM THE SCREEN

To remove a window from the screen, use:

```
unset_wn(&wn);
```

If **wn** is a pop-up window, the underlying image will be restored; otherwise the area covered by the window will be cleared to the screen background. Memory used to store the underlying image for a pop-up window will automatically be released by this function.

This function restores the "working dimensions" of the window (the dimensions within which the output functions operate) to the full outside dimensions, including the margins and the borders.

Function **unset_wn()** sets window member **wn.setsw** to 0. You can test the value of **wn.setsw** to find out whether the window is on the screen or not.

CONTROLLING THE LOCATION OF OUTPUT

THE VIRTUAL CURSOR

Window output functions begin writing at the location in a window specified by the "virtual cursor". The virtual cursor is independent of the physical cursor that appears on the screen. A separate virtual cursor is defined for each window.

In the "Hello, world" program, output starts at row 0, column 0, because function **set_wn()** sets the virtual cursor location to that position.

The location of the virtual cursor is maintained as part of the information contained in the window structure. The location is automatically updated by the string output functions (unless automatic updating is disabled). With this system, output can be made to several different windows alternately without any need for the programmer to move the cursor location each time.

Working Dimensions and the Origin for Measuring the Virtual Cursor

The virtual cursor position is always measured relative to the origin of the window, as recorded within the window structure in members **wn.rb** and **wn.cb**, the beginning screen row and the beginning screen column of the window.

The window origin from which the virtual cursor location is measured depends upon whether the working dimensions of a window are FULL or INSIDE. The working dimensions are boundaries within which **Window** output functions will write. The working dimensions are changed automatically by **set_wn()**.

Before the window is set on the screen, its working dimensions are **FULL**; they include the border and margins. (To draw the border, the dimensions must be **FULL**.) After **set_wn()** draws the border, it changes the working dimensions to **INSIDE**. It does this by changing the values of the beginning and ending rows and columns of the window by the width of the border and margins. The values of the members in the window structure are actually changed.

When working dimensions are adjusted, the setting for the virtual cursor is not changed. Thus, if the location of the virtual cursor is 0, 0, it will be at the top left corner of the border when the dimensions are **FULL**, and it will be at the first position inside the borders and margins when the dimensions are changed to **INSIDE**.

When a window is removed from the screen (see below), its dimensions are automatically restored to **FULL**. A function, **dim_wn()**, is also provided to allow you to change the working dimensions (see Chapter 8 and the reference page in Appendix 2 for more information).

Moving the Virtual Cursor

To move the virtual cursor to the window location where you want to write, use

```
mv_cs(row, column, &wn);
int row;
int col;
WINDOW wn;
```

Note: the mnemonic used for the virtual cursor is always **cs** and that for the physical screen cursor is **csr**. Function **mv_cs()** does not affect the physical cursor. The similarly named function, **mv_csr()**, which has the same arguments as **mv_cs**, moves both the virtual and screen cursor.

To write a string at row 3, column 5 of a window:

```
char *string = "This is easy.";

mv_cs(3, 5, &wn);
v_st(string, &wn);
```

After **string** is written, **cs** will be located immediately after the last written character.

Direct Assignment of the Virtual Cursor Location

You can also locate the cursor by direct assignment:

```
wn.r = 3;
wn.c = 5;
```

This is equivalent to the call to **mv_cs()** made above.

Direct assignment is especially useful for iterations within **for** and **while** loops, eg:

```
wn.r = 0;
wn.c = 0;
while(wn.r++ <= wn.re - wn.rb)
{
      v_st("Write this on each row of the window", &wn);
      wn.c = 0;
}
```

Macros for Row and Column Size of Windows

Macros are provided that return the quantity of rows and columns in a window:

```
row_qty(&wn);
col_qty(&wn);
```

These are quite useful, as you will find that you need to know the size of a window for many purposes. The above **while** statement could have been written:

```
while(wn.r++ < row_qty(&wn))
```

CONTROLLING THE SCREEN CURSOR

Automatic Placement of the Screen Cursor

The position of the screen cursor is normally independent of the position of the virtual cursor position. With the default window settings, the screen cursor will not move when you call **v_st()** or other **Window** output functions.

You can, however, change the default window setting so that the output functions automatically place the screen cursor at the virtual cursor location after it finishes

writing a string. To do this, make the following call after you initialize the window values:

```
sw_plcsr(ON, &wn);
```

Direct Placement of the Screen Cursor

At any time, whether or not automatic cursor placement is implemented, you can place the screen cursor directly at the virtual cursor location by:

```
pl_csr(&wn);
```

You can also move the location of the virtual cursor and place the screen cursor at that location with the function:

```
mv_csr(row, column, &wn)
int row;
int column;
WINDOW wn;
```

To move the physical cursor to a specified location on the screen (as opposed to a location within a window), you can use **mv_csr()** in a pre-defined full screen window, **wn0**. See "Using Window Functions on the Full Screen," later in this chapter.

Reading the Location of the Screen Cursor

You can read and save the location of the screen cursor with:

```
rd_csr(&row, &col, page)
int row;
int col;
int page;
```

Warning: you must use **pointers** to the row and column variables. The page argument applies only to the Color Graphics Adapter or the Enhanced Graphics Adapter.

Hiding and Restoring the Screen Cursor

You can hide the cursor so that it doesn't show on the screen by moving it just below the last row of the screen:

```
mv_csr(v_rwq, 0, &wn0);
```

This call uses the global variable **v_rwq**, which is set to the quantity of rows on the screen. Because the screen rows are measured from a 0 origin, the **v_rwq** row will be one below the screen. The cursor will not show on the screen.

You can hide the cursor and save its location with:

```
csr_hide();
```

Function **csr_hide()** stores the location of the cursor internally, in static variables.

To restore the cursor to the screen at the location saved by **csr_hide()**, use:

```
csr_show();
```

Warning: You cannot use **csr_show()** to restore the cursor if you make two or more calls to **csr_hide()** without an intervening call to **csr_show()**. After the second straight call to **csr_hide()** the stored location of the cursor will be off-screen. A call to **csr_show()** will simply leave the cursor off-screen. If this is a possibility, use **rd_csr()** to store the location and **pl_csr()** or **mv_csr()** to restore the screen cursor.

WRITING TO WINDOWS

All functions that write to the screen are prefaced by **v_**. This is an abbreviated mnemonic for "video". Functions that read from the screen are prefaced by **vo**, which stands for "video out", short for "out from video".

You can use window functions without the windows being set on the screen. As soon as a window is declared and initialized, window functions can refer to it. Note, though, that if a window has borders, writing will cover the **FULL** dimensions of the window, including the border, until its dimensions are set to **INSIDE** (either by **set_wn()** or by **dim_wn()**).

BASIC STRING OUTPUT: v_st()

The "video string" function, **v_st()**, is the basic string output function of **Windows for C**. The format of the call is:

```
v_st(st, &wn);
char *st;
WINDOW wn;
```

Function **v_st()** will write output beginning at the location of the virtual cursor. With default window values, output will have attribute LNORMAL and word wrap will be implemented. You can change these defaults, as described in an earlier section of this chapter.

Function **v_st()** will stop writing when it fills the window. If scrolling is enabled and you call **v_st()** when the window is already full, the window will be scrolled up one line and **v_st()** will write on the bottom line of the window. Only one row will be written. If the string is longer than the window width, it will stop when the window becomes full.

If you know the strings you are writing are all less than a window width, you can call **v_st()** repeatedly, and it will always write the string, scrolling up the window with each call after it becomes full. If you are not sure of the string lengths, and you want to be certain that a string is entirely written, use **v_fst()** (see below).

FULL-STRING OUTPUT FUNCTION: v_fst()

If you want to ensure that a string is written entirely, use "video full-string", **v_fst()**. It is identical to **v_st()**, except that it will ensure that the end of a string is written to a window, automatically scrolling up the text in the window when this is required.

For more information on controlling scrolling for string output in direct display windows, see the reference page for **v_st()** in Appendix 2.

WRITING A STRING AT A SPECIFIED LOCATION: v_plst()

Moving the virtual cursor to a specific location and then writing a string can be done with one call to the "video place-string" function:

```
v_plst(row, column, string, &wn);
int row;
int column;
char *string;
WINDOW wn;
```

This function places the virtual cursor at **row** and **column** and calls **v_st**.

CENTERING, LEFT-JUSTIFYING AND RIGHT-JUSTIFYING TEXT

You can automatically center output within a window row by using **CENTER_TXT** as the column value in **v_plst()**, e.g.

```
v_plst(3, CENTER_TXT, "Hello, world", &wn);
```

will center the text string in the fourth row (row 0 is the first row).

Similarly, using **RIGHT_TXT** will place the last character of the string in the right-most position of the window, and **LEFT_TXT** will place the first non-whitespace character in the left-most position of the window.

FORMATTED STRING OUTPUT: v_printf()

The equivalent of **printf()** is provided by

```
v_printf(&wn, control, arg1, arg2, ...)
```

See your compiler reference manual or K&R, Chapter 7, for the interpretation of **control** and the **arg's** parameters in **v_printf()**. Function v_printf() calls **v_st()** and, thus, has the same options as **v_st()**.

Warning: The number of arguments in **v_printf()** is limited to 20 **ints**, 10 **longs** and 5 **doubles** or **floats**.

Note: This function calls **sprintf()**, which is quite large. If code size is a concern, you can save thousands of bytes by avoiding **v_printf()** (and **sprintf()** and its cousins, **scanf()**, **printf()**, etc.). Use **v_st()** as the standard output function and do explicit conversions from numbers to strings when required.

WRITING CHARACTERS

Single Characters

The window equivalent of **putchar()** is:

```
v_ch(character, &wn)
char character;
WINDOW wn;
```

This writes a character at the virtual cursor position. The default setting is for advance of the virtual cursor, but this can be changed with **sw_csadv()**.

Rows and Columns of Characters

Functions are provided for writing rows and columns of identical characters:

```
v_rw(character, quantity, &wn)
char character;
int quantity;
WINDOW wn;

v_co(character, quantity, &wn)
char character;
int quantity;
WINDOW wn;
```

Both of these functions advance the virtual cursor and will continue writing until the quantity of characters specified is written or the window is full.

Filling a Window with a Specified Character

Setting the virtual cursor to 0, 0 and calling **v_rw()** or **v_co()** with a quantity larger than the number of screen positions is an easy way to fill the window with a desired character:

```
mv_cs(0, 0, &wn);
v_rw('X', 4000, &wn);
```

SOUNDING THE BELL (BEEP)

The speaker can be made to beep by

```
bell();
```

so named for historical reasons.

CHANGING THE ATTRIBUTE OF A CHARACTER

You can change the attribute of a character at the location of the virtual cursor with:

```
v_att(attribute, &wn)
char attribute;
WINDOW wn;
```

The default setting is for advance of the virtual cursor, but this can be changed with **sw_csadv()**.

Library functions are provided for changing specified parts of the screen in one call. See Chapter 8.

SCROLLING DIRECT DISPLAY WINDOWS

You can scroll a direct display window upward or downward with:

```
mv_rws(nlines, dir, &wn);
int nlines;                    number of lines to scroll
char dir;                      direction
WINDOW wn;                     window structure
```

EXECUTING SUBROUTINES WHILE WAITING FOR KEYSTROKES

The keyboard-read function, **ki()**, allows you to install a user-developed function that will be executed whenever the program is waiting for input from the keyboard. When **ki()** is called, the program will execute an endless loop that consists of checking for a keystroke and, if not found, executing the user-developed function. Exit from the loop will occur when a keystroke is found.

The following code fragment shows how to install the keyboard loop function:

```
int loop_func();

s_keyloop(loop_func);
```

Whenever you call ki() and there are no keystrokes waiting in the keyboard buffer, the loop function will be executed.

To remove the keyboard loop function, give the following call

```
s_keyloop(NULLFP);
```

For an example on how to install and use a keyboard loop function, refer to the file **LOOP.C** on the system diskette.

Warning: The keyboard loop function is not available in the UNIX/XENIX version of **Windows for C.**

CREATING A PAUSE IN A PROGRAM

You can create a user-controlled pause in a program by using **ki()**, as illustrated in the following code fragment.

```
v_plst(0, 0, "Type any key to proceed: ", &wn);
pl_csr(&wn);
ki();
```

The message is written on the next to first line of the window, the screen cursor is placed after the message, and a call is made to read the keyboard. The program will halt until a key is pressed.

CHECKING THE KEYBOARD BUFFER: ki_chk()

Function **ki_chk()** checks the keyboard buffer. If a keycode is in the buffer, it returns its value, but does not remove the code from the buffer. If no code is in the buffer, **ki_chk()** returns a 0. It does not wait for a keystroke.

You can use this function to see if a key has been entered without halting operations until a key is entered. In some applications you may wish to check for a keystroke periodically and to continue processing until one needs to be serviced.

CLEARING THE SCREEN AND WINDOWS

CLEARING THE SCREEN

To clear the screen, use:

```
cls();
```

CLEARING WINDOWS

To clear a window (but leave the border intact), use:

```
cl_wn(&wn);
```

The window will be cleared with spaces of the window attribute value.

REMOVING A WINDOW FROM THE SCREEN

To remove a window (including borders) from the screen, use

```
unset_wn(&wn);
```

For pop-up windows, this function will restore the underlying image; for overwrite windows, it will clear the area covered by the window to the background. See the discussion earlier in this chapter, under "Displaying and Removing Windows", for more details.

CONTROLLING THE COLOR OF THE SCREEN BACKGROUND

The attribute value of the screen background is stored in the background attribute portion of the global variable **cl_att**. Blank **spaces** of attribute **cl_att** are written by **cls()** to clear the entire screen and by **unset_wn()** to clear the area covered by an overwrite window.

The value of **cl_att** is initialized to **NORMAL** for physical attributes and to **LDOS** for logical attributes. The pre-defined value of **LDOS** is **white** on **black** for both monochrome and color modes. As initialized, the system will clear the screen to a black background.

You can change the background to which the screen is cleared by directly assigning a new value to **cl_att**. After changing the value of **cl_att**, call **cls()** to clear the screen with the new background. For example:

```
cl_att = LNORMAL;
cls();
```

will clear the screen to the background portion of **LNORMAL** (**blue** on color screens and black on monochrome screens).

If you are operating with logical attributes, you can only set the screen background to background attributes contained in one of the defined logical attributes. If you want a background attributes not included in the pre-defined logical attributes, you can add a new logical attribute with the desired background color; see Chapter 5.

If you are operating with physical attributes, a special function is provided to assist you in setting color backgrounds; see Chapter 5.

The direction must be specified as either **UP** or **DOWN**. For example

```
mv_rws(3, UP, &wn);
```

will move all lines in the window up 3 rows. The top 3 rows will be lost, and the bottom 3 rows will be cleared. The virtual cursor will be moved by the number of rows scrolled (but will always stay within the window).

Note: This scrolling function works on the contents of a window displayed on the screen. If you want to scroll through text that is larger than a window, you should use memory-file functions. These functions allow horizontal and vertical scrolling. See Chapter 5.

Hint: If you want to scroll less than the entire window, define a separate window to cover the area that you desire to scroll, and then use mv_rws() on the separate window.

READING THE CONTENTS OF WINDOWS

READING A CHARACTER

You can read the ASCII value of the character located at the virtual cursor with the "video-out character" function, as in the following code fragment:

```
char character;

character = vo_ch(&wn);
```

The virtual cursor is not advanced.

READING AN ATTRIBUTE

You can read the attribute associated with character located at the virtual cursor with the "video-out attribute" function, as in the following code fragment:

```
char attribute;

attribute = vo_att(&wn);
```

The virtual cursor is not advanced.

READING THE CHARACTER CONTENTS OF PARTS OF THE WINDOW

You can transfer the character contents of specified portions of a window to or from a character string with the function, **v_mova()**. You can transfer a character, a row, a column, the entire window, and other portions of the window. This function is described fully in Chapter 8.

READING THE CHARACTER-ATTRIBUTE CONTENTS OF PARTS OF THE WINDOW

You can transfer the character-attribute pairs of specified portions of a window to or from a special "video string" with the function, **v_mov()**. You can transfer a character, a row, a column, the entire window, and other portions of the window. This function is described fully in Chapter 8.

READING THE KEYBOARD

KEYCODE CONVENTIONS

The IBM PC has two sets of keycodes: 1) the set of standard keycodes, which are numbered from 0 through 255, include all of the standard ASCII character set and a special character set; and 2) a set of "extended codes." The extended codes range from 3 to 132. The extended codes thus overlap the standard codes. The IBM BIOS differentiates between them by setting an extra switch when an extended code is being returned.

The keyboard functions of **Windows for C** return both standard and extended codes in the same variable. To differentiate between them, extended codes are returned as negative values and standard codes as positive values.

Unfortunately, when we **#defined** keycode values for the keys, we used IBM's convention of positive values for the extended codes, instead of the convention of negative values used by our own keyboard functions. This was not a good design decision. Because of this, when you are looking for a particular keystroke, you must know if the key returns an extended code. For extended codes, you must compare the **negative** of the keycode with the value returned by the keyboard function (because extended codes are returned as negative values). This creates an additional burden on you for which we apologize.

Fortunately, keys returning extended codes fall, for the most part, in a few groups: function keys, cursor pad keys and <ctrl-cursor-keys>, and <alt-keys> (a standard key pressed simultaneously with the <alt> key). The **#defines** for the extended codes for the function and cursor pad keys are listed separately in **computer.h**.

Another solution is for you to change the **#defines** for extended codes to negative values. Although we do not want to change the key definitions and create incompatibilities for existing programs of our customers, you are free to change them to negative values. All library functions will work without any problem. You will, however, need to modify our demonstration programs by changing the key values for keys returning extended codes. If you are also using **Windows for Data,** you will need to make similar changes to **wfd_glob.h**.

THE READ-KEYSTROKE FUNCTION: ki()

The **Windows for C** basic function for reading the keyboard is:

```
ki();
```

When a keystroke is entered by a user, the IBM operating system places it its code in a **keyboard** buffer until it is removed. Function **ki()** will remove a code and return its value each time it is called. If no code is in the buffer when it first looks, **ki()** will wait until one is entered.

Extended codes (see above) are returned as negative values.

As noted above, when using **ki()** to check for an extended code, the return value must be compared with the negative of the #defined key value. For example, the following code line obtains a keystroke and checks to see if it is function key F-10:

```
if(ki() == -K_F10)
```

USING WINDOW FUNCTIONS ON THE FULL SCREEN

At times you may wish to use **Window** functions on the full screen. To simplify this, a window equal to the full screen, **wn0** (wn zero), is pre-defined and globally available. The full-screen window has no border. It has the attribute **LDOS** (or **NORMAL** for physical attributes). This window is useful in a number of ways.

SAVING, CLEARING, AND RESTORING THE ORIGINAL DOS SCREEN

Use **wn0** to save the DOS screen that existed prior to entering your program if you want to restore it at the end of your program. At the beginning of the program, before you change the screen, include the statements:

```
sw_popup(ON, &wn0);
set_wn(&wn0);
```

This will save the image of the full-screen, because **wn0** has been made into a pop-up window. The screen will be cleared to **black** (unless you have changed the definition of **LDOS**).

Note: Saving a full 25 by 80 screen will use 4000 bytes of memory.

At the end of the program, restore the DOS screen by:

```
unset_wn(&wn0);
```

You will also want to save and restore the DOS screen cursor (see below).

MOVING, SAVING, AND RESTORING THE SCREEN CURSOR

Moving the Screen Cursor

You can move the screen cursor to any row and column on the full screen by using the **mv_csr()** function on the full-screen window:

```
mv_csr(row, column, &wn0);
```

Use this to place the cursor on the last line (or the first line) at the end of a program.

Programming hint: If you place the screen cursor on row 23 before exiting, the cursor will move to the last line of the screen when you return to DOS, but the screen will not scroll.

Saving and Restoring the Screen Cursor

You can save the location of the cursor at any point in a program with:

```
rd_csr(&row, &column, page)
int row;
int column;
int page;                          page number of graphics card
```

The **page** argument will zero, **unless you are using** additional memory pages on the graphics adapter board. **Note that the arguments for row and column are pointers.**

To restore the cursor to the saved position at a later point in the program, use:

```
mv_csr(row, column, &wn0);
```

WRITING TO THE FULL SCREEN

At times, especially during program development, you may wish to write information to the full-screen. Using the pre-defined window **wn0** avoids the need to first declare and define a window before using the **Window** output functions. For example,

```
pl_csr(5, 0, &wn0);
v_st("This writes its output on row 5, starting a column 0.", &wn0);
```

Note: Output will be written with attribute **LDOS** (or **NORMAL**), unless you change the attribute of **wn0**.

ANOTHER VERSION OF "HELLO, WORLD"

Using the features discussed above, we can write another version of "Hello, world:"

```
#include <bios.h>
#include <window.h>

main()
{
     WINDOW wn;
     defs_wn(&wn, 10, 30, 8, 25, BDR_DLNP);
     sw_popup(ON, &wn);                    /*make a popup window        */
     sw_name("First Window", &wn);         /*give it a name             */
     csr_hide();                           /*hide the cursor            */
     set_wn(&wn);                          /*pop wn up on screen        */
     v_plst(3, CENTER_TXT, "Hello, world\n", &wn);    /*center string   */
     sw_att(LHIGHLITE, &wn);               /*change output attribute    */
     v_plst(7, 0, "Press any key to exit.", &wn);
     ki();                                 /*wait for a keystroke       */
     unset_wn(&wn);                        /*restore the original screen*/
     csr_show();                           /*restore the cursor         */
}
```

PRACTICAL EXAMPLES OF WINDOWS

AN ERROR MESSAGE WINDOW

The following code creates a one-row pop-up window for error messages on the last row of the screen. The "window" has no border and has attribute **LERROR**.

```
WINDOW wn_err;

defs_wn(&wn_err, v_rwq - 1, 0, 1, v_coq, BDR_0P);
sw_popup(ON, &wn_err);
sw_att(LERROR, &wn_err);
```

The variable **v_rwq** is a system global variable that contains the number of rows on the screen (which will generally be 25 on the IBM PC, but can be 43 with the enhanced graphics adapter and may have other values for other display boards). Variable v_coq similarly contains the number of columns on the screen.

To write an error message:

```
set_wn(&wn_err);
v_st("Undefined key", &wn_err);
```

To remove the message:

```
unset_wn(&wn_err);
```

ESTABLISHING A STATUS LINE

The following example uses a one line window to establish a status line on the next to last line of the screen (leaving the last line for an error window). The status line is not a popup but will remain permanently. Automatic cursor advance and automatic clear-to-the-end-of-the-row are turned off on the status line. This gives better control over output on the line and will prevent status information near the end of the line from being erased when the positions to the left are updated.

The top part of the screen, above the status line is defined as another window. General output will go to the main window. Automatic screen cursor placement is turned on in the main window.

```
WINDOW wn_stat, wn_main;

defs_wn(&wn_stat, v_rwq - 2, 0, 1, v_coq, BDR_0P);
sw_csadv(OFF, &wn_stat);
sw_cleor(OFF, &wn_stat);
sw_att(LREVERSE, &wn_stat);

defs_wn(&wn_main, 0, 0, v_rwq - 2, v_coq, BDR_0P);
sw_plcsr(ON, &wn_main);
```

Variables **v_rwq** and **v_coq** are system global variables that contain the number of rows and columns on the screen. They are set by **init_wfc()** and may change if vid_mode() is used.

When you wish to write to the main part of the screen, reference **wn_main** in the output functions. Output will not be permitted to go to the status line. Conversely, when you want to write to the status line, reference **wn_stat**.

For example, if you are keeping the value of the column position of the screen cursor in the main window at position 70 on the status line, you could update this position with:

```
pl_csr(0, 70, &wn_stat);
v_printf(&wn_stat, "%d", wn_main.c);
```

The current location of the cursor is in **wn_main.c**.

This page intentionally left blank.

Chapter 4

TUTORIAL ON WINDOWS FOR C

CONTENTS

ANNOTATED CODE LISTING

This page intentionally left blank.

TUTORIAL ON WINDOWS FOR C

This chapter consists of a commented program listing that uses many of the functions introduced in Chapter 3. It shows how windows are initialized, established on the screen, cleared, written to, scrolled, and removed from the screen.

This program calls a function, **rd_lines()**, that draws on material introduced in later chapters. Code for this function is listed at the end. You will be referred back to **rd_lines()** at the appropriate time.

Source code for this listing is on the system diskette in file **tutor.c**

TUTORIAL ON WINDOWS FOR C

```
/* tutor.c  -- tutorial program for Windows for C

  ************* (C) Copyright Vermont Creative Software 1985 *****************

A simple program that establishes two windows, reads lines typed by the user in
one window, and writes the lines in the other window.

This program illustrates how windows are initialized, established on the screen,
cleared, written to, scrolled, and removed from the screen.  It shows how window
functions can be used to simplify constructing line editors.

*/

#include <bios.h>
#include <window.h>
#define MAX_CHAR 73                     /*max characters in edit line        */


WINDOW wn1, wn2;                        /*declare two windows                */

/*-------------------------------------------------------------------------*/
/*  The following strings provide the text that will be put into wn2       */
/*  by v_st().                                                             */
/*-------------------------------------------------------------------------*/

char *st[] =
    {"\nTo place lines of text into the top window on the screen, ",
    "type the lines here, pressing Enter (Carriage Return) after ",
    "each line:\n"
    };

char *st1 = "                       To exit, press 'Escape'.";

main()
{
    int i, csr_row, csr_col;
    char line[MAX_CHAR + 1];            /*array for user input of text       */

/*-------------------------------------------------------------------------*/
/*  First save the cursor location by calling rd_csr(); then               */
/*  Save the screen by using the predefined full-screen window wn0;        */
/*  setting wn0 will clear the screen to black, because wn0.att = LDOS.    */
/*-------------------------------------------------------------------------*/

    init_wfc();                         /*always initialize WFC first        */
    rd_csr(&csr_row, &csr_col, 0);      /*store cursor location              */
    sw_popup(ON, &wn0);                 /*make a popup to save screen        */
    set_wn(&wn0);                       /*saves screen and then clears it    */
```

```
/*-------------------------------------------------------------------------*/
/* Initialize the windows.  Text entered in window 2 will be copied to     */
/* window 1 for display.  When window 1 is full, text will be automatically */
/* scrolled upward to make room for additional lines of text.              */
/*                                                                         */
/* Window 1 will have attribute LNORMAL and border attribute LRED.  Margins */
/* will be 3 on each side.  The border is a double line.                   */
/* The window goes from row 3 through row 10, column 20, through column 60. */
/*                                                                         */
/* Window 2 will have attributes of LNORMAL for text and LNORMAL for border.*/
/* Margins are 2, 2.  The window has a single line border                  */
/* The window goes from row 14 through row 20, column 0, through column 79. */
/*                                                                         */
/* Function def_wn(), which allows explicit setting of margins, is used for */
/* initialization.                                                         */
/*-------------------------------------------------------------------------*/

    def_wn(&wn1, 3, 10, 20, 60, 3, 3, BDR_DLNP);
    sw_bdratt(LRED, &wn1);                  /*change border att from default*/
    def_wn(&wn2, 14, 20, 0, 79, 2, 2, BDR_LNP);

/*-------------------------------------------------------------------------*/
/*  Place the windows on the screen with calls to set_wn.  A pointer to    */
/*  the WINDOW struct must be passed as the calling parameter.             */
/*                                                                         */
/*  set_wn returns NULL if there is an inconsistency in the initial values */
/*  of wn.                                                                 */
/*                                                                         */
/*  The error message can be written in a window even though it is not set. */
/*  Setting of a window clears it and sets border and margins, but none of */
/*  these are essential to the writing of the error message                */
/*-------------------------------------------------------------------------*/

    if(set_wn(&wn1) == 0)
    {
        v_st("wn1 definitions inconsistent",&wn2);
        exit(1);                        /*exit to DOS with error code set    */
    }
    if(set_wn(&wn2) == 0)
    {
        v_st("wn2 definitions inconsistent",&wn2);
        exit(1);                        /*exit to DOS with error code set    */
    }
```

```
/*-------------------------------------------------------------------------*/
/*  Repeated calls to v_st() are made to write the st[] array to wn2.      */
/*                                                                         */
/*  The initial location of of cs, the virtual cursor is at 0,0; v_st()    */
/*  automatically updates the location of cs so it points to the next      */
/*  space after the last character written.                                */
/*                                                                         */
/*  After st[] is written, the attribute of wn2 is changed to LERROR and   */
/*  wn.r is advanced by two, moving cs down two rows.                      */
/*                                                                         */
/*  v_st() is called to write st1, the exit instruction; the attribute is  */
/*  changed to LFIELDA,  the location of cs is moved to the location       */
/*  where the user is to type in text, and the screen cursor is placed at  */
/*  cs by a call to pl_csr().                                              */
/*-------------------------------------------------------------------------*/
    for(i = 0; i < 3; i++)
        v_st(st[i], &wn2);

    wn2.att = LERROR;                   /*use LERROR to make prominent      */
    wn2.r += 2;
    v_st(st1, &wn2);

    wn2.att = LFIELDA;
    wn2.r -= 2;
    wn2.c = 0;
    pl_csr(&wn2);

/*-------------------------------------------------------------------------*/
/*  Clear the edit row to LFIELDA and restore the virtual cursor to the    */
/*  beginning of the row.  Note that the screen cursor is not moved by     */
/*  the call to v_rw().                                                    */
/*-------------------------------------------------------------------------*/

    v_rw(' ', MAX_CHAR, &wn2);
    wn2.c = 0;

/*-------------------------------------------------------------------------*/
/*  The following code sets up a while() loop.  The loop is broken         */
/*  when the user satisfies the exit condition:                            */
/*                                                                         */
/*  User text is read into line[], using rd_line(), a function defined     */
/*  following main.   Text is read until <Escape> is pressed on an empty   */
/*  line.                                                                  */
/*                                                                         */
/*  Rd_line() returns -1 when <Escape> is pressed; otherwise the number    */
/*  of characters in the returned string (excluding the terminal null)     */
/*  The loop is exited upon the -1  return.                                */
/*-------------------------------------------------------------------------*/
```

```
    while(rd_line(line, MAX_CHAR, &wn2) != -1)   /*set up loop              */
    {

/*-------------------------------------------------------------------------*/
/*  The contents of line[] are written to wn1 by a call to v_fst().        */
/*  For strings longer than the width of wn1, v_fst() automatically performs*/
/*  word wrap.                                                             */
/*                                                                         */
/*  When v_fst() fills the window without finishing writing the string, it */
/*  scrolls the window to make room for the remainder of the string.       */
/*-------------------------------------------------------------------------*/

        v_fst(line, &wn1);

/*-------------------------------------------------------------------------*/
/*  After the line has been transferred to wn1, the user text in wn2 is    */
/*  cleared by writing MAX_CHAR  "spaces" using v_rw().                    */
/*  The virtual cursor and screen cursor are returned to the initial       */
/*  position.                                                              */
/*-------------------------------------------------------------------------*/

        wn2.c = Ø;
        v_rw(' ', MAX_CHAR, &wn2);
        wn2.c = Ø;
        pl_csr(&wn2);
    }

/*-------------------------------------------------------------------------*/
/*  When this point is reached, the user has requested  exit.              */
/*  The original screen is restored by popping down window wnØ.  The cursor */
/*  is restored to its original location by calling mv_csr() in wnØ.       */
/*-------------------------------------------------------------------------*/

    unset_wn(&wnØ);
    mv_csr(csr_row, csr_col, &wnØ);
    return;
}
```

```
/*-------------------------------------------------------------------------*/
/*                      The rd_line() function                             */
/*                                                                         */
/*  The rd_line() function calls ki() to read keystrokes and v_ch() to write*/
/*  the corresponding ASCII character to the window. Backspace and left and */
/*  right cursor keys are implemented; but not insert and delete.          */
/*                                                                         */
/*  ki() returns ASCII codes or the IBM "extended codes."  The extended    */
/*  codes are returned as negative values.                                 */
/*                                                                         */
/*  Keystrokes are written to the window until <Enter> is pressed,         */
/*  max_q characters have been written to the window,                       */
/*  <Escape> is pressed.                                                   */
/*                                                                         */
/*  When <Escape> is pressed, -1 is returned immediately;                  */
/*  otherwise the contents of the window row are copied to the passed      */
/*  string.  Trailing blanks are stripped; if <Enter> was pressed, a       */
/*  newline is inserted prior to the terminal null.                        */
/*                                                                         */
/*  The number of characters in the string (excluding the NULL terminator) */
/*  is returned.                                                           */
/*                                                                         */
/*  The passed string must have at least max_q + 1 spaces in it            */
/*-------------------------------------------------------------------------*/


rd_line(stp, q_max, wnp)
char *stp;
int q_max;
WINDOWPTR wnp;                          /*typedef WINDOWPTR is in bios.h    */
{
    int c;
    int kenter = FALSE;                 /*set to TRUE if user pressed <ENTER> */
    int done = FALSE;                   /*set to TRUE when line is full or  */
                                        /*user pressed <ENTER>              */
    int count;
    int cmax;                           /*maximum value of wn.c             */

    cmax = q_max -1;                    /*cs has zero origin; so q_max-1    */
    if(q_max < 1) return(0);            /*need q = 1 to get 1 char          */
    while(! done)                       /*set up loop                       */
    {
        switch(c = ki())
        {
            case K_ENTER:               /*user pressed <ENTER>              */
                kenter = 1;
                done = 1;
                break;

            case K_ESC:                 /*user wants to exit                */
                return(-1);
```

```
            case K_BACK:                /*handle backspace                    */
                if(wnp->c > 0)          /*if not at left boundary             */
                {
                    wnp->c--;           /*go back one position                */
                    v_ch(' ', wnp);     /*print "space"                       */
                    wnp->c--;           /*go back one position                */
                    pl_csr(wnp);        /*place screen cursor                 */
                }
                else                    /*can't backspace beyond left boundary*/
                    bell();
                break;

            case -K_LEFT:               /*handle left cursor                  */
                if(wnp->c > 0)          /*if not at left boundary             */
                {
                    wnp->c--;           /*move cursor 1 column left           */
                    pl_csr(wnp);        /*place screen cursor                 */
                }
                else                    /*can't move cursor left              */
                    bell();
                break;

            case -K_RIGHT:              /*handle right cursor                 */
                if(wnp->c < cmax)       /*if not at right boundary            */
                {
                    wnp->c++;           /*move cursor 1 column right          */
                    pl_csr(wnp);        /*place screen cursor                 */
                }
                else                    /*can't move cursor right             */
                    bell();
                break;

            default:                    /*handle all other possibilities      */
                if(c > 0)               /*check for printable character       */
                {
                    if(wnp->c == cmax)/*Done. write last char & goto end      */
                    {                   /*check cs limit first, because wn.c  */
                        v_ch(c, wnp); /*will go to zero if at edge of wind.  */
                        done = 1;
                    }
                    else
                    {
                        v_ch(c, wnp); /*else write character and continue     */
                        pl_csr(wnp);
                    }
                }
                else                    /* extended code                      */
                    bell();             /* let user know it's illegal         */
                break;
        }
    }
```

```
    v_mova(stp, wnp, ROW, OUT);        /*transfer window row to string       */
    strip_wh(stp);                     /*strip white spaces from end of strin*/
    count = strlen(stp);
    if(kenter)                         /*need to append newline              */
    {
        stp += count++;                    /*go to null, increment count     */
        *stp++ = '\n';                     /*append newline                  */
        *stp = '\Ø';                       /*append NULL terminator          */
    }
    return(count);                     /*return number of char in string     */
}
```

Chapter 5

CONTROLLING COLOR: LOGICAL AND PHYSICAL ATTRIBUTES

CONTENTS

THE LOGICAL ATTRIBUTE SYSTEM
- Why Logical Attributes?
- Logical Attributes
  - System Logical Attributes and Window-Specific Logical Attributes
- Changing the Physical Attributes of Logical Attributes
- Adding New Logical Attributes
  - Step 1: Define a Logical Attribute Name
  - Step 2: Change the Number of Logical Attributes
  - Step 3: Add the New Definition to the Logical Attribute Table
- Adding New Columns of Logical Attributes
  - Step 1: Change the Physical-Attribute-Quantity Parameter
  - Step 2: Add a New Column of Physical Attributes
- Initializing the Logical Attribute Array
  - Coding Example
- Constructing and Using Window-Specific Logical Attribute Tables
  - Setting a Window to Use a Specified Logical Array: sw_latt()

USING PHYSICAL ATTRIBUTES
- Setting the System To Use Physical Attributes
  - Physical and Logical Attributes Don't Mix
- Monochrome Attributes
- Avoid Underline in System Logical Attributes
- Managing Window Colors
  - Differences Between the Enhanced Graphics and Color/Graphics Adapters
  - Foreground and Background Colors
  - Changing the Color of Window Borders
  - Removing Color Windows: Maintaining the Background
- Selecting the Screen Border Color
- "Hello, World" with Physical Color Attributes

DETERMINING AND CHANGING VIDEO MODES
- Using Graphics in Programs that Use Windows for C
- Coding Example

This page intentionally left blank.

CONTROLLING COLOR: LOGICAL AND PHYSICAL ATTRIBUTES

Windows for C provides complete, easy control of the color capabilities provided by the IBM Color/Graphics Adapter. Background, foreground, and border colors can be specified for each window. Video modes can be changed within programs, and the color of the screen border specified.

This chapter explains how to control the color and monochrome **attributes** of screen displays. Attribute is the technical term for the color or monochrome appearance of characters written on the screen.

Windows for C optionally implements a system of **logical attributes.** Logical attributes allow you to write one program that utilizes the full color capabilities of color monitors and is fully legible on monochrome monitors. You can modify the colors associated with a given logical attribute and add new logical attribute definitions.

Logical attributes are strongly recommended if you are developing software that may run on either monochrome or color displays.

For software intended to run on one type of display, you can use **physical attributes. Windows for C** provides a number of functions to facilitate controlling color via physical attributes.

The first part of this chapter explains the logical attribute system. The latter part of the chapter describes the functions provided for handling physical attributes and the function provided for changing video modes.

This chapter assumes that you have read the introductory material on attributes in "Controlling the Appearance of Output: Attributes," in Chapter 3.

THE LOGICAL ATTRIBUTE SYSTEM

WHY LOGICAL ATTRIBUTES?

Windows for C uses the attribute system of the IBM PC for controlling display attributes. Each character displayed on the screen has associated with it an **attribute byte** that determines its appearance. The attribute-byte values are termed **physical attributes.**

Use of physical attributes in programs creates problems. Not all color attributes will display properly on black and white monitors. If you use physical attributes in your programs, you must either restrict the color choices or write separate code for color and black and white video modes.

If you are developing software to run on both monochrome and color displays, we strongly recommend that you use the logical attribute system of **Windows for C.** In addition to providing color and monochrome display compatibility, logical attributes make it much easier to change color choices in a program.

LOGICAL ATTRIBUTES

Logical attributes allow you to write one program that utilizes the full color capabilities of color monitors and is fully legible on monochrome monitors.

Each logical attribute has a physical color attribute and a physical monochrome attribute defined for it. When a computer is in a color mode, the display functions will use the color attribute. When in a monochrome mode (black and white) mode, the monochrome attribute will be used.

Logical attributes are generally categorized by function or purpose. For example, some of the pre-defined logical attributes of **Windows for C** are: **LNORMAL, LHIGHLITE, LERROR, LMESSAGE.** You can also have **logical color attributes**, which associate a given color attribute with a chosen monochrome attribute. For example, **LRED and LGREEN** are pre-defined logical attributes. **LRED** provides a red foreground color against a blue background in color modes and white foreground against black background in monochrome modes.

Table 3.2 lists the pre-defined logical attributes, together with associated monochrome and color attributes.

You can change the physical attributes associated with the pre-defined logical attributes and you can define as many additional logical attributes as you wish.

System Logical Attributes and Window-Specific Logical Attributes

When logical attributes are being used, there will always be **system logical attributes.** These are pre-defined by the system (although you can modify the definitions) and can be used to write in any window. For most applications, the system logical attributes will suffice.

For some applications, primarily in developing software tools, you may wish to have logical attributes that are separate from those of the system. **Windows for C** allows you to define one or more additional sets of logical attributes and assign a specific set to be used with each window. When you do not assign a set to a window, the system logical attributes will be used by output functions.

System logical attributes will be described first.

CHANGING THE PHYSICAL ATTRIBUTES OF LOGICAL ATTRIBUTES

The physical attributes that are associated with each system logical attribute are specified in a **logical attribute table, datt_tbl[][]** contained in file **att_glob.h.** This table has a row for each pre-defined logical attribute, and each row has two columns. The first column contains the physical attribute to be used in monochrome modes and the second the one to be used for color modes.

The output functions will use the Column 0 (first column) entry for IBM video modes 0, 2, and 7 (the black and white and monochrome/adapter modes) and Column 1 for modes 1 and 3 (color modes).

To change the physical attributes associated with a logical attribute, edit file **att_glob.h** and change the entries in the logical attribute table **datt_tbl[][]**. For example, the row in the table for **LNORMAL** now reads:

```
{NORMAL, c_att(WHITE, BLUE)},                    /*LNORMAL  */
```

NORMAL, WHITE, and **BLUE** are #defined values of physical attributes (see Table 5.1 for a complete listing of #defined physical attributes). Function **c_att()** returns the value of a physical attribute byte. The first argument in **c_att()** specifies the foreground color and the second argument specifies the background color.

To change **LNORMAL** so that **GREEN** on **RED** would be displayed in color modes, make the **LNORMAL** line read:

```
{NORMAL, c_att(GREEN, RED)},                    /*LNORMAL   */
```

You can change the colors or monochrome attribute for any logical attribute in similar fashion. See the later section in this chapter for more information on how to specify physical attributes.

ADDING NEW LOGICAL ATTRIBUTES

As supplied, the system logical attribute table defines 19 logical attributes and reserves space for 13 more. You can add as many logical attributes as you wish onto the end of this table.

Warning: The first 32 positions in the logical attribute table are reserved for system use. If you use any of these positions to define new logical attributes, you may conflict with definitions used within the system at some point in time. At present, only the first two logical attributes are used internally within **Windows for C**, and only the first 11 are used by **Windows for Data**. This may change at a later time.

To add a logical attribute, you need to take 3 steps: 1) #define the name for the new attribute, 2) change the parameter for the number of logical attributes, and 3) add the logical attribute to the logical attribute table.

We will go through these steps for an example: creating the logical attribute **LSTATUS**.

Step 1: Define a Logical Attribute Name

Logical attribute names are #defined in **def_att.h**. Edit this file and go to the last entry. The last several entries currently read:

```
#define LAVR11      30          /*reserved logical attribute   */
#define LAVR12      31          /*reserved logical attribute   */
```

Add the new entry to the end and give it the next number in sequence:

```
#define LAVR11      30          /*reserved logical attribute   */
#define LAVR12      31          /*reserved logical attribute   */
#define LSTATUS     32          /*logical status line attribute*/
```

Step 2: Change the Number of Logical Attributes

At the top of **def_att.h**, the number of logical attributes is #defined as **LATTQ**. This originally is set to 32. There are now 33 logical attributes (note that the attributes are numbered starting with 0):

```
#define LATTQ 33                    /*number of logical attributes*/
```

Step 3: Add the New Definition to the Logical Attribute Table

Edit file **att_glob.h** and go to the bottom of **datt_tbl[][]**, which now reads:

```
    {0, 0}, {0, 0}, {0, 0}                          /*RESERVED */
};
```

and add the physical attributes you want to use for **LSTATUS.** In the example, we use **REVERSE** for monochrome and **BLUE** on **CYAN** for color modes. The last two lines now read:

```
   {Ø, Ø}, {Ø, Ø}, {Ø, Ø},                          /*RESERVED */
   {REVERSE, c_att(BLUE, CYAN)}                     /*LSTATUS  */
};
```

Note: A comma has been appended to the previous last row. The new last row is not terminated by a comma.

ADDING NEW COLUMNS OF LOGICAL ATTRIBUTES

The system logical attributes, as provided, have only two categories of physical attributes: monochrome modes and color modes. You may wish to make further categories. For example, the IBM Enhanced Graphics Adapter allows high-intensity colors in the background as the default setting. You may wish to make a separate column for this adapter. Or you may wish to support an alternate color card that has more colors than the IBM cards.

There are two steps required to add an additional column of physical attributes to the logical attribute table: 1) change the parameter for the number of physical attributes for each logical attribute, and 2) add the new column to the logical attribute table.

Step 1: Change the Physical-Attribute-Quantity Parameter

The number of physical attributes per logical attribute is #defined in **def_att.h** as **PATTQ.** There will be three when you add a column:

```
#define PATTQ 3          /*number of phys. atts per logical att*/
```

Step 2: Add a New Column of Physical Attributes

Add your choice of attributes to each row in **datt_tbl[][]** in **att_glob.h.** For example, you might want **LMESSAGE** to appear as black on light blue. The line in the attribute table would read:

```
{NORMAL + HIGH_INT, c_att(WHITE + LIGHT, BLUE),  /*MESSAGE  */
      c_att(BLACK, LIGHT + BLUE)},               /*MESSAGE  */
```

When you define a new column of logical attributes, you need to add code to use this column under the appropriate conditions, as described in the following section.

INITIALIZING THE LOGICAL ATTRIBUTE ARRAY

When the system is initialized by **init_wfc()**, the appropriate column of the logical attribute table is copied into a single-index **logical attribute array, latt[].** The entries in this array specify the physical attributes used by the output functions when given logical attributes are specified.

When you add a new column of logical attributes, you must see that they are copied to **latt[]** under the appropriate circumstances. Add the code to do this in a subroutine **u_init()**, which will be automatically called at the end of the initialization routine.

Coding Example

Listing 5.1 provides an example of how **u_init()** can be written to initialize the system logical attribute array. It assumes that a third column (column 2) of physical attributes has been defined to be used when the IBM Enhanced Graphics Adapter (EGA) is being used in a color mode. The added column has been reflected in the value of **PATTQ**, which has been changed to 3 (see above).

When the EGA is present and in a color mode (1 or 3), **u_init()** uses **s_latt()** to copy column number 2 (based on a zero origin) of the attribute table to **latt[]**, the system logical attribute array.

To know when to call **s_latt()**, function **u_init()** makes use of system globals **_ibmega**, which is set to 1 when the EGA is present, and **v_mode**, which contains the current video mode number. Function **s_latt()** makes use of system globals **_attrowq**, which contains the number of system logical attributes, and _attcolq, which contains the number of columns in the system logical attribute table. For more information on **s_latt()**, see its reference page in Appendix 2.

CONSTRUCTING AND USING WINDOW-SPECIFIC LOGICAL ATTRIBUTE TABLES

System logical attributes provide a very flexible means of color control and will suffice for most applications. We have provided for logical attributes separate from the system ones primarily for our own development of software tools, but they are available for your use. We expect to use them in a forthcoming data-entry screen generator. They will allow us to maintain our own logical attributes within the editor, while simultaneously allowing the user to specify a set of logical attributes to use with the data forms he or she creates.

The WINDOW structure that controls output to a window contains a member whose purpose is to allow window-specific logical attributes. When this member is not **NULL**, the system assumes it points to a logical attribute array; **Window** output functions will use the contents of this array, rather than the system array, to determine the physical attribute that corresponds to a specified logical attribute.

To use logical attributes other than the system ones, you must:

1) define a new, separate logical attribute table,

2) provide code in **u_init()** (see above) to copy the appropriate column of the attribute table into a separate logical attribute array that you define, and

3) place a pointer to your logical attribute array in each window that is to use this array rather than the system array.

To illustrate the principles, we have constructed a two row by two column logical attribute table, uatt_tbl[][], and written a simple program that writes output to a window with these "user logical attributes". The code is contained in Listing [5.2]. Included in the listing is the code for a version of **u_init()** that will copy the appropriate column of the logical attribute table to the logical attribute array, ulatt[].

Setting a Window to Use a Specified Logical Array: sw_latt()

In the example in Listing [5.2] the set-window-member function **sw_latt()** is used to set the window to use **ulatt[]**. The form of this function is:

```
sw_latt(latt, &wn)
char latt[];
WINDOW wn;
```

Note: When a pointer to an array is required in function arguments or elsewhere, the name of the array, without terminal '[]'s, is used. In **C**, the name of an array is a pointer to the beginning of the array.

USING PHYSICAL ATTRIBUTES

SETTING THE SYSTEM TO USE PHYSICAL ATTRIBUTES

As you receive **Windows for C**, it is set up to use logical attributes rather than physical attributes. If you use physical attributes in the system as configured, you will create errors.

You can easily change to using physical attributes. Logical attributes are implemented in the system by a #define statement near the top of **bios.h**:

```
#define ATT_LOGIC       /*use logical  attributes          */
```

To use physical attributes, all you need to do is to comment out or remove this statement:

```
/*  #define ATT_LOGIC       use physical attributes          */
```

Alternatively, you can #undef **ATT_LOGIC** in your application program:

```
#include <bios.h>
#undef ATT_LOGIC
#include <window.h>
```

If you use this approach, the undefine statement must be inserted between the two #includes.

Users of Windows for Data take note: logical attributes are required; you cannot use physical attributes.

Physical and Logical Attributes Don't Mix

Be aware that once you set the type of attributes that the system is to use, you must be consistent throughout the program. You cannot mix physical and logical attributes in a single program.

You should make your subroutines use physical or logical attributes conditionally if you are going to make some programs that use physical and some that use logical attributes. When and only when logical attributes are being used, the global variable **_lattsw** will be 1. Use this in the conditional test, as in the following code fragment:

```
if(_lattsw)
    sw_att(LNORMAL, &wn);
else
    sw_att(NORMAL, &wn);
```

MONOCHROME ATTRIBUTES

The IBM Monochrome Display Adapter has three "base" attribute states: normal, reverse, and underline. Values are #defined for these attributes: **NORMAL, REVERSE,** and **UNDERLINE.** These base states can be modified by adding to them the values #defined for **HIGH_INT** (high intensity) and **BLINK.**

To set window **wn** to produce normal (white-on-black) text in high intensity, with blink:

```
sw_att(NORMAL + HIGH_INT + BLINK, &wn);
```

With the exception of **UNDERLINE,** these monochrome attributes will produce the same effects on IBM color adapters operating in black and white modes (modes 0 or 2).

AVOID UNDERLINE IN SYSTEM LOGICAL ATTRIBUTES

UNDERLINE is not supported by either the Color/Graphics Adapter or the Enhanced Graphics Adapter (except when emulating the Monochrome Adapter).

If you use **UNDERLINE** in the definitions of the monochrome physical attribute for a system logical attribute, the attribute will not be properly translated in the black and white modes (0 and 2) on either color adapters.

If you wish to use **UNDERLINE,** you will need to have separate logical-attribute columns for the Monochrome Adapter and for the black and white modes of the color adapters.

MANAGING WINDOW COLORS

Differences Between the Enhanced Graphics and Color/Graphics Adapters

The Color/Graphics Adapter (CGA) is the original color adapter supplied by IBM. In 1984, the Enhanced Graphics Adapter (EGA) was introduced. The EGA provides compatibility with the CGA. Programs built to run on the CGA will run identically on the EGA.

One important difference between the two adapters is that the EGA can write output to the screen at any point during the video cycle without causing "snow," whereas this cannot be done on the CGA. **Windows for C detects the presence of the EGA and takes advantage of its faster screen output capability.**

Major differences between the adapters are in graphics modes, which **Windows for C** does not support, but there are also differences in the text modes. There is a wider choice of colors on the EGA, and a special BIOS call installed on the EGA allows you to choose between using the LIGHT modifier for background colors and the BLINK

modifier for foreground colors (see below). The default setting of the EGA has blink enabled, as in the CGA, and accepts the same logical attribute definitions as the CGA.

The discussion here assumes that the EGA is operated with its default settings, so it behaves like a CGA. **Windows for C** does not provide special support for the EGA. If you modify the default settings of the EGA, you are responsible for adapting the attribute definitions to fit the settings you choose.

Foreground and Background Colors

Both foreground and background colors in a window are specified by the attribute byte stored in the attribute member of a window. A function is provided to simplify setting the window attribute value for different colors:

```
color_wn(foreground, background, &wn);
```

The eight basic color values that can be assigned these arguments are #defined in **computer.h** (which is nested within **bios.h**): **BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN**, and **WHITE.**

For reference, the attribute definitions are listed in Table [5.1].

Adding the value **LIGHT** to the basic colors yields the second eight of the 16 colors available as **foreground colors** in the text modes, e.g. specifying **LIGHT + RED** as the foreground parameter will provide light red characters. For convenience, the color **LIGHT + BROWN** is #defined as **YELLOW**. Note that on some monitors **BROWN** will also show up as yellow.

The **LIGHT** colors (including **YELLOW**) are not supported for background colors (because the color values would then overlap the attribute bit that creates blinking).

Adding **BLINK** to the foreground parameter value will cause blinking.

The attribute byte value also can be set directly by using **c_att()**, which accepts foreground and background colors as inputs and returns the corresponding color attribute. For example,

```
sw_att(c_att(RED + LIGHT + BLINK, BLUE), &wn);
```

or more directly,

```
wn.att = c_att(RED + LIGHT + BLINK, BLUE);
```

will set the window attribute for blinking light red characters on a blue background.

Underlining is not supported by the Color/Graphics adapter. Specifying the **UNDERLINE** attribute for text to be displayed on the Monochrome Adapter will show up on a color display as a blue character against a black background. Conversely, characters specified as having a **BLUE** foreground attribute will be appear as an underlined character on the Monochrome Adapter display.

A foreground color of **BLACK** and a background color of **WHITE** will display as **REVERSE** on the Monochrome Adapter display. All other basic colors will display on the Monochrome Adapter as **NORMAL**, and the **LIGHT** colors will display as **HIGH_INT.**

Warning: For the window to appear on the screen with a desired background color inside the window, the window attribute must be specified before setting the window on the screen with **set_wn()**.

To set the attribute to a color, call **color_wn()** after calling **defs_wn()** or **def_wn()** but before calling **set_wn()**.

Changing the Color of Window Borders

A separate member in the window structure specifies the attribute of window borders. The default value for physical attributes is NORMAL, which displays as white on black. To change the value, use the "set window border-attribute" function in combination with **c_att()**:

```
sw_bdratt( c_att(foreground, background), &wn);
```

For example, to make the border **RED** on **BLUE**:

```
sw_bdratt( c_att(RED, BLUE), &wn);
```

You must change the border attribute prior to calling a function that draws the border on the screen. If the window is already on the screen, changing the border attribute will have no effect until the border is redrawn. Chapter [3] describes how to change borders for windows displayed on the screen.

Removing Color Windows: Maintaining the Background

Windows for C will restore the background color of the screen when removing overwrite windows from the screen. The space occupied by the window will be replaced with spaces of the color of the background.

A function is provided to change the background color:

```
color_sc(background);
```

Functions cls() and unset_wn() will clear the screen to the background color set by **color_sc()**. To set the screen color initially, call **color_sc()** with the desired background color and then call **cls()**. Overwrite windows that are later removed from the screen with **unset_wn()** will be replaced with the same background color.

The function **color_sc()** assigns a color value to the background section of the global variable **cl_att**. See "Controlling the Color of the Screen Background" in Chapter 3 for more information on how **cl_att** is used by the system.

SELECTING THE SCREEN BORDER COLOR

When the Color/Graphics Adapter is in use, a border can be drawn around the screen. The color of the screen border is specified by:

```
vid_bdr(color);
```

All sixteen color values may be specified for the border.

A call to this function will have no effect when the Monochrome Adapter is in use.

Warning: Only a physical attribute can be specified as an argument in this function.

"HELLO, WORLD" WITH PHYSICAL COLOR ATTRIBUTES

In Listing [5.3], **hello_wn.c** is rewritten as **hello_wc.c** to provide a demonstration of using physical attributes to produce color windows. The program also illustrates the use of **vid_mode()** (see below) to switch between video adapters on systems that have both the Color/Graphics and Monochrome Adapters (and corresponding monitors). Program **hello_wn.c** is included on the system diskette.

This demonstration program will only display well on a color monitor. If you are going to provide color in a program for general distribution, use the logical attribute system.

DETERMINING AND CHANGING VIDEO MODES

The current video mode number is returned by:

```
rd_mode();
```

The video mode can be selected from within a program by:

```
vid_mode(mode);
```

This function also can be used to switch between the Monochrome and Color/Graphic Adapters if both are installed.

The values that can be assigned to the "mode" parameter are #defined in **computer.h** (which is nested within **bios.h**). The abbreviations are the same as those used by the DOS **MODE** command:

```
          Color/Graphics Adapter Text Modes

BW40            Black and white 40 column screen
BW80            Black and white 80 column screen
CO40            Color 40 column screen
CO80            Color 80 column screen

             Monochrome Adapter Mode

MONO            Monochrome adapter display
```

USING GRAPHICS IN PROGRAMS THAT USE WINDOWS FOR C

You can use vid_mode() to access the graphics modes available on color adapters (see the reference page for vid_mode() in Appendix 2), but **graphics modes are not supported by any of the display functions of Windows for C.**

If you want to use graphics and **Windows for C** in the same program, you can first save the text-mode screen (using **wn0** or a full-screen pop-up window declared for this purpose) and then set a graphics mode. You must not attempt to use the **Window** output functions while in the graphics mode. When you are done with graphics, call **vid_mode()** to set the text mode, and then restore the saved text screen.

Coding Example

The following code saves the screen prior to entering a graphics mode and restores it upon exit. It makes use of functions **sav_wi()** and **unsav_wi()**, which are discussed in Chapter 8. This example assumes that **wn0** has not been used to store a screen image previously.

```
int tmode;
tmode = v_mode;        /*save current mode                    */
sav_wi(&wnØ);          /*save the current screen              */
vid_mode(6);           /*graphics mode                        */
. . .
. . .                  /*your graphics functions              */
. . .
vid_mode(tmode);       /*restore original text mode           */
unsav_wi(&wnØ);        /*restore text screen, free image      */
```

This page intentionally left blank.

Chapter 6

CREATING AND VIEWING MEMORY FILES

CONTENTS

CREATING MEMORY FILES
- Coding Example
- Three Steps to Creating a Memory File
  - Step 1: Declare a File-Record
  - Step 2: Initialize the File-Record
    - Error Handling
  - Step 3: Place Lines in the Memory File
- Reading a Memory File from Disk
  - Error Handling in di_file()
- Writing Lines Directly to a Memory File
  - Warning: Blank Lines In Memory Files
  - Error Handling in sti_file()
  - Memory Handling by sti_file()

VIEWING MEMORY FILES THROUGH WINDOWS
- Setting a Window to View a File
- Viewing a File
  - Moving the Origin of a Window in a File
- Scrolling a File in a Window
  - Turning off the Top and Bottom File Messages
- Performing Operations on a File Displayed in a Window

MODIFYING MEMORY FILES
- Accessing Memory-File Lines
- Modifying and Replacing Memory-File Lines
- Scrolling The Contents of Memory Files

HOW WINDOWS FOR C FUNCTIONS MANAGE MEMORY FILES

CLEARING AND FREEING A MEMORY FILE

This page intentionally left blank.

CREATING AND VIEWING MEMORY FILES

This chapter explains how to build and open windows on **memory files.** A memory file consists of a sequence of strings, each of which is terminated by a newline. Memory files provide an alternative to using the string output functions to write directly to windows on the screen.

With memory files, you can:

* retrieve and display text files stored on disk
  * files can be of any size
  * files can be scrolled horizontally or vertically in a window.

* store information off-screen for later display
  * capture real-time data for later display
  * build and modify menus dynamically

The following chapter explains how memory files can be used for:

* help files
* menus
* communication display buffers

CREATING MEMORY FILES

CODING EXAMPLE

The following code fragment illustrates the three steps required to create a memory file and read in a file, **help.txt**, from the **a:** drive. The maximum number of lines in the file is 201 (including an "end of file" line and the maximum length of line that will be read in is 512 characters. Explanations follow the code.

```
#define MAXLINES 200
#define MAXCOLS  512
. . .
. . .
begin main program

FREC mfile;
char *filespec = "a:help.txt";

if(def_fr(&mfile, filespec, MAXLINES, MAXCOLS) == 0)
    errout("Error creating memory file ", filespec);
if(di_file(&mfile) == 0)
    errout("Error reading ", filespec);
```

THREE STEPS TO CREATING A MEMORY FILE

Step 1: Declare a File Record

The information needed for managing and displaying a memory file is contained in a **memory file** structure of type **FREC** (file record). The first step in creating a memory file is to declare a file record:

```
FREC mfile;
```

Step 2: Initialize the File-Record

A file record is initialized by:

```
def_fr(&mfile, filespec, maxlines, maxcols)
FREC mfile;
char *filespec;
int maxlines;
int maxcols;
```

This function assigns values to members of the memory file record specified as the first argument (**mfile**, in our example). The other arguments have the follow interpretation:

filespec: a string specifying the filename (and optionally, the drive and path) of an ASCII file on disk. This is used if the contents of the memory file are to read from disk or are to be stored to disk. If the memory file will not refer to disk files, this can be specified as **NULLP** (NULL pointer).

maxlines: the number of lines in the memory file (not including a position for an end of file marker)

maxcols: The maximum number of columns allowed on any one line, not including the newline and terminal null that end each line.

Function **def_fr()** also allocates storage for a **memory file array**, which is an array of pointers to FLINE structures. FLINE structures are used to hold information on each line in the memory file. The size of the memory file array equals the specified maximum number of lines in the memory file, plus one additional space for an end-of-file marker. All elements of the array are initialized to NULLP. A pointer to this array is assigned to a member of the memory file record:

```
mfile.farray        pointer to an allocated array of FLINEPTRS
```

See Appendix 5 for a description of the members of the FREC structure and their use in managing memory files.

Error Handling

If there is insufficient memory to allocate the memory file array, **def_fr()** function will return a zero and set _wn_err to MEMLACK.

Step 3: Place Lines in the Memory File

Information can be placed in a memory file either from a disk file or by writing strings directly to the file. These two alternatives are described in the following sections.

READING A MEMORY FILE FROM DISK

In our coding example above, lines were read from a file on disk by calling the "disk-in_file" function:

```
di_file(&mfile)
FREC mfile;
```

This function performs all the work of reading lines from the disk file and adjusting values in the **FREC** so that the video-file function (see below) will work properly. See the reference page on **di_file()** in Appendix 2 for more details.

Error Handling in di file()

If **di_file()** is successful, it returns a 1. Function **di_file()** can encounter a number of different errors. If it encounters a fatal error (one that prevents all lines of the file from being read, it sets global error variable **_wn_err** to a code that indicates the type of error and returns a 0. The possible fatal errors and codes are:

Error	Error Code
Unable to open file	ERR_OPEN
Error reading file	READERR
Insufficient memory to copy file	MEMLACK
File lines exceed maxlines	FILETOOBIG

When **di_file** returns a zero, you can check **_wn_err** to determine the cause of error. See Appendix 6 for more information on system error handling.

Function **def_fr()** specifies the maximum number of lines that can be placed in the memory file. If the file to be read in has more lines than **maxlines**, some of the lines will not be read in. If you want to view the truncated file, you can do so. If a line in the disk file is longer than **maxcols**, the line transferred to the memory file is truncated at that length, but processing continues.

Source is provided for **di_file()** so that you can modify it to work with a different file structure than the sequential array used here.

WRITING LINES DIRECTLY TO A MEMORY FILE

Instead of reading lines from a disk file to the memory file, you can write strings directly to the memory file, using the "string-in_file" function:

```
sti_file(st, frow, &mfile);
char *st;
int frow;
FREC mfile;
```

Argument **st** is the string to be placed in the file, **frow** is the row (0 origin) of the memory file in which a copy of the string is to be placed.

Uses of function **sti_file()** include:

* building menus for display using our **menu2()** function

* storing strings of information off-screen for later display

* transferring lines from an edit buffer to an underlying memory file

Warning: Blank Lines In Memory Files

If you want a blank line in a memory file, you must explicitly write a line consisting of a **newline** character, **'\n'**, on the desired file row. After a memory file is initialized, all lines in the file are "empty." The file viewing functions described later in this section will stop displaying lines at the first empty file line. You must write each line of the memory file up to the last line that you want displayed. This is done automatically by **di_file()**, but you must ensure that this is the case when creating memory files using **sti_file()**.

Error Handling in sti file()

If the string is successfully placed in the memory file, **sti_file()** returns a one. If **sti_file()** encounters an error, it will set global error variable **_wn_err** to a code that indicates the type of error and returns a zero. The possible errors and codes are:

Error	Error Code
Insufficient memory to copy string	MEMLACK
File line number exceeds maxlines	FILETOOBIG

When **sti_file** returns a zero, you can check **_wn_err** to determine the cause of error. See Appendix 6 for more information on system error handling.

The operation of **sti_file()** is subject to the values of **maxlines** and **maxcols** placed in the memory file record by **def_fr()** (see above). The second error listed above occurs if **frow** exceeds **maxlines**.

If the string to be copied is longer than **maxcols**, the string transferred to the memory file is truncated at that length, and **sti_file()** returns a -1. You can use the -1 return to ensure that you place all of long strings in the memory file. The following code fragment will copy long strings on multiple lines, if necessary:

```
int val;
char *stp;
int frow;
FREC mfile;
. . .

[initialize file, point stp to string information.]

. . .
```

```
while((val = sti_file(stp, frow, &mfile)) <= 0)
{
     if(val == 0)
     {
          err_proc();                    /* process error        */
          break;
     }
     stp += mfile.fmaxcol                /* advance stp          */
     frow++;                             /* advance file row     */
}
```

If the string is not fully copied by a call to **sti_file()**, the string pointer is advanced by the number of characters copied and **sti_file()** is called again. The string is advanced by the value of **mfile.fmaxcol**. (The **mfile** member **fmaxcol** equals the value of **maxcols** specified in **def_fr()**, and this is the number of characters placed on the file row when the string length exceeds the limit.)

The **while** loop will exit when the return is > 0, meaning that the copy has been successfully made.

Memory Handling by sti file()

Function **sti_file** allocates memory from the heap (via **malloc()**) to hold a copy of the string passed as an argument. To simplify replacing lines in a memory file, memory allocated previously for the line being replaced is automatically released by **sti_file()**.

VIEWING MEMORY FILES THROUGH WINDOWS

SETTING A WINDOW TO VIEW A FILE

To view a memory file through a window, first place a pointer to a file record (FREC) in the window structure with:

```
sw_mfile(&mfile, &wn);
```

VIEWING A FILE

After the file record has been referenced in a window structure, the window can be placed on the screen and the file displayed within it by a call to the "video_file" function:

```
v_file(&wn);
```

Warning: If you have created a memory file using **sti_file()** and not all of the file can be viewed, check your code to ensure that you have placed newlines on all blank lines. The file viewing functions will stop display at the first empty file line. See above: "Warning: Blank Lines in Memory Files" for more information.

Moving the Origin of a Window in a File

The file record contains the location where the origin of the window will be placed in the file by **v_file()**. When initialized, the file record sets the window origin at the file origin. You can change the location of the window in the file with the "place_memory-file-window" function:

```
pl_mfwn(frow, fcol, &wn);
int frow;
int fcol;
WINDOW wn;
```

The values of the memory file row, **frow**, and column, **fcol**, are measured from a 0 origin.

For example, to view the memory file referenced in **wn** through a window placed at row 5, col 10 of the file:

```
pl_mfwn(5, 10, &wn);
v_file(&wn);
```

Source for **v_file()** is provided to allow you to modify it.

SCROLLING A FILE IN A WINDOW

You can display a memory file in a window and then give the user the ability to scroll through the file using the cursor-pad keys with the "view-scroll_file" function:

```
vs_file(exit_key, &wn);
```

The user will remain within **vs_file()** until he presses **exit_key**. See **computer.h** for key definitions.

Warning: If you use a key that returns an "extended code", you must use the negative of value #defined in **computer.h**. For example, if you want <F10> to be the exit key, use:

```
vs_file(-K_F10, &wn);
```

Table 6.1 lists the keys implemented by vs_file(). The keys assigned to the cursor functions are assigned in **k_vcom()**, the function used by **vs_file()** to implement scrolling.

Source is provided for **k_vcom()**; so you can change key assignments and add other key functions.

Warning: If you have created a memory file using **sti_file()** and not all of the file can be viewed, check your code to ensure that you have placed newlines on all blank lines. The file viewing functions will stop display at the first empty file line. See above "Warning: Blank Lines in Memory Files" for more information.

Turning off the Top and Bottom File Messages

In the default setting, **vs_file()** will display "Top" and "Bottom" of file messages in the window. The display of these messages can be switched on and off by:

```
s_tbfmsg(switch);
```

where **switch** can be either **ON** or **OFF**.

PERFORMING OPERATIONS ON A FILE DISPLAYED IN A WINDOW

As written, **vs_file()**, does not allow any functions to be executed on a memory file, other than the scrolling functions provided by **k_vcom()**. Source is provided for **vs_file()** to allow you to modify it to call your own functions. You can use **vs_file()** as the starting point for building a simple editor, or you can perform more limited operations by assigning the desired operations to function keys and checking for these keys within the key-processing loop of **vs_file()**.

MODIFYING MEMORY FILES

ACCESSING MEMORY-FILE LINES

You can access a line in a memory file with:

```
char *file_lnp(frow, &mfile);
```

which returns a pointer to the specified line in the memory file. Thus, to write the contents of file row number 5 (the sixth line) to a window, you could use the following code fragment:

```
char *file_lnp();
char *stp;

stp = file_lnp(5, &mfile);
v_st(stp, &wn);
```

MODIFYING AND REPLACING MEMORY-FILE LINES

If you wish to modify a line in a memory file, copy the contents of the desired line to a buffer. Make the modifications in the buffer, and then use **sti_file()** to write the contents of the buffer back to the file. For example, to change a line to all upper case:

```
char stbuf[81];          /* file lines no more than 80 char*/
char *file_lnp();
char *stp;
int frow;

/*change file row 5 to upper case                          */
frow = 5;
stp = file_lnp(frow, &mfile);

/*copy contents of file line to buffer                     */
strcpy(stbuf, stp);          /*standard library function  */

/*modify contents of buffer to be all upper case           */
stp = stbuf;             /*start at beginning of buffer    */
while(*stp != '\0')
     toupper(*stp++);    /*standard library function       */

/*replace file line with contents of buffer                */
sti_file(stp, frow, &mfile);
```

In this example, which does not change the length of the file line, it would have been possible to do the operations directly on the file line, rather than using a buffer. The example used the buffer to illustrate the procedure that is required when the operation increases the length of the file line.

Note that when **sti_file()** is used to write to a file row that already contains a file line, as in the above example, the memory used to hold the original line will automatically be freed before the new line is installed in the file.

SCROLLING THE CONTENTS OF MEMORY FILES

The lines in a memory file can be scrolled up or down, creating space for the addition of one or more new file lines without overwriting any existing lines.

The scrolling of lines in a memory file is different than scrolling a window over a memory file. When the lines of a memory file are scrolled, the positions of the lines in a file are actually changed; lines scrolled beyond the top and bottom of the file are lost. When a window is scrolled over a memory file, the contents of the memory file are not changed in any way; only the viewing window is changed.

The file scrolling function is:

```
scrl_file(nrows, dir, beg_row, &mfile)
int nrows;                  number of rows to scroll
int dir;                    direction to scroll
int beg_row;                starting row of scroll region
FREC mfile;                 FREC structure
```

This function will scroll in either direction, starting at the specified file row number. By specifying the #defined value **ALL_ROWS** as the value of **beg_row**, the entire file will be scrolled.

When lines are scrolled in a memory file, the blank lines that are created contain a single **newline** character, **'\n'**. Use **sti_file()** to replace the "empty" lines with text.

For more details and an example, see "Managing An Offscreen Display Buffer" in Chapter 7.

HOW WINDOWS FOR C FUNCTIONS MANAGE MEMORY FILES

When a memory file is created by a call to **def_fr()**, space is allocated for an array of pointers equal to the number of lines in the memory file **plus** one. All of these pointers are set equal to **NULLP** (a null pointer). When lines are written to the memory file by **sti_file()** or **di_file()**, the **NULLP**'s are replaced by pointers to the specified lines. If the file is completely filled with lines, the last position in the array of pointers will still equal **NULLP**, but all other positions will contain the addresses of the file line strings.

The functions that allow you to display a memory file for viewing, **v_file()** and **vs_file()**, treat the first **NULLP** ("empty line") detected in the array as an end-of-file marker. Any file lines that follow the first **NULLP** will not be displayed.

For more detailed information on the internal management of memory files, see **Memory File Structures** in Appendix 5.

CLEARING AND FREEING A MEMORY FILE

When you no longer need the contents of a memory file, you can free the memory allocated to hold the contents by calling:

```
free_file(&mfile)
FREC mfile;
```

Note: This function should have been named clear_file() because it does not actually free the file. Rather it clears the contents of the file. The array of pointers to file lines will still exist after this function returns, but all of the lines of the file will be empty. You can place new lines in the memory file with **sti_file()**.

If you do not want to use a file further and want to free the memory allocated for the array of pointers to file lines, you must free this memory by a call to:

```
free_mem((char *)mfile.farray);
```

This page intentionally left blank.

Chapter 7

HELP FILES, MENUS, AND OFF-SCREEN BUFFERS

CONTENTS

USING A POP-UP HELP FILE
- Preparing the Help File
- Reading Help Files Into Memory
- Displaying Files
- Coding Example
- Multiple Help Files
  - Changing the Filename of a Memory File
  - Coding Example

CREATING AND DISPLAYING POP-UP MENUS
- Preparing to Call a Menu
  - Step 1: Creating a Menu Memory File
    - Menu Format
    - Placing the Menu in a Memory File
  - Defining the Menu Display Window
    - Note to Previous Users of Windows for C
    - Pointing To the Memory File in the Display Window
- Calling the Menu
- Coding Example of a Menu
- Menu Demonstration Program

MANAGING AN OFF-SCREEN DISPLAY BUFFER
- Memory Files as an Alternative to Virtual Screens
- Using a Memory File as an Off-Screen Buffer
  - Scrolling the Contents of a Memory File
  - Coding Example

This page intentionally left blank.

HELP FILES, MENUS, AND OFF-SCREEN BUFFERS

This chapter illustrates the use of **memory files**, introduced in the previous chapter, to provide pop-up help files and menus and off-screen buffers for the storage of real time information.

You must read Chapter 6 to understand this chapter.

USING A POP-UP HELP FILE

With the facilities provided by **Windows for C**, implementing a pop-up help file is very easy. You prepare the help file with any editor, read the contents into a memory file, and call the file-display function. These steps are explained in more detail below.

PREPARING THE HELP FILE

Prepare the contents of the help file using any editor that produces standard ASCII files. Generally, you will want to format the text to the width of the window that you plan to use for the help file, but this is not a requirement. Give it the name you will use in your program.

You could have also have multiple help files, each one covering a different subject. Multiple files are discussed below.

The program must know where to find the help file. The simplest way is to ask the user to keep the file on the default drive.

READING HELP FILES INTO MEMORY

Before the help file can be displayed in a window, it must be read into memory. This can be done either at the beginning of a program or whenever the file is requested.

Multiple help files can be read in all at once or when requested. If you plan to read files in as requested, you will need only one memory file. Free the contents of the file each time you have finished displaying a requested file.

See Chapter 6 for information on creating memory files and placing information in them.

DISPLAYING FILES

You can display the help file in a window and then give the user the ability to scroll through the file using the cursor-pad keys with the "video-scroll_file" function:

```
vs_file(exit_key, &wn);
```

If the file will fit entirely within a window without the need for scrolling, display the file with:

```
v_file(&wn);
```

CODING EXAMPLE

Listing 7.1 shows how a help file can be displayed. The program is included on the system diskette in file **tut_help.c**. This example assumes that file **help.txt** will be

available to the program on the default drive. The help file is read in only when requested and freed after the user is done with it. The help file is called up and exited by pressing <F1> while within a key processing loop. The key processing loop exits on <F10>.

In this example, calling help is the only function within the key processing loop. In a practical program, there would be other key functions within the loop.

MULTIPLE HELP FILES

Changing the Filename of a Memory File

If you wish to choose from among multiple help files for display, you can use a variation of the code in Listing 7.1. When help is called, you need to place the name of the appropriate help file in the **mfile** structure before calling **di_file()**. The file name is in member **mfile.fn**. Place the name in this member by direct assignment:

```
FREC mfile;

mfile.fn = "filename.txt";
```

Coding Example

If the file names were in an array of strings, **fnames[]**, the code to read in and display the **j**th file (0 origin) would be:

```
mfhelp.fn = fname[j];
if(! di_file(&mfhelp))                /*read in help file    */
     err_out("error reading file ", fname[j]);
vs_file(HELPKEY, &mfhelp);           /*turn over to viewer  */
free_file(&mfhelp));
```

Note that this code is identical to the central part of the program in Listing 7.1, except that the filename is specified before reading in and displaying the file.

CREATING AND DISPLAYING POP-UP MENUS

Windows for C contains a function, **menu2()**, that simplifies the task of providing pop-up menus within your programs. (The "2" in "menu2" indicates that this is a revision of our earlier **menu()** function.)

Function **menu2()** will place a pre-defined menu on the screen, allow the user to move from item to item with the cursor control keys, highlight the current item, and return a number identifying the item selected by the user. The original screen image is restored after the selection is made.

The menu can be arranged either horizontally or vertically. The menu can be wider or longer than the menu window if desired.

Source code is provided for **menu2()** on the system diskette.

PREPARING TO CALL A MENU

Before you can call **menu2()** to get a menu choice, you must prepare the menu and the display window. The steps involved are:

* Create a menu memory file
* Define a display window for the menu
* Point to the menu file in the display window

Step 1: Creating a Menu Memory File

Function **menu2()** requires that the menu to be displayed be in a **memory file.** You can either construct menus with an editor, place them in separate ASCII files on disk, and read them into a memory file, or you can code them directly in your program.

Menu Format

* Menus can consist of any number of items.
* Menu items can be arranged in one or more rows, and each row can have one or more items. The same number of items must be on each row except the last one.
* The same space must be allocated to each item in the menu. Use spaces to pad items to make them all the same length.

For example, you might have a menu of 6 items, with the longest item 8 characters. They could be in a vertical format of 6 rows, with each item eight characters. If you wanted a horizontal format, you could have 1 row of 6 items or two rows of three items. If you use a horizontal format, make each item 9 characters, including 1 space to separate the menu items when they are displayed.

Placing the Menu in a Memory File

You can prepare the menu with an editor and read it into a memory file with **di_file().** See Chapter 6 for details. Alternatively, you can code the menu lines and place them in a memory file with **sti_file().** Here is a fragment for preparing a simple menu in code. The menu is two rows of 3 items. Each item is 10 characters.

```
FREC mfile;

def_fr(&mfile, NULLP, 2, 30);
sti_file("Apples    Oranges   Pears", 0, &mfile);
sti_file("Avocados  Bananas   Pineapples", 1, &mfile);
```

The memory file is first defined and initialized by the call to **def_fr().** No name is specified for the file, because the name will never be referenced. The memory file will have two rows of thirty characters each. The two calls to **sti_file()** place the specified strings in the two rows of the memory file.

Defining the Menu Display Window

The window for displaying the menu can be of any size. The user will be able to scroll to the boundaries of the memory file, just as with any memory file. Use **defs_wn()** to define and initialize the window to the desired size. Remember to allow for margins and borders when defining the window. Function **defs_wn()** assigns margins of 1 on each side of the window. If you do not want these margins, use **def_wn()** instead of **defs_wn()**, or use **sw_margin()** to change the margins.

For example, the menu size is 30 columns by two rows and you want a single-line border on the display window, you could define the window with:

```
defs_wn(&vmenu, 0, 0, 4, 34, BDR_LNP);
```

This allows 4 columns for the border and the margins, yielding the desired 30 columns for the inside dimensions of the window.

Note to Previous Users of Windows for C

In versions of **Windows for C** prior to version 4.0, window dimensions had to be chosen carefully to avoid unwanted horizontal scrolling. In the present version, horizontal scrolling will not occur as long as the inside dimensions of the window are at least as wide as a row of the menu, as defined in the call to **def_fr()** by **maxcol**. For example, horizontal scrolling would not occur if you had specified the window width in the previous example as 36.

Further, there is no need, as there was formerly, to allow a space at the end of a menu item. The largest menu item in a vertical-format menu can completely fill the inside of the window.

Pointing To the Memory File in the Display Window

After the memory file and window are both defined, the display window must be made to point to the memory file containing the menu:

```
sw_mfile(&mfile, &vmenu);
```

CALLING THE MENU

After the menu is placed in a memory file and the display window is defined, the menu is called to the screen by:

```
menu2(&wn, qitems, itemlength, rowitems, initial_item);
WINDOW wn;
int qitems;
int itemlength;
int rowitems;
int initial_item;
```

These arguments have the following meaning:

qitems: the number of items in the menu

itemlength: the number of column positions filled by each item

rowitems: the number of items in a single row

initial_item: the number (1 origin) of the item where the highlight cursor-bar should be placed initially

When **menu2()** is called, the menu is popped up on the screen. The initial item is highlighted, and control is turned over to the user. The user can move the cursor to items by using the cursor keys. When he presses <Enter>, the menu is popped down and the item number (1 origin) of the selected menu item is returned.

Thus, to call up a menu with 7 items arranged in rows of 3 items per row (except for the last row), with 10 characters per menu item (including padding-spaces), use:

```
int item_selected;
. . .
. . .

item_selected = menu2(&menu_wn, 7, 10, 3, 1);
```

CODING EXAMPLE OF A MENU

Listing 7.2 provides an example of a vertical pop-up menu. The source code is included on the system diskette as **vmenu.c**.

The menu for the example is contained in an ASCII file, **vmenu.txt** (also on the system diskette). The menu has 10 lines. Each line contains one menu item, and the longest item is 10 spaces, including one space at the end of the text. The menu **itemlength** is 10, items per row (**itemrow**) 1, and the total number of items (**qitems**) is 10.

A window for the menu display is declared as **WINDOW** vmenu and initialized with the following characteristics: origin at 0,0; single-line border, left and right margins of 1; overall width of 14 spaces (10 inside borders and margins); overall height of 8 rows (6 inside of borders).

The menu file is read into memory, information messages printed, and control turned over to the user. Pressing function key **<F9>** calls **menu2()** to display the menu file. Function **k_vcom()** interprets cursor-pad keystrokes for moving through the menu. Pressing <Enter> exits from the menu and the underlying screen is restored. Pressing **<F10>** exits from the program.

MENU DEMONSTRATION PROGRAM

The system diskette contains source for a demonstration program that incorporates a pop-up menu in a program that also calls in a text file for display in a window. Compile and link **dem_menu** for a further illustration of how pop-up menus can be configured.

MANAGING AN OFF-SCREEN DISPLAY BUFFER

MEMORY FILES AS AN ALTERNATIVE TO VIRTUAL SCREENS

Some windowing systems provide "virtual screens." Information can be written to virtual screens without affecting the actual screen display until a call is made to a function that opens or updates a window onto the virtual screen. This capability is especially useful for real-time processes, such as communication programs, where you may want to keep track of incoming information but not to display it constantly.

Memory files provide the same capability as a virtual screen (but use much less memory). Use **sti_file()** to write strings of data to a memory file, **file_lnp()** to access lines in the file, and **vs_file()** to open a window on the file when desired.

USING A MEMORY FILE AS AN OFF-SCREEN BUFFER

Scrolling the Contents of a Memory File

Using memory files, you can create off-screen text buffers of any desired number of columns and rows, up to the maximum value of an integer (about 32,000 on the 8086).

The amount of memory used for each line in the buffer will equal (approximately) the memory required for two pointers plus an integer for each line in the file, plus the memory allocated to hold the strings actually placed in the file. The memory used in this buffer can be freed at any time by calling **free_file()**.

A function, **scrl_file()** is provided for scrolling lines in a memory file; so it is easy to write to the buffer until it is full, and then to scroll the file upward by one line each time you write an additional line to the file.

The file scrolling function is:

```
scrl_file(nrows, dir, beg_row, &mfile)
int nrows;                      number of rows to scroll
int dir;                        direction to scroll
int beg_row;                    starting row of scroll region
FREC mfile;                     FREC structure
```

This function will scroll in either direction, starting at the specified file row number. By specifying the #defined value **ALL_ROWS** as the value of **beg_row**, the entire file will be scrolled.

When lines are scrolled in a memory file, the blank lines that are created contain a single **newline** character, **'\n'**. Use **sti_file()** to replace the "empty" lines with text.

If a scrolling operation causes file lines to be scrolled off the top or bottom of the file, the memory used to hold these lines is freed, and the contents of these lines can no longer be accessed.

Coding Example

Listing 7.3 is for a function, **sti_buf()**, that uses **scrl_file()** to assist in maintaining an off-screen buffer. Function **sti_buf()** will place a specified string in the first empty line of the specified memory file, if space is available. If the file is full, all of the lines in the file will be scrolled up one line and the specified string will be placed in the last line of the file.

Upward scrolling of the memory file will cause the first line in the file to be scrolled off the top of the file and lost. If you want to save this in a permanent location, you would need to check on whether scrolling will occur before calling **sti_buf()**. If scrolling will occur, move the first line to its permanent location and then call **sti_buf()**. One way to accomplish this would be to always store the first line of the file in a temporary location before calling **sti_buf()**. If the return value indicates that scrolling occurred, move the first line from its temporary place to permanent storage. Copy the new first line to the temporary storage and continue.

Function **sti_buf()** returns values that tell when file scrolling has occurred and if a line has been truncated.

Before using **sti_buf()**, you must first initialize the memory file by calling **def_fr()**.

The source for **sti_buf()** is included on the system diskette. It is not in the **Windows for C** library, but you can add it if you want to use it.

This page intentionally left blank.

Chapter 8

ADVANCED TOPICS, UTILITIES, AND DEMONSTRATIONS

CONTENTS

WINDOW VIEWING OF MULTIPLE FILES
- Memory Requirements
- Handling Multiple Files
  - Using the Same Window for More than One File
  - Overlapping Windows
  - Multiple Windows on the Same File
- Demonstration of Viewing Multiple Files
  - Managing Multiple Files with Arrays of Structures

MOVING INFORMATION FROM AND TO WINDOWS
- Moving the Character Contents of Windows
  - Window Dimensions: dim_wn()
  - Allocating Memory for the String: get_mem() and size_wn()
  - Applications of v_mova()
- Using a Window as an Edit Buffer
  - Coding Example
- Copying the Contents of a Window to a File
- Moving Character and Attribute Contents of Windows
  - Coding Example

MOVING, SAVING, AND RESTORING WINDOW IMAGES
- Moving Windows
  - Moving Windows on Memory Files
  - Moving Windows Under User Control
- Saving and Replacing Window Images
  - Memory Management
- Character-Graphics Animation
  - A Demonstration of Moving Images
- Storing Window Images on Disk

HIGHLIGHTING AND CHANGING ATTRIBUTES
- Highlighting a Specified Number of Characters

FORMATTING TEXT FOR PRINTING WITH WINDOWS
- A Demonstration of Printing Side-by-Side Labels

GRAPHING FUNCTIONS

USING AND MODIFYING SYSTEM GLOBALS

USING ALTERNATIVE DISPLAY ADAPTERS

STRING UTILITIES

MISCELLANEOUS UTILITIES
- **Low-Level Character and String Functions: v_qch() and v_st_rw()**
- **Macros for Window Row and Column Quantities**
- **Error Exit Function**
- **Duplicating Window Structures**

DEVELOPING YOUR OWN APPLICATIONS
- **User-Reserved Pointers**
- **Building New Functions**

ADVANCED TOPICS, UTILITIES, AND DEMONSTRATIONS

This chapter covers a variety of topics

Viewing multiple files in multiple windows
Overlapping windows
Moving information from windows
Text editing
Saving, moving, and restoring window images
Character animation
Highlighting text and changing attributes
Printing window contents
Graphing functions
Using alternative display adapters
Miscellaneous utilities
String utilities

Also described are demonstration and tutorial programs related to several of these topics:

Viewing multiple files	demo_wn.c
Moving window images	dem_cmov.c
Printing window contents	prt_labl.c
Graphing Functions	dem_grph.c

Source for these programs is included on the system diskette. You will need to compile and link these programs to obtain an executable file.

WINDOW VIEWING OF MULTIPLE FILES

MEMORY REQUIREMENTS

Chapters 6 and 7 explained how to create memory files and display them in windows. Subject only to memory limitations, you can have as many files and windows as you want in your program. The windows themselves require very little memory (about 50 bytes per window). Memory use will be closely related to the amount of text in the files (empty lines and trailing blanks use little space). When you are done with a memory file, you can free the memory used to store the contents of the file by calling **free_file()**.

HANDLING MULTIPLE FILES

When your program has multiple memory files resident, you can open windows on them simultaneously. See the previous chapters for the steps required to display a single memory file in a window. Follow these steps for each file you want to display.

Using the Same Window for More than One File

You can use the same window to display multiple files, as long as you display the files sequentially. You can't have the window on the screen in two locations at the same time. To display the memory file controlled by FREC **mfile** in window **wn**, use:

```
sw_mfile(&mfile, &wn);
vs_file(&wn);
```

To display another file in this window, repeat the above statements, using the new memory file record in the first statement.

Overlapping Windows

Pop-up windows opened onto memory files can overlap one another without creating problems, as long as the following rules are observed:

Only the top window can be written to, scrolled, or otherwise updated. For most applications, this is not an important restriction, because normally you will want to have the window on top when you are going to change information in it.

Remove windows in the order they were placed on the screen. If you do not, the underlying screen will not be properly restored. This is the normal sequence. The top window is always the active window and will be the one from which you call up the next window, which will in turn become the active window until it is removed from the screen or another window is called up.

Multiple Windows on the Same File

The management structure for viewing memory files was designed primarily for having one window open on a memory file at one time. You can open multiple windows on the same file, but you must have a separate memory file record (**FREC** structure) for each window. The contents of the **FRECs** should be identical, except for the members that specify the location of the window in the file, **mfile.wfr** and **mfile.wfc**. Refer to Appendix 5 for information on the members of the **FREC** structure.

As an example, consider setting up two windows for the same file. The sequence of steps would be:

1. Declare two windows and two memory file records:

```
WINDOW wn1, wn2;
FREC mfile1, mfile2;
```

2. Take all of the steps required to initialize **mfile1** and fill the associated memory file with information.

3. Copy the members of **mfile1** to **mfile2**. You will need to copy the following members:

```
mfile1.fn
mfile1.fmaxlines
mfile1.fmaxcol
mfile1.farray
mfile1.ln_q
mfile1.c_q
mfile1.wfr
mfile1.wfc
```

   You must fill the memory file before you duplicate the memory file record.

4. Reference the second memory file record in **wn2**, that is:

```
sw_mfile(&mfile2, &wn2);
```

You are now ready to use **v_file()** or **vs_file()** with both windows.

If you want **wn1** and **wn2** initially to display different parts of the file, use **pl_mfwn()** to place the windows in the desired locations before calling them to the screen.

Warning: Except for the values of **mfile.wfr** and **mfile.wfc**, the two structures must have identical values. If you change any lines in the file (using **sti_file()**), you should copy the values of **mfile.c_q** and **mfile.ln_q** to the other structure to ensure that they are the same in both structures.

DEMONSTRATION OF VIEWING MULTIPLE FILES

Included on the system diskette is a source file, **demo_wn.c**, for a program that reads in multiple files and allows the user to view them in different windows. Compile, link, run, and review this program to get a better idea of capabilities for file management.

For purposes of study, the source code can be printed. It can also be viewed by running **demo_wn.exe**, because one of the files which this program permits the user to view is **demo_wn.c**.

Managing Multiple Files with Arrays of Structures

Multiple files and windows can be managed by using arrays of structures to define the windows and file-records. **Demo_wn** uses an array **wn[]** to define six windows and an array **fr[]** to define four file-records. Standard subroutines can then manage different windows and files by using different index values in **wn[]** and **fr[]**.

MOVING INFORMATION FROM AND TO WINDOWS

Chapter 3 described the functions for reading and writing single characters and attributes in a window. This section describes functions for reading and writing larger parts of a window in a single call. These "move" functions move information in both directions, **OUT** from a window and **IN** to a window. The functions are primarily intended for applications, like editing and highlighting, where information is first read from a window, stored or modified, and then later returned to the window.

MOVING THE CHARACTER CONTENTS OF WINDOWS

You can transfer the character contents of designated parts of a window to a string, and vice versa, with function "video_move-ASCII":

```
v_mova(st, wnp, part, direct)
char st[];                  string for window data
WINDOWPTR wnp;              pointer to a window structure
int part;                   part-of-window parameter
int direct;                 direction-of-move parameter
```

Seven different **parts** of a window can be moved with this function:

CH	the character at the location of the virtual cursor
ENDROW	from present location of virtual cursor to end of row
ROW	row on which the virtual cursor is located
COL	column on which cs is located

ENDCOL	from the present location of virtual cursor to the last row in this column
ENDWIND	row on which the virtual cursor is located to end of window
WIND	entire window

The values of the window **parts**, **ENDROW**, etc., are #defined in **wfc_defs.h**.

The direction of the move parameter in the call determines whether the data is moved out from the screen to a string (direct = **OUT**) or in to the screen from a string (direct = **IN**). The values of **IN** and **OUT** are #defined in bios.h.

When you move a string into a window, it will be written with the current window attribute.

Window Dimensions: dim_wn()

Function **v_mova()** operates on the **working dimensions** of a window. You can change the dimensions (using **dim_wn()**). When a window is set on the screen, the dimensions are set to **INSIDE**. This means that only information inside the margins and borders will be moved by **v_mova()**. If you want to include the border, you need to set the working dimensions to **FULL**. To change the working dimensions of a window, use:

```
dim_wn(size, &wn);
int size;
WINDOW wn;
```

The argument size can be either **INSIDE** or **FULL**.

Programming Hint: If you change the working dimensions to full, change them back when you are done. The value of the working dimensions is maintained in **wn.setsw**. If you consistently keep it to **INSIDE** while a window is on the screen and to **FULL** when it is off, you will always be able to tell if a window is on the screen or not.

Allocating Memory for the String: get mem() and size wn()

The **string** specified in the call must be large enough to hold the character contents of the portion of the window read out. You can use **Windows for C** functions to allocate the required memory from the heap. The following code fragment illustrates this for a full window:

```
char *st, *get_mem();
WINDOW wn;

if((st = get_mem((size_wn(WIND, &wn)/2 + 1)) == NULLP)
      error_process();
```

Function **size_wn()** returns the number of bytes within the working dimensions of a window. It counts two byte for each position in a window: one for the character and one for the accompanying attribute. This is the amount of space actually filled by the window in the video display buffer. Because **v_mova()** moves only characters, we divide by two, and then add one for the terminal null.

Function **get_mem()** calls the standard C library function **malloc()**. It returns a **NULLP** and sets the global error code **_wn_err** if the requested memory cannot be allocated. See Appendix 6.

Applications of v_mova()

Use **v_mova()** whenever you want to store or manipulate the character contents of a part of the window. We use the equivalent of this function in the library routines that change the attribute of a part of a window (**v_natt()**) and that copy the character contents of a window to a file (**copy_wc()**) and to a printer (**prt_wn()**). These functions are described in later sections of this chapter.

Use **v_mova(OUT)** to store the character contents of a window in user memory for later restoration by **v_mova(IN)**.

Because **v_mova()** does not do word wrap, cursor advance, or provide other options, it is much faster than **v_st()**. You can use this to advantage in many applications.

Function **v_mova()** is useful in constructing a simple editor, as described in the next section.

USING A WINDOW AS AN EDIT BUFFER

Function **v_mova()** can assist in constructing simple editors. To edit a single line, define a one-row window without borders within which you will operate. Locate the window where you want it on the screen. It could be placed over existing text or at a place for the entry of new text. Do not set it on the screen. Window functions will operate properly after a window is defined, whether or not it appears on the screen.

For entry of text use **v_ch()** to write to the window. At the simplest level, where only writing and destructive backspacing are allowed, all you need to write text is **v_ch()**.

If you want to insert and delete characters, you will need to read information to the right of the cursor and move it one space in the appropriate direction. You can use **v_mova() OUT** for **ENDROW**. This string will include trailing blanks. You can remove them with **strip_wh()** (see "String Utilities" later in this chapter). Locate the virtual cursor where you want to replace the line, and use **v_st()** to write the string back to window.

Programming Hint: When editing, it is helpful to disable advance of the virtual cursor:

```
sw_csadv(OFF, &wn);
```

and keep track of it yourself. Otherwise the virtual cursor will sometimes unexpectedly advance to the next row.

Do not disable auto-clear-to-end-of-row, because you do want to erase text to the right of what you write back in to the window.

Coding Example

See the function **rd_line()**, which appears at the end of the program listing for **tutor.c** in Chapter 4.

COPYING THE CONTENTS OF A WINDOW TO A FILE

Function **copy_wc()** copies the character contents of a window to a file:

```
copy_wc(dimen, filename, dmode, wnp)
char dimen;                    INSIDE or FULL window copy
char *filename;                name of the file to copy to
char *dmode;                   mode of disk operation
WINDOWPTR wnp;                 pointer to a window structure
```

The **dmode** parameter specifies whether the file is to be opened in the **FWRITE** mode, which erases contents of the file specified, or **FAPPEND** mode, which appends the copied information to existing information in the file. If the specified file does not exist, it is created. **FWRITE** and **FAPPEND** are #defined in **wfc_defs.h**.

The filename can include the drive and path if supported by your compiler and version of DOS.

MOVING CHARACTER AND ATTRIBUTE CONTENTS OF WINDOWS

Function **v_mov()** moves both the character and attribute contents of windows to and from strings. Its definition, arguments, function are identical to **v_mova()** (see above), except that rather than dealing with characters, it deals with attribute-character pairs.

This function is a low-level function used to build higher-level functions in the **Window** library. For most purposes, you will find higher-level functions that are easier than **v_mov()**.

Because attributes can have the value zero, which is the value of the standard string terminator, **'\0'**, the strings in which **v_mov()** stores information are not standard strings. To distinguish them from standard strings, they are termed **video strings**. Function **v_mov()** returns the number of bytes it moves to the specified video string. A terminal null **is not** appended.

See the discussion of **v_mova()** above for more information.

Note: The character precedes its associated attribute in a video string.

Coding Example

The following example illustrates the use of **v_mov()**. This is intended as purely as an example. The function accomplished in this code is much more easily done using the library function **save_wi()**, described later in this section.

To move a window into a video string, obtaining the memory from the heap, use the following code:

```
char *vst, *get_mem();
WINDOW wn;
. . .
. . .
. . .
[declare the window, initialize it, and write in it]

dim_wn(FULL, &wn);
vst = get_mem(size_wn(FULL, WIND, &wn));
v_mov(vst, &wn, WIND, OUT);
```

You can restore the window at a later time by calling:

```
v_mov(vst, &wn, WIND, IN);
```

You do not need to set the window on the screen.

Warning: Do not change window dimensions between calls.

MOVING, SAVING, AND RESTORING WINDOW IMAGES

MOVING WINDOWS

Function **mv_wi()** can be used to move any type of window from one screen location to another. It works whether a window is a pop-up or an overwrite window and whether it is a direct-display window or a window on a memory file. If it is a pop-up window, the underlying images are properly handled and saved. The function definition is:

```
mv_wi(rw, co, &wn)
int rw;                    screen row of new window origin
int co;                    screen column of new window origin
WINDOW wn;                 window structure
```

Moving Windows on Memory Files

Function **mv_wi()** stores the image of the window on the screen before making the move. This is not necessary for windows on memory files, because the window information is already in memory. You can reduce the transitory memory requirement (and perhaps move the window faster) by using the following sequence of calls:

```
unset_wn(&wn);
pl_wn(row, col, &wn);
v_file(&wn);
```

This is only applicable to windows on memory files.

Moving Windows Under User Control

Be aware that moving window images requires substantial computation. If you are going to allow a user to move a window dynamically, we recommend that you break the process into two steps: 1) provide indicators of the window location, such as reverse video corners or borders, and allow the user to move these with the cursor keys; 2) when the new location is determined, move the window in one step. This will look better than moving the window after each cursor movement.

SAVING AND REPLACING WINDOW IMAGES

Underlying **mv_wi()** are lower-level functions for storing and replacing window images:

sav_wi(&wn) saves the window image to memory.

unsav_wi(&wn) replaces on the screen a previously saved image and frees the memory that held the image.

repl_wi(&wn) replaces on the screen a previously saved image and retains the image in memory.

The **window image** functions operate on the window image within the area defined by a WINDOW structure. Window images are the full dimensions of the window, including the borders.

Memory Management

Memory allocation is handled automatically by the window image functions. The only time that you need to be concerned about the memory allocations associated with using the window image functions is if you have not unsaved previously stored images. These images will consume memory until explicitly released. If you do not need the images further and do not want to call **unsav_wi()**, which will replace the stored image on the screen, free memory by calling:

```
free_mem(wn.storp);
```

CHARACTER-GRAPHICS ANIMATION

The window image functions simplify moving images around the screen. This capability can be used to create movement and simple animation for games.

The images to be moved can be built with the block-graphics characters provided on the PC as part of the "extended ASCII" character set. (These extended characters will be much easier to use if they can be assigned to single keys, using a keyboard redefinition utility). The images can be built with an editor and then read into a window using **di_file()** and **v_file()**. Alternatively, they can be created directly in a window with a sequence of calls to **v_st()**. Place them on the screen in a window with a null border (bdr_0) and move them as desired with **mv_wi()**.

To create simple animation, create several sequential-movement images, store them as window images in separate windows, and place and remove them from the screen in sequence. In this case, you will want to retain the images in memory, so use **repl_wi()** to place them on the screen and **unset_wn()** or **cl_wn()** to erase them from the screen.

A Demonstration of Moving Images

The demonstration program **dem_cmov.c** included on the system diskette shows how windows can be used to move images around on the screen. Movement is rapid enough to make it practical for gaming applications. (On the AT, movement is actually more rapid than desirable. A time-delay should be inserted in the movement loop to slow it down.)

Program **dem_cmov** was written using physical attributes and illustrates what must be done to use physical attributes on both black and white and color monitors.

STORING WINDOW IMAGES ON DISK

Saved window images can be stored in disk files using standard C library functions.

Save the window image with **sav_wi(&wn)**. Function **sav_wi()** will place a pointer to the video string containing the window image in wn.storp.

Open the file in the untranslated or binary mode. Consult your compiler manual for the appropriate file open command. After you have opened the file, you can write the video string to the file with the following line of code:

```
write(file, wn.storp, size_wn(FULL, WIND, &wn));
```

The size (in bytes) of the **video string** that holds the image in memory is returned by size_wn(FULL, WIND, &wn).

When you no longer need the stored window image, remember to free the memory allocated to hold it by calling **free(wn.storp)**.

In order to redisplay the stored image, read the file back into a video string and point to it with **wn.storp**, then call **unsav_wi()** or **repl_wi()**. You will need to use a window of the same size as the one for which the image was saved.

HIGHLIGHTING AND CHANGING ATTRIBUTES

Highlighting is accomplished by changing the attribute of selected portions of the screen. A function is provided for this purpose:

```
v_natt(att, part, &wn);
```

This function sets the part of wn specified to attribute **att**. It operates within the current working dimensions of the window. Seven different **parts** of a window, relative to the location of the virtual cursor, can be moved with this function: **CH, ENDROW, ROW, ENDCOL, COL, ENDWIND,** and **WIND.** See the discussion in a previous section of this chapter under "Moving the Character Contents of Windows" for definitions of the window parts. The values of the window **parts**, ENDROW, etc., are #defined in **wfc_defs.h**.

As an example of changing text attributes, you might wish to highlight a row of text appearing in a window. To do this, locate the virtual cursor on the desired line and call:

```
v_natt(REVERSE, ROW, &wn);
```

The function **v_natt()** is not restricted to highlighting but can be used to set any desired attribute to a specified part of a window.

HIGHLIGHTING A SPECIFIED NUMBER OF CHARACTERS

If you want to highlight less than a row of a window, you can do this in two ways:

* define a one-row window of the desired length, locate it over the text to be highlighted, and call **v_natt()** for the window.

* call function **v_att()** for each position to be changed

See the source code for **menu2()** for an example of the first approach. See Chapter 3 for more information on **v_att()**.

FORMATTING TEXT FOR PRINTING WITH WINDOWS

Window functions can simplify the often-frustrating task of printing independent groups of text in parallel columns. For example, on an invoice form, you may wish to have the "Ship To" address adjacent to the "Bill To" address. With standard string functions, this is relatively complex. With **Windows for C**, it is easy.

The print window function

```
prt_wn(&wn);
```

copies the character contents of a window to a printer. Use this in combination with other **Window** functions to format and print text.

You can format independent groups of text side-by-side, or in any other desired arrangement, by placing windows of appropriate size in the desired locations on the screen and then using a **Window** string output function to write the text into the windows. The windows can be borderless if boxes are not desired around the text. If you do want borders and your printer cannot print the IBM block-graphics characters, define a border using a character, such as "*", from the standard character set.

To transfer the formatted text to paper, define a window of width equal to that of the screen (paper) and of height appropriate to the text that is to be copied to the printer. Place it in the proper location with pl_wn(), and call prt_wn().

A DEMONSTRATION OF PRINTING SIDE-BY-SIDE LABELS

The system diskette includes a program, **prt_labl.c**, that illustrates the formatting and printing capabilities of **Windows for C**. This program reads labels stored sequentially in a disk file and places them on the screen in the format required by five-line, three-abreast labels. The **prt_wn()** function is then used to print the windows in this format.

Program **prt_labl** must be compiled and linked before running. After linking, a sample file of addresses, **test.adr** can be printed by issuing the command from DOS, "prt_labl test.adr".

Complete instructions for running and modifying the program for other label formats are included in **prt_labl.c**. In its present form, the program must be re-compiled to print different formats, but it could be easily changed to allow the user to select from and modify label formats stored on disk.

GRAPHING FUNCTIONS

Windows for C provides two functions that can be used to draw simple bar graphs in text modes. One function, **v_axes()**, draws axes for the graph. The second, **v_bar()**, draws the bars. Horizontal or vertical bar graphs can be drawn.

The use of the graphing routines is illustrated in **dem_grph()**, for which source code is included on the diskette.

Beyond their direct usefulness, the graphing routines are interesting as examples of the versatile, non-obvious ways in which windows can help to simplify programming tasks. Both the axes and bars are drawn by defining windows of the appropriate size

and location. The axes are filled with reverse-video spaces and the bars with block-graphics characters.

USING AND MODIFYING SYSTEM GLOBALS

Windows for C uses a number of global variables to control various aspects of its operation. All of these global variables are defined in **window.h** and given external declarations in **vextern.h**.

Brief descriptions of the global variables are provided in **window.h**. You can make use of the values contained in these variables for your own program. For example:

v_mode contains the current video mode.
v_rwq contains the number of rows in the video display
v_coq contains the number of columns in the video display
_ibmega equals one when an Enhanced Graphics Adapter is present

You can also modify the system global variables, but exercise care, especially with those variables that specify the location of the video display buffer (see below). **Do not modify _l_ptr, v_contig, or _d_seg.**

Global variable **tv_upd** is part of the system for compatibility with **TopView** and **Microsoft Windows**. It can be set to zero to disable checking for the existence of these programs. This may be desirable if you are developing a program that you know will not use **TopView** or **MS Windows**, because certain IBM "compatibles" are incompatible with the code that does this checking. See the chapter entitled "Microsoft Windows and TopView Compatibility" for more information.

USING ALTERNATIVE DISPLAY ADAPTERS

You can adapt **Windows for C** to operate with display adapters that have special features, such as more columns or rows than the standard IBM display adapters. To do this, you must correctly set the values of the global variables that specify the size of the screen and the video display buffer. Do this in the special "user-initialization" routine, **u_init()**, which is called by the system initialization routine, **init_wfc()**. The variables that you must set properly are:

```
v_seg               /* video regen buffer segment          */
v_coq               /* number of columns in screen display*/
v_rwq               /* number of rows in screen display    */
v_pbytes            /* number of bytes in vrb page         */
ADDR v_vrb;         /* address of video regen buffer --    */
                    /*      offset and segment             */
```

The **ADDR** structure is defined as:

```
typedef struct addr_struct
{
    int off;          /*address offset                     */
    int seg;          /*address segment                    */
} ADDR, *ADDRPTR;
```

You will need to consult the documentation for the display board for the correct values to assign to these variables.

STRING UTILITIES

The library contains several string utilities to assist in editing and entry tasks. See the listing under **stringf.c** in Appendix 2 for more details on these functions.

lower_st(st)	Each position in the string is converted to lowercase.
skip_wh(st)	Skips the leading white space in the string and returns a pointer to the first non-whitespace character.
stblank(q)	Allocates memory to hold a string of length q and the terminating '\Ø' and initializes the string to blanks. Returns a pointer to the string.
strcpyp(d, s)	Copies the contents of the source string to the destination string. Returns a pointer to the terminating '\Ø' in the destination string.
strip_wh(st)	Strips the trailing white space from the string and repositions the terminating '\Ø'.
upper_st(st)	Converts the string to uppercase.

Note: the functions that return pointers have been so declared in **vextern.h**. You do not need to declare them as functions returning pointers in functions you build. If you do not like this convenience feature, delete the declarations in **vextern.h**.

MISCELLANEOUS UTILITIES

LOW-LEVEL CHARACTER AND STRING FUNCTIONS: v_qch() and v_st_rw()

The lowest-level character output function in **Windows for C** is:

```
v_qch(character, q, &wn);
char character;          character to write
int q;                   number of characters to write
WINDOW wn;               window structure
```

This function does not advance the virtual cursor and does not observe window boundaries.

The lowest-level string output function in **Windows for C** is:

```
char *v_st_rw(st, q, &wn)
char *st;                string to be put
int q;                   number of char in string
WINDOW wn;               window structure
```

This function is similar to **v_st()**, except:

1) you can specify how many characters, maximum, are to be written.

2) only one row of output to a window will be written; the function returns when it detects a newline.

3) the function does not observe window boundaries.

See the reference pages in Appendix 2 for more details on these functions.

MACROS FOR WINDOW ROW AND COLUMN QUANTITIES

Macros have been defined that return the quantities of the rows and columns in a window:

```
row_qty(&wn);
col_qty(&wn);
```

You will need these quantities for many purposes. As with all macros, be careful of side effects if you incorporate them in complex expressions.

ERROR EXIT FUNCTION

An error-exit function is available:

```
errout(st1, st2);
char *st1;
char *st2;
```

This function will print both strings on the screen at the bottom, one after the other, and call the system function **exit()** with an argument value of 1, the normal error exit code.

Having two strings available is convenient. One can be a fixed message and the other can be a variable argument. For example, to report an error opening a file with the name contained in string variable **filename**:

```
errout("Error opening file ", filename);
```

DUPLICATING WINDOW STRUCTURES

A function is available for copying the values of the members in one window into another:

```
dup_wn(&dwn, &swn)
WINDOW dwn;                    destination window structure
WINDOW swn;                    source window structure
```

Use this function when you want to replicate a window that has already been created.

DEVELOPING YOUR OWN APPLICATIONS

The library of functions provided by **Windows for C** will permit you to efficiently program just about any screen task you can imagine. To select the best functions for the task you wish to accomplish, first study the examples provided, and then look through the summary listing in Appendix 2 that groups the functions by category of use. Refer to the reference pages for detailed explanations of function and usage. In many instances, it will be possible to start with the code provided in one of the demonstration programs, adapting sections of it to your needs.

USER-RESERVED POINTERS

Two pointers are reserved within **WINDOW** structures for users. You can use these pointers to refer to information you want associated with a specific window. They

can point to other structures, so there is no limit on the number of variables you can tie to a window structure. These pointers will be maintained in future revisions.

BUILDING NEW FUNCTIONS

The **Window** library contains an extensive set of primitive functions. These permit you to construct higher-level functions that meet your needs. The library routines are flexible and easy to combine. A little thought will usually allow you to quickly adapt **Window** functions to meet your special needs.

Chapter 9

MICROSOFT WINDOWS AND TOPVIEW COMPATIBILITY

CONTENTS

VIDEO MANAGEMENT UNDER MICROSOFT WINDOWS AND TOPVIEW
- How Windows for C Operates Under MSW/TV
  - Incompatibility With Some IBM PC Compatibles
- Program Control of Screen Updates Under MSW/TV
- Direct Control of Screen Updates Under MSW/TV

MSW/TV PROGRAM INFORMATION FILES

RUNNING WINDOW DEMONSTRATION PROGRAMS UNDER MSW/TV

This page intentionally left blank.

MICROSOFT WINDOWS AND TOPVIEW COMPATIBILITY

NOTE: In this chapter we use the abbreviation MSW/TV to refer to both Microsoft Windows and TopView. The compatibility features described in this chapter apply equally to Microsoft Windows and TopView. Exactly the same functions of **Windows for C** are used to interface Microsoft Windows and TopView. The interface routines were originally written for TopView and, thus, bear names that reflect this origin, but this does not affect their applicability to MS Windows.

Windows for C (and **Windows for Data**) are fully compatible with **Microsoft Windows** and IBM's **TopView**. Programs that use Windows for C for screen output and keyboard input can operate within a window in MS Windows and TopView and run in the background.

Programs built using **VCS Windows** will automatically run under TopView and MS Windows. You do not need to buy Microsoft's or IBM's Programmer's Toolkits or incorporate any special code in your programs. Compatibility is handled automatically by Windows for C, which detects the presence of MS Windows or TopView and adjusts its screen handling to conform to their requirements.

The Program Interface File (PIF) format is exactly the same in TopView and MS Windows; thus the demonstration programs for which PIF files are provided on the system diskette can be run under either TopView or MS Windows.

VIDEO MANAGEMENT UNDER MICROSOFT WINDOWS AND TOPVIEW

Programs that can operate in the background mode under MSW/TV are termed "well-behaved" or "MSW/TV compatible." There are a number of conditions that programs must fulfill to be MSW/TV compatible. (See the TopView or Microsoft Windows documentation for a complete list of conditions.)

The most important and difficult condition for MSW/TV compatibility is that programs not write information directly to the computer's video regeneration buffer. This means either that 1) programs must access the screen only through DOS or BIOS calls, which put one character to the screen at a time and are, therefore, very slow, or 2) programs must explicitly deal with MSW/TV's video management functions.

To permit programs to operate in the background while another program has control of the screen, MSW/TV provides the capability, via a "Get Video Buffer" function, to assign each program an individual "video buffer" located in user memory. The individual video buffers are distinct from the computer's video regeneration buffer that controls the information displayed on the video screen; thus output written to the individual video buffers will not appear on the screen. MSW/TV provides an Update Video Display function to move information from the individual video buffers to the regeneration buffer when the program has control of or shares the video display.

HOW WINDOWS FOR C OPERATES UNDER MSW/TV

In normal operation, **Windows for C** directly writes all screen output to the computer's video buffer, providing rapid screen updating. Under MSW/TV, all screen output is directed to the individual video buffer assigned by MSW/TV. **Window** output functions call the MSW/TV Update Video Display function to transfer this information to the screen when a program is operating in the foreground mode.

Windows for C adjusts to operation under MSW/TV automatically. No special coding is required. You can write your programs just as you would for operation under DOS. All of the screen-handling requirements of MSW/TV will be taken care of by **Windows**

for C. Window **functions automatically update the video display when in the MSW/TV environment.**

Incompatibility with Some IBM PC Compatibles

Checking for MSW/TV availability causes **Windows for C** to be incompatible with some IBM PC compatible computers. On the IBM PC, MSW/TV adds several new BIOS interrupt calls which are assigned to previously unused interrupts. To determine if MSW/TV is resident, we must call one of these new interrupts. Some IBM PC compatibles use these interrupts to perform other tasks. When this is the case, **Windows for C** incorrectly interprets the return values causing the program to crash. This problem can be eliminated by setting the global variable tv_upd to 0. The variable tv_upd is found in window.h. When tv_upd is initialized to 0, **Windows for C** will not check to determine if MSW/TV is resident.

PROGRAM CONTROL OF SCREEN UPDATES UNDER MSW/TV

The rate at which screen changes occur under MSW/TV can vary dramatically, especially when the display is controlled by the Color/Graphics Adapter, depending upon how the MSW/TV Update Video function is used. Each call to the Update function takes an appreciable amount of time; thus if many calls to this function are made while updating a single screen, performance will suffer.

We have tried to minimize the number of Update Video calls made by **Window** functions, consistent with the need to insure that the screen is updated at the end of each function call. The functions cls(), cl_wn(), mv_rws(), v_f(), mv_wi(), repl_wi(), unsav_wi(), v_mov(), and v_natt() only make one Update Video call. The character output functions, v_co() and v_rw(), also make one Update call. The string output functions, v_st(), v_fst(), v_st_nop(), and v_st_rw(), make one Update call per screen row. Because these last functions generally only write to one or two screen rows in a single call, there is little performance penalty to updating after writing each screen row.

The screen updating algorithms used within **Windows for C** may degrade performance under MSW/TV when repeated calls to character or string output functions are made to perform a single screen update. In this case, it may be desirable to disable the automatic screen-updating capability of the functions at the beginning of the procedure and to explicitly call the Update Video function at the end of the procedure.

DIRECT CONTROL OF SCREEN UPDATES UNDER MSW/TV

Automatic screen updating of **Window** functions is controlled by a global variable, tv_upd. This variable is 1 in the MSW/TV environment and 0 otherwise. The **Window** output functions call the MSW/TV Update Video function only when tv_upd is 1. Thus, automatic updating can be disabled simply by setting tv_upd to 0.

A function, v_tv(), is provided in the library for calling the MSW/TV Update Video function. See the reference page for v_tv() for details of its use.

To directly control screen updating in a part of your program:

1) save the value of tv_upd (so it can be restored at the end);

2) set tv_upd to 0 to disable automatic updating;

3) carry out the procedures for which screen updating is not desired;

4) issue a Update Video call, using v_tv();

5) restore the original value of tv_upd.

For an example, see v_file(), for which source code is provided on the system disk.

MSW/TV PROGRAM INFORMATION FILES

MSW/TV requires specific information about a program in order to run it. For programs that you develop, this should be supplied along with the program in a special file called a Program Information File (PIF), with the extension PIF. This file is created with a utility supplied with the MSW/TV Programmers Toolkit. (Alternatively, the information required can be supplied to the user, and the user can enter the information directly into MSW/TV when adding your program to MSW/TV).

When preparing a PIF for your program, you will need to know that **Windows for C**:

1) does not swap any software interrupt vectors,

2) permits you to answer no to each of the four yes/no questions.

The use of **Windows for C** in your program does not place any special restrictions on its use under MSW/TV.

RUNNING WINDOW DEMONSTRATION PROGRAMS UNDER MSW/TV

Program Information Files (PIFs) have been supplied on the system diskette for **demo_wn(), dem_menu()**, and **demo_wfd() (Windows for Data)**; thus these can be run under MSW/TV without the need to supply program information to MSW/TV. If you wish to run other demonstration programs under MSW/TV, refer to the previous section for information you will need to supply MSW/TV.

This page intentionally left blank.

TABLES AND LISTINGS

ALL CHAPTERS

Table 3.1: **Window Members, Default Settings, and Change Functions**

Window Feature	Window Member[1] (default value)	Change Function[2]
Window origin and size	rb [row begin] re [row end] cb [col. begin] ce [col. end]	mod_wn(rb, cb, row_q, col_q, &wn)
pop-up	popup (OFF)	sw_popup(switch, &wn)
Window name	wname (NULLP)	sw_name("name", &wn)
Margins	l_mg [left margin] r_mg [right margin] (Ø, Ø)	sw_margin(l_mg, r_mg, &wn)
Attribute	att (LNORMAL)	sw_att(attribute, &wn)
Border Attribute	bdratt (LNORMAL)	sw_bdratt(attribute, &wn)
Word Wrap	options parameter[3] (ON)	sw_wwrap(switch, &wn)
Auto Scroll	scr_q[4] (ON)	sw_scroll(switch, &wn)
Place Cursor	options parameter[3] (OFF)	sw_plcsr(switch, &wn)
Clear end row	options parameter[3] (ON)	sw_cleor(switch, &wn)
Virtual cursor advance	options parameter[3] (ON)	sw_csadv(switch, &wn)
Logical attributes	larray	sw_latt(latt, &wn)
Memory file pointer	frp	sw_mfile(&mfile, &wn)

(Continued)

Table 3.1 (Continued)

(1) The "window member" is the declared name of the window-structure member (or members) that control the given window characteristic. Where the names are not self explanatory, descriptive name are given in brackets. The default values assigned to the members by **defs_wn()** are given in parentheses.

(2) Function arguments denoted as "switch" can have the #defined values **OFF** (Ø) or **ON** (1).

(3) This window feature is controlled by a bit-value within a single **options integer** member, which has the name **wrap**.

(4) The **scr_q** member specifies the number of lines that will be scrolled when an attempt is made to write to a full window. The default value is one (which is the #defined value of **ON**).

Table 3.2: **Logical Attribute Definitions**

Logical Attribute	Monochrome Attribute	Color Attribute	Primary Function or Purpose
LDOS	normal	white/black	DOS display
LNORMAL	normal	white/blue	Regular text
LHIGHLITE	high int.	bright white/blue	Emphasize text
LREVERSE	reverse	blue/white	Emphasize text
LURGENT	high int/blink	red/black	Urgent attention
LHELP	high int.	blue/white	Help text
LERROR	reverse	red/black	Error messages
LMESSAGE	high int.	blue/white	Information message
LFIELDI	high int.	cyan/blue	Inactive entry field
LFIELDA	reverse	black/cyan	Active entry field
LMARK	reverse	blue/white	Mark text area
LNODISPLAY	Ø	blue/blue	Non-display of text
LBLACK	reverse	black/blue	Black foreground
LBLUE	reverse	blue/white	Blue on white
LGREEN	normal	green/blue	Green foreground
LCYAN	normal	cyan/blue	Cyan foreground
LRED	reverse	red/blue	Red foreground
LMAGENTA	reverse	magenta/blue	Magenta foreground
LBROWN	normal	brown/blue	Brown foreground
LWHITE	normal	white/blue	White foreground

Logical attribute names are #defined in **def_att.h**. Physical attributes are assigned to logical attributes in **att_glob.h**.

Table 5.1: **Physical Attribute Definitions**[1]

Monochrome Mode	Color Modes
NORMAL	BLACK
UNDERLINE	BLUE
REVERSE	GREEN
HIGH_INT[2]	CYAN
BLINK[2]	RED
	MAGENTA
	BROWN
	WHITE
	LIGHT[2]
	YELLOW[3]

(1) The physical attributes listed here are #defined in **computer.h**.

(2) This is an "attribute modifier" and must be added to one of the basic physical attributes to form a legitimate value.

(3) This attribute is defined for convenience. It is identical to LIGHT + BROWN.

Listing 5.1: **Initializing the Logical Attribute Array**

```
#include <bios.h>

u_init()
{
    int row;

    if(_ibmega)                          /*if EGA active                    */
    {
        if(v_mode == 1 || v_mode == 3)  /*if color mode                    */
            /*copy column 2 of datt_tbl[][] to latt[]                       */
            s_latt(2, _attrowq, _attcolq, datt_tbl, latt);
    }
}
```

Listing 5.2: **Window-Specific Logical Attributes**

```
#include <bios.h>
#include <window.h>

#define LODD Ø
#define LODDER 1

#define ULATTQ 2
#define UPATTQ 2

uatt_tbl[ULATTQ][UPATTQ]
{
    {REVERSE, c_att(RED, GREEN)},                          /*LODD          */
    {REVERSE + BLINK, c_att(GREEN, MAGENTA)}               /*LODDER        */
};

ulatt[ULATTQ];                          /*user logical attribute array      */

main()
{
    WINDOW wn;

    init_wfc();                         /*initialize window system          */
    defs_wn(&wn, 1Ø, 2Ø, 1Ø, 5Ø, BDR_DLN); /*initialize window               */
    sw_latt(ulatt, &wn);                /*set logical attributes to ulatt[] */
    sw_att(LODD, &wn);                  /*use LODD for output in wn         */
    v_st("This output is written with logical attribute LODD.", &wn);
    return;
}

u_init()                                /*called by init_wfc()              */
{
    int latt_col;

    if(v_mode == 1 || v_mode == 3)      /*if color mode                     */
        latt_col = 1;                   /*use second column of table        */
    else
        latt_col = Ø;                   /*use first column of table         */
    s_latt(latt_col, ULATTQ, UPATTQ, uatt_tbl, ulatt); /*copy col to ulatt[]*/
    return;
}
```

Listing 5.3: **Hello World in Color Using Physical Attributes**

```
/* hello_wc.c -- demo of color capabilities using physical attributes       */

#include <bios.h>
#undef ATT_LOGIC                          /*set for physical attributes       */
#include <window.h>

main()
{
     WINDOW wn;
     int kc;

     vid_mode(CO8Ø);                                  /*8Ø column color display  */
     vid_bdr(YELLOW);                                 /*yellow screen border     */
     color_sc(GREEN);                                 /*clear attrib. to green   */
     cls();                                           /*clear with green spaces  */
     defs_wn(&wn, 5, 2Ø, 1Ø, 4Ø, &bdr_dln);
     color_wn(WHITE, BLUE, &wn);                      /*must follow defs_wn      */
     sw_bdratt(c_att(LIGHT + BROWN, BLACK), &wn);     /*yellow window border     */
     set_wn(&wn);                                     /*set window on screen     */
     v_st("\nHello, world\n", &wn);
     mv_cs(24,Ø, &wnØ);                               /*locate for msg at bottom */
     sw_att(c_att(RED, BLACK), &wnØ);                 /*set attribute in wnØ     */
     v_st("Do you want to change to a Monchrome Display Adapter? (y/n): ", &wnØ);
     pl_csr(&wnØ);                                    /* cursor at end of message*/
     kc = ki();                                       /*wait for keystroke       */
     if(kc == 'y' || kc == 'Y')
         vid_mode(MONO);                              /*go to Monochrome display */
     return;
}
```

Table 6.1: **File Viewing Key Assignments**[1]

Key	Function
Cursor arrow keys	one space in direction of the arrow; except that left and right arrows cause horizontal scrolling of five spaces when cursor is at screen edge.
Ctrl-left-arrow	five spaces left
Ctrl-right-arrow	five spaces right
Ctrl-PgUp	five spaces up
Ctrl-PgDn	five spaces down
Home	top of file
End	end of file
PgUp	page up
PgDn	page down

(1) Key assignments are made in **k_vcom()**, for which source is provided.

Listing 7.1: **Displaying and Scrolling a Help File**[1]

```
/*tut_help.c -- displaying and scrolling a help file                    */

#define HELPFILE "help.txt"
#define HELPKEY -K_F1                    /*"extended keys" return negative code*/
#define EXIT -K_F10
#define MAXCOL 76                        /*window will have only 76 col for txt*/
#define MAXLINES 200                     /*the help file is < 200 lines        */

#include <bios.h>
#include <window.h>

main()
{
    WINDOW hwn;
    FREC mfhelp;                                 /*memory file for help file   */
    int kc;                                      /*key code                    */

    init_wfc();                                  /*initialize program          */
    cls();
    def_fr(&mfhelp, HELPFILE, MAXLINES, MAXCOL);  /*init. file record          */
    defs_wn(&hwn, 16, 0, 9, 80, BDR_LNP);        /*define help window          */
    sw_popup(ON, &hwn);                          /*make a pop-up window        */
    sw_att(LHELP, &hwn);                         /*set attributes for help     */
    sw_bdratt(LHELP, &hwn);
    sw_mfile(&mfhelp, &hwn);                     /*install pointer to mfhelp   */

    v_st("Press key F1 to toggle help window\n" ,&wn0);
    v_st("When not in help window, press F10 to exit program.", &wn0);
    while((kc = ki()) != EXIT)
    {
        if(kc == HELPKEY)                        /*help requested              */
        {
            if(! di_file(&mfhelp))               /*read in help file           */
                errout("error reading file ", HELPFILE);
            vs_file(HELPKEY, &hwn);              /*turn over to viewer         */
            free_file(&mfhelp);
        }
    }
}
```

(1) Source for this program is on the system diskette in file **tut_help.c**

Listing 7.2: **Demonstration of a Vertical-Format Pop-Up Menu**[1]

```
/* vmenu.c -- tutorial showing vertical pop-up menu

INTRODUCTION

This program shows the steps required to implement a pop-up menu, using the
functions provided in the Window library.

The menu chosen for display is a vertical menu with one item per row.  The menu
is longer than the menu window; so vertical scrolling occurs.

To keep the program simple, no files other than the menu-information file are
read in for display.  For a more complex and realistic example of implementing a
pop-up menu, see dem_menu.c.

OUTLINE OF PROGRAM

The menu for this example is contained in an ASCII file, vmenu.txt, (also on
the system diskette).  The menu has 10 lines.  Each line contains one menu item,
and the longest item is 10 spaces.  The total number of items (qitems) is 10,
the length of of a menu item (itemlength) is 10, and the items per row
(rowitems) 1.

The menu is displayed in a window, declared as WINDOW vmenu_wn, with the
following characteristics:  origin at 0,0; single-line border, left and right
margins of 1; overall width of 14 spaces (10 inside borders and margins);
overall height of 8 rows (6 inside of borders).

The menu file is read into a memory file, information messages printed, and
control turned over to the user.  Pressing function key <F9> calls menu2()
to display the menu file and interpret cursor-pad keystrokes for moving through
the menu.  Pressing <Enter> exits from the menu and the underlying screen is
restored.  Pressing <F10> exits from the program.

*/
```

(Continued)

Listing 7.2 (Continued)

```
#include <bios.h>
#include <window.h>

#define FILENAME "vmenu.txt"
#define MAXLINES 10                     /*maximum number of lines in mem. file*/
#define QITEMS 10                       /*number of items in menu             */
#define ITEMLENGTH 10                   /*maximum no. of columns in menu item */
#define ROWITEMS 1                      /*items per row                       */
#define ROWCOL 10                       /*items/row times itemlength          */

WINDOW vmenu_wn;                        /*window for file display             */


main()
{
    int nlines;                         /*number of lines read into memory    */
    int itemnumber;                     /* number or selcted menu item        */
    int kc;                             /* keycode value                      */
    FREC mfile;                         /* memory file record for menu        */

    cls();                              /*clear screen                        */
    v_qch('C', 2000, &wn0);             /*fill screen with C's, just to fill  */

    defs_wn(&vmenu_wn, 0, 0, 8, ROWCOL + 4, &bdr_ln); /*define vmenu_wn window*/
/*-------------------------------------------------------------------------*/
/*Note: menu2() will treat vmenu_wn as a pop-up; no need to set popup switch */
/*-------------------------------------------------------------------------*/

    def_fr(&mfile, FILENAME, MAXLINES, ROWCOL);   /*define menu file record   */
    if(di_file(&mfile) == 0)                  /*read menu file to memory file */
        errout("error in reading file ", FILENAME);
    sw_mfile(&mfile, &vmenu_wn);              /*point to menu in vmenu_wn     */

/*-------------------------------------------------------------------------*/
/* Ready for menu display.  Inform user of the rules                       */
/*-------------------------------------------------------------------------*/

    mv_cs(21, 0, &wn0);
    v_fst("To call pop-up menu, press function key F9.  \nUse cursor ", &wn0);
    v_fst("pad keys to move through menu.  \nPress Enter key to select ", &wn0);
    v_fst("item and exit from menu.  \nPress F10 to exit from program." , &wn0);

    mv_csr(25, 0, &wn0);                /*hide cursor                         */

/*-------------------------------------------------------------------------*/
/*  check for keystrokes and implement correct action.                     */
/*  menu2() will allow user to user cursor keys and will redraw window     */
/*  display when scrolling occurs.                                         */
/*-------------------------------------------------------------------------*/
```

(Continued)

Listing 7.2 (Continued)

```
    while((kc = ki()) != -K_F10)       /*exit on  F10                    */
    {
        if(kc == -K_F9)
        {
            /*-------------------------------------------------------------*/
            /*  The following call implements the pop-up menu              */
            /*-------------------------------------------------------------*/
            itemnumber =  menu2(&vmenu_wn, QITEMS, ITEMLENGTH, ROWITEMS, 1);

            /*display item number in reverse video in full-screen window    */

            wn0.att = LREVERSE;
            mv_cs(20, 0, &wn0);
            v_printf(&wn0, "Item %d selected", itemnumber);
            wn0.att = LNORMAL;          /*restore normal attribute          */
        }
    }
    mv_csr(24, 0, &wn0);                /*place cursor on last line         */
}
```

---

(1) Source for this program is on the system diskette in file **vmenu.c**

Listing 7.3: **Function for Writing a String to an Off-Screen Buffer**[1]

```
/* sti_buf.c -- an example of how sti_file() and scrl_file() can be used to
                maintain an "off-screen buffer".

FUNCTION

Before calling this function, a memory file must have been initialized with a
call to def_fr(). The maximum number of lines and the maximum number of columns
in the file will be dependent on your requirements.

This function will place the specified string in the first empty line of the
memory file, if space is available. If the file is full, all of the lines in
the file will be scrolled up one line and the specified string will be placed in
the last line of the file.

Upward scrolling of the memory file will cause the first line in the file to be
scrolled off the top of the file and lost.

CALL

sti_buf(st, &mfile)
char *st;                       pointer to string to be placed in file
FREC &mfile;                    FREC structure

RETURNS

= -2 if string is truncated and file is scrolled

= -1 if string is truncated and file not scrolled

= Ø if unable to allocate memory

= 1 if full string is written and file not scrolled

= 2 if full string is written and file is scrolled

CAUTIONS

Before using this function, the memory file must be initialized by a call to
def_fr().

*/

(Continued)
```

Listing 7.3 (Continued)

```
#include <bios.h>

sti_buf(st, mfp)
char *st;
FRECPTR mfp;
{
    int ln_q;
    int retval;
    int scroll;                          /* = 1 if no scroll; = 2 if scroll    */

    scroll = 1;
    ln_q = mfp->ln_q;                    /*avoid indirection                   */

    if(ln_q >= mfp->fmaxline)            /*no more room -- must scroll file    */
    {
        scroll = 2;
        if(scrl_file(1, UP, ALL_ROWS, mfp) == 0)       /*scroll whole file    */
            return(0);
        if((retval = sti_file(st, ln_q - 1, mfp)) == 0)/*replace last line    */
            return(0);
    }
    else                                 /*room available -- simply append     */
        if((retval = sti_file(st, ln_q, mfp)) == 0)    /*add new line to end */
            return(0);

    return(retval * scroll);
}
```

(1) Source for this program is on the system diskette in file **sti_buf.c**

This page intentionally left blank.

APPENDIX 1

#INCLUDE FILES

```
/*bios.h -- Top level include file for Windows for C

    ************** Copyright 1985 by Vermont Creative Software ***************

VERSION: 4.04

*/

/*-------------------------------------------------------------------------*/
/*  Define the version number of Windows for C                             */
/*-------------------------------------------------------------------------*/
#define WFC_VER "v4.04"

/*-------------------------------------------------------------------------*/
/* Implement Logical Attributes as the default                             */
/* To use physical attributes, comment out the following #define           */
/*-------------------------------------------------------------------------*/
#define ATT_LOGIC                          /*use logical  attributes       */

/*-------------------------------------------------------------------------*/
/*  Include compiler definition header file                                */
/*-------------------------------------------------------------------------*/
#include <wfc_comp.h>

/*-------------------------------------------------------------------------*/
/*  linked list definitions                                                */
/*-------------------------------------------------------------------------*/
#include <llist.h>

/*-------------------------------------------------------------------------*/
/*  computer/terminal specific defintions                                  */
/*-------------------------------------------------------------------------*/
#ifdef MSDOS
#include <computer.h>
#endif

#ifdef UNIX
#include <terminal.h>
#endif

/*-------------------------------------------------------------------------*/
/* logical attribute definitions                                           */
/*-------------------------------------------------------------------------*/
#include <def_att.h>

/*-------------------------------------------------------------------------*/
/* standard error codes definitions                                        */
/*-------------------------------------------------------------------------*/
#include <wn_errd.h>

/*-------------------------------------------------------------------------*/
/*  Windows for C definitions                                              */
/*-------------------------------------------------------------------------*/
#include <wfc_defs.h>
```

bios.h (continued)

```
/*-------------------------------------------------------------------------*/
/*  Windows for C structures                                               */
/*-------------------------------------------------------------------------*/
#include <wfc_stru.h>

/*-------------------------------------------------------------------------*/
/*  External variables                                                     */
/*-------------------------------------------------------------------------*/
#include <vextern.h>                    /*extern declarations of global var.'s*/

#ifdef UNIX
#include <ux_xtrn.h>
#endif
```

```
/*computer.h  - contains definitions for the specific computer being used

    These definitions are for the IBM PC/XT/AT

*/

#define RW_QSCRN 25                     /*number of rows on PC video screen   */
#define CO_QMAX 80                      /*max number of cols on PC video scr. */
#define CO_80 80                        /*80 columns in display               */
#define CO_40 40                        /*40 columns in display               */
#define V_OFFSET 0                      /*offset of the video regen buffer    */
#define MONO_SEG 0xB000                 /*monochrome board video segment      */
#define GRPH_SEG 0xB800                 /*graphics board video segment        */
#define BYTES_80 4096                   /*number of bytes on page in 80 cols  */
#define BYTES_40 2048                   /*number of bytes on page in 40 cols  */
#define MONOMODE 7                      /*mode number for monochrome board    */
#define EGA43ROWS 43                    /*number of rows on EGA screen        */
#define EGA43PBYTES 8192                /*number of bytes on page in 43 rows  */

/*--------------------------------------------------------------------------*/
/*  The following are definitions for the DOS interrupt routines INT21h     */
/*--------------------------------------------------------------------------*/
#define DOS_INT 0x21
#define GET_DATE 0x2a00
#define GET_TIME 0x2c00

/*--------------------------------------------------------------------------*/
/*The following are definitions for the video interupt routines INT10H      */
/*--------------------------------------------------------------------------*/
#define VI_INT 0x10                     /*video interupt number             */
#define VI_MODE 0x0                     /*set mode                          */
#define VI_CSR_TYPE 0x0100              /*set cursor type                   */
#define VI_PL_CSR 0x0200                /*place cursor                      */
#define VI_RD_CSR 0x0300                /*read cursor positions             */
#define VI_SET_DP 0x0500                /*set active display page           */
#define VI_UPSCROLL 0x0600              /*scroll up                         */
#define VI_DNSCROLL 0x0700              /*scroll down                       */
#define VI_RD_CHATT 0x0800              /*read char-att at csr position     */
#define VI_WR_CHATT 0x0900              /*write char-att at csr position    */
#define VI_COLOR 0x0B00                 /*set color pallete or border       */
#define VI_WR_TTY 0X0E00                /*write teletype char at csr pos    */
#define VI_VSTATE 0x0F00                /*current video state               */

/*--------------------------------------------------------------------------*/
/*The following definitions are for control of attribute bytes              */
/*                                                                          */
/*  Attribute values for the Monochrome Adapter                             */
/*--------------------------------------------------------------------------*/
#define NORMAL 0x07                     /*attribute base state              */
#define UNDERLINE 0x01                  /*       ditto                      */
#define REVERSE 0x70                    /*       ditto                      */
#define HIGH_INT 0x08                   /*attribute added state             */
#define BLINK 0x80                      /*attribute added state             */
```

computer.h (continued)

```
/*-------------------------------------------------------------------------*/
/* Color attribute values for use with the Color/Graphics Adapter          */
/*-------------------------------------------------------------------------*/
#define BLACK   0
#define BLUE    1
#define GREEN   2
#define CYAN    3
#define RED     4
#define MAGENTA 5
#define BROWN   6
#define WHITE   7

#define LIGHT   8
#define YELLOW  14                      /* = LIGHT + BROWN                 */

/*-------------------------------------------------------------------------*/
/* For use with vid_mode(): parameters as defined in DOS 2.0 MODE          */
/*-------------------------------------------------------------------------*/
#define BW40  0
#define CO40  1
#define BW80  2
#define CO80  3
#define MONO  7

/*-------------------------------------------------------------------------*/
/*The following are for the keyboard interupt routines INT 16H             */
/*-------------------------------------------------------------------------*/
#define KI_INT 0X16                     /*keyboard interupt number          */
#define KI_READ   0x0                   /*read buffer                       */
#define KI_CHECK  0x0100                /*check buffer                      */

/*-------------------------------------------------------------------------*/
/* -- defines key values used in Window functions                          */
/*                                                                         */
/* IBM INT 16H returns the following key code as normal codes              */
/* WFC keyboard functions, ki(), ki_chk(), and ki_cum() return positive    */
/* codes                                                                   */
/*-------------------------------------------------------------------------*/
#define K_BACK 8
#define K_ESC 27
#define K_SPACE 32
#define K_LINE 10
#define K_ENTER 13
```

computer.h (continued)

```
/*-------------------------------------------------------------------------*/
/* IBM INT 16H returns the following key code as "extended codes"          */
/* WFC keyboard functions, ki(), ki_chk(), and ki_cum() return negative    */
/* codes                                                                   */
/*-------------------------------------------------------------------------*/
#define QK_BREAK 1
#define K_F1 59
#define K_F2 60
#define K_F3 61
#define K_F4 62
#define K_F5 63
#define K_F6 64
#define K_F7 65
#define K_F8 66
#define K_F9 67
#define K_F10 68
#define K_HOME 71
#define K_UP   72
#define K_PGUP 73
#define K_LEFT 75
#define K_RIGHT 77
#define K_END  79
#define K_DN   80
#define K_PGDN 81
#define K_INS  82
#define K_DEL  83
#define QK_LEFT   115
#define QK_RIGHT  116
#define QK_END    117
#define QK_PGDN   118
#define QK_HOME   119
#define QK_PGUP   132

/*-------------------------------------------------------------------------*/
/* defines values for block-graphics characters                            */
/*-------------------------------------------------------------------------*/
#define LIGHT_SHADE  178              /*lightest shading character        */
#define MEDIUM_SHADE 177              /*medium shading character          */
#define DARK_SHADE   176              /*dark shading character            */
#define SOLID        219              /*solid bar character               */
```

```
/* wfc_defs.h -- definitions for Windows for C

    ************** Copyright 1985 by Vermont Creative Software **************


*/

typedef int (*PFI)();                   /*PFI - pointer to a function         */
                                        /*      returning an integer          */

/*-------------------------------------------------------------------------*/
/*  If not already defined by wfc_comp.h, then define NULLP, NULLFP and NULL*/
/*-------------------------------------------------------------------------*/
#ifndef NULLP
#define NULLP (char *)0                 /*NULL pointer to data                */
#endif

#ifndef NULLFP
#define NULLFP (int(*)())               /*NULL pointer to function returning  */
                                        /*an integer                          */
#endif

#ifndef NULL
#define NULL NULLP
#endif

#define TAB '\t'                        /*C definition of ASCII tab number    */
#define NEWLINE '\n'                    /*C definition of ASCII newline       */
#define BACKSPACE '\b'                  /*C definition of ASCII backspace     */

#define VPSTMAXLEN 133                  /*buffer size for use with v_printf() */

/*-------------------------------------------------------------------------*/
/*  Definitions for use with mv_rws()                                      */
/*-------------------------------------------------------------------------*/
#define UP   6                          /*moves rows up (mv_rws)              */
#define DOWN 7                          /*moves rows down (mv_rws)            */

/*-------------------------------------------------------------------------*/
/*  Definitions for scrl_file()                                            */
/*-------------------------------------------------------------------------*/
#define ALL_ROWS -1                     /*scroll all rows in file             */

/*-------------------------------------------------------------------------*/
/*  Bit switch definitions for use with v_st(), v_st_rw(), v_plst(), etc.  */
/*-------------------------------------------------------------------------*/
#define NO_WRAP    0                    /*No wrap, cs adv, clear in v_st_rw   */
#define WRAP       1                    /*Specifies word-wrap in v_st_rw, etc */
#define NO_CLEAR   2                    /*disable auto clear to end row       */
#define NO_CS_ADV  4                    /*disable cs advance on v_st_rw       */
#define PL_CSR     8                    /*enable auto place of csr at cs      */
```

wfc_defs.h (continued)

```
/*-------------------------------------------------------------------------*/
/*  Definitions for use with v_mov(), v_mova()                             */
/*-------------------------------------------------------------------------*/
#define CH      0                   /*size definition for v_mov, etc      */
#define ENDROW  1                   /*size definition for v_mov, etc      */
#define ROW     2                   /*        ditto                       */
#define ENDWIND 3                   /*        ditto                       */
#define WIND    4                   /*        ditto                       */
#define ENDCOL  5                   /*        ditto                       */
#define COL     6                   /*        ditto                       */

#define OUT 0                       /*type definition for v_mov, etc      */
#define IN  1                       /*        ditto                       */

/*-------------------------------------------------------------------------*/
/*  The following definitions are for possible cursor shapes               */
/*-------------------------------------------------------------------------*/
#define LINE    0                   /* a line at the bottom               */
#define BLOCK   1                   /* a full block                       */
#define BOT_BLK 2                   /* a block in the bottom half of the  */
                                    /* character box                      */
#define TOP_BLK 3                   /* a block in the top half of the     */
                                    /* character box                      */

/*-------------------------------------------------------------------------*/
/*  Definitions for border pointers                                        */
/*-------------------------------------------------------------------------*/
#define BDR_0P   &bdr_0             /*No border                           */
#define BDR_LNP  &bdr_ln            /*Single line border                  */
#define BDR_DLNP &bdr_dln           /*Double line border                  */
#define BDR_REVP &bdr_rev           /*Reverse border                      */
#define BDR_DOTP &bdr_dot           /*Dot border                          */
#ifdef UNIX
#define BDR_STARP &bdr_star         /*Star border                         */
#endif

/*-------------------------------------------------------------------------*/
/*  Different move types for internal low-level move function              */
/*-------------------------------------------------------------------------*/
#define VM_MOVE 0                   /*video to user-memory type of move   */
#define MV_MOVE 1                   /*user-memory to video type of move   */
#define VV_MOVE 2                   /*video to video type move            */
#define MM_MOVE 3                   /*user-memory to user-memory type move*/

/*-------------------------------------------------------------------------*/
/*  Definitions for use with dim_wn()                                      */
/*-------------------------------------------------------------------------*/
#define INSIDE 1                    /* def for wn.setsw, copy_wn, dim_wn  */
#define FULL 0                      /*        ditto                       */
```

wfc_defs.h (continued)

```
/*-------------------------------------------------------------------------*/
/*  Definitions for use with copy_wc()                                     */
/*-------------------------------------------------------------------------*/
#define FREAD "r"                       /*fopen() mode for read open       */
#define FWRITE "w"                      /*fopen() mode for write open      */
#define FAPPEND "a"                     /*fopen() mode for append open     */

/*-------------------------------------------------------------------------*/
/*  Definitions for use with boolean parameters                            */
/*-------------------------------------------------------------------------*/
#define YES 1                           /*for use with boolean parameters  */
#define NO 0                            /*          ditto                  */
#define ON 1                            /*          ditto                  */
#define OFF 0                           /*          ditto                  */
#define TRUE 1                          /*          ditto                  */
#define FALSE 0                         /*          ditto                  */

/*-------------------------------------------------------------------------*/
/*  Centers text within window - for use with v_plst()                     */
/*-------------------------------------------------------------------------*/
#define LEFT_TXT 0                        /*left justifies text in window  */
#define CENTER_TXT -1                     /*centers text in window         */
#define RIGHT_TXT -2                      /*right justifies text in window */

/*-------------------------------------------------------------------------*/
/*  Macro definitions                                                      */
/*-------------------------------------------------------------------------*/
#define min(a,b) ((a) <=(b) ? (a): (b))      /*minimum value function     */
#define max(a,b) ((a) >=(b) ? (a): (b))      /*maximum value function     */

#define col_qty(wnp) ((wnp)->ce - (wnp)->cb + 1)  /*number of col in window */
#define row_qty(wnp) ((wnp)->re - (wnp)->rb + 1)  /*number of rows in window*/

#define hi_byte(a)  (((a) >> 8) & 0x00ff)      /*returns high byte of word  */
#define lo_byte(a)  ((a) & 0x00ff)             /*returns low byte of word   */

#define c_att(fground,bground)  ((bground << 4) + fground)

#define s_tbfmsg(a)                (tbf_msg = (a))

#ifdef MSDOS
#define s_keyloop(p)               (_wfckifp = (p))
#endif
```

wfc_defs.h (continued)

```
/*-------------------------------------------------------------------------*/
/* Macros for setting members of the WINDOW structure                      */
/*-------------------------------------------------------------------------*/
#define sw_att(attrib, wnp)         ((wnp)->att = (attrib))
#define sw_bdratt(att, wnp)         ((wnp)->bdratt = (att))
#define sw_border(borderp, wnp)     ((wnp)->bdrp = (borderp))
#define sw_cleor(a, wnp)            ((wnp)->wrap = ((a) == ON ? (wnp)->wrap\
&~NO_CLEAR : (wnp)->wrap | NO_CLEAR))
#define sw_csadv(a, wnp)            ((wnp)->wrap = ((a) == ON ? (wnp)->wrap\
& ~NO_CS_ADV : (wnp)->wrap | NO_CS_ADV))
#define sw_latt(a, wnp)             ((wnp)->larray = (a))
#define sw_margin(l, r, wnp)        ((wnp)->l_mg = (l), (wnp)->r_mg = (r))
#define sw_mfile(a, wnp)            ((wnp)->frp = (a))
#define sw_name(name, wnp)          ((wnp)->wname = (name))
#define sw_plcsr(a, wnp)            ((wnp)->wrap = ((a) == ON ? (wnp)->wrap\
| PL_CSR : (wnp)->wrap & ~PL_CSR))
#define sw_popup(a, wnp)            ((wnp)->popup = (a))
#define sw_scroll(a, wnp)           ((wnp)->scr_q = (a))
#define sw_wwrap(a, wnp)            ((wnp)->wrap = ((a) == ON ? (wnp)->wrap\
| WRAP : (wnp)->wrap & ~WRAP))

/*-------------------------------------------------------------------------*/
/*  String images of maximum integer and long integer                      */
/*-------------------------------------------------------------------------*/
#define MAXLONGSTR "2147483647"         /* (2**31) - 1                     */
#define MAXINTSTR  "32767"              /* (2**15) - 1                     */
#define MAXEXP 3Ø7                      /* maximum floating point exponent */
```

/* window.h -- include file that defines global variables for Windows for C

********** Copyright 1985 by Vermont Creative Software **********

COMMENT

#includes an include file (att_glob.h) that specifies default logical attributes. To implement logical attributes, the parameter ATT_LOGICAL must be #defined equal to 1 here or in a prior include file.

Defines several standard borders: a null border, a dotted border, a line border, and a light (reverse) border.

A null border is indicated by a Ø (NULL) in ch_h (the second parameter in the BORDER list).

Includes wnØ, which is the basic 8Ø column screen without borders or word wrap and with scrolling with scr_q = 1.

Initial declaration of cl_att, attribute variable used to clear window area by unset_wn() and cls(). Set to NORMAL.

The definitions here assume the use of standard (not logical) attributes. When logical attributes are implemented (by #defining ATT_LOGICAL), _v_init() will change the standard attributes assigned to the border structures, wnØ, and cl_att to logical attributes. The logical attribute used is LNORMAL, except that cl_att and wnØ are set LDOS.

Initial values are set for global variables.

Reserves space for a permanent string buffer, v_sbuf[], used by video functions.

Reserves space for a global error-code variable, _wn_err. This may be set by routines that experience errors so that higher level routines can know the cause of the error.

CAUTION

Place in main program following "bios.h".

Do not use v_sbuf[]. This must be reserved for internal use by video routines.

*/

window.h (continued)

```
/*-------------------------------------------------------------------------*/
/*  Defines the default logical attributes                                 */
/*-------------------------------------------------------------------------*/
#include <att_glob.h>                   /*defines default logical attributes  */

#ifdef UNIX
/*-------------------------------------------------------------------------*/
/*  Include the global variables for the standard terminal interface       */
/*-------------------------------------------------------------------------*/
#include <ux_glob.h>
#endif

/*-------------------------------------------------------------------------*/
/*  Initialize the border structures                                       */
/*-------------------------------------------------------------------------*/
BORDER bdr_0 = {0};
BORDER bdr_rev = {REVERSE,32,32,32,32,32,32};

#ifdef MSDOS
BORDER bdr_dot = {NORMAL,178,178,178,178,178,178};
BORDER bdr_dln = {NORMAL,205,186,201,187,188,200};
BORDER bdr_ln = {NORMAL,196,179,218,191,217,192} ;
#endif

#ifdef UNIX
BORDER bdr_dot = {NORMAL, '.', ':', '.', '.', '.', '.'};
BORDER bdr_star = {NORMAL, '*', '*', '*', '*', '*', '*'};

/*-------------------------------------------------------------------------*/
/*  For bdr_dln and bdr_ln, the proper characters will be inserted into the */
/*  structures during initialization if specified in WFCTERMCAP.           */
/*-------------------------------------------------------------------------*/
BORDER bdr_dln = {BLK_GRAPH, ' ', ' ', ' ', ' ', ' ', ' '};
BORDER bdr_ln =  {BLK_GRAPH, ' ', ' ', ' ', ' ', ' ', ' '};
#endif

WINDOW wn0;                             /*initial values assigned during WFC  */
                                        /*initialization routine              */

/*The following are definitions for globally known variables.  They are     */
/*declared as extern variables in vextern.h, which is nested in bios.h      */
/*Initial values are assigned here to avoid serious errors; but the correct */
/*initial values are assigned by _v_init().                                 */
/*initial values assigned here are for IBM Monochrome Display Adapter,      */
/*with standard (not logical) attributes.                                   */

char cl_att = NORMAL;                   /* used by unset_wn() and cls() in    */
                                        /* clearing windows                   */
char v_mode = MONOMODE;                 /* current video mode                 */
```

Mono=7
Compaq=3

window.h (continued)

```
char tv_upd = 1;                    /* enable updating under TopView or   */
                                    /* UNIX.  For MSDOS systems, this will*/
                                    /* be reset to zero if TopView is not */
                                    /* present.  User can set to zero to  */
                                    /* to manually inhibit all video      */
                                    /* updating under TopView or UNIX.    */
char tbf_msg = 1;                   /* controls if the "top of file" and  */
                                    /* "bottom of file" messages are      */
                                    /* displayed by v_file().             */
char v_contig = 1;                  /* adjacent lines on the display are  */
                                    /* contiguous in video regen buffer   */
char v_sbuf[CO_QMAX * 2];           /* string buffer for video functions  */
int _wn_err;                        /* error code variable                */
int _vpstlen = VPSTMAXLEN;          /* maximum length of buffer associated*/
                                    /* with v_printf()                    */

#ifdef MSDOS                        mono  compaq
char v_retr = 0;                      0     1     /* set by v_init(), depending on mode */
char no_retr = 0;                     0     0     /* set to one to disable retrace      */
int _ibmega = 0;                    /* IBM EGA not present = 0;           */
int v_seg = MONO_SEG;               /* video regen buffer segment         */
int v_coq = CO_QMAX;                /* number of columns in screen display*/
int v_rwq = RW_QSCRN;               /* number of rows in screen display   */
int v_pbytes = BYTES_80;            /* number of bytes in vrb page        */
ADDR v_vrb;                         /* address of video regen buffer --   */
                                    /*      offset and segment            */
int _l_ptr = 0;                     /* = 1 if long pointers (large data)  */
int _d_seg = 0;                     /* data segment, set = 0 initially    */
                                    /* to indicate v_init() not done      */
PFI _wfckifp;                       /* pointer to keyboard loop function  */
#endif

#ifdef UNIX
int v_coq;                          /* number of columns in screen display*/
int v_rwq;                          /* number of rows in screen display   */
int v_pbytes;                       /* number of bytes in vrb page        */
char *v_vrb;                        /* pointer to virtual video buffer    */
int _wn_init = 0;                   /* initialization flag                */
#endif
```

```
/* att_glob.h --  contains default definitions and declarations of logical
                  attribute tables for use in Windows for C.
                  Nests in window.h.  The #defines for logical attributes are
                  in def_att.h, which is nested in bios.h

    ************* Copyright 1985 by Vermont Creative Software **************

IMPLEMENTING LOGICAL OR PHYSICAL ATTRIBUTES

    Logical attributes will be implemented if ATT_LOGIC is defined.  This is
    the default condition set in bios.h.

    To use physical attributes, #undefine ATT_LOGIC here or in an include file
    that precedes this one.  Alternatively, the #define ATT_LOGIC statement in
    bios.h could be commented out.  Generally, the include file for a specific
    application would set ATT_LOGIC.

THE LOGICAL-ATTRIBUTE TABLE

    The physical attributes associated with each logical attribute are
    specified in a logical-attribute table (two dimensional array),
    datt_tbl[][].  There is a row for each logical attribute; the number of
    rows in the table equals LATTQ.  The number of columns equals PATTQ, the
    number of physical attributes for each logical attribute.  LATTQ and PATTQ
    are #defined in def_att.h.

    The system, as supplied has two physical attributes for each logical
    attribute:  a monochrome attribute, and a color attribute; thus PATTQ is
    set to two in att_def.h.  You can add more physical attributes by
    increasing the value of PATTQ and adding more columns to the logical
    attribute table.

*/

#ifdef ATT_LOGIC                        /*LOGICAL ATTRIBUTES in use         */
int _lattsw = 1;
char latt[LATTQ];                       /*array of logical attributes       */

char  datt_tbl [LATTQ] [PATTQ] =
{
#ifdef MSDOS
   {NORMAL, c_att(WHITE, BLACK)},                                   /*LDOS     */
   {NORMAL, c_att(WHITE, BLUE)},                                    /*LNORMAL  */
   {REVERSE, c_att(BLUE, WHITE)},                                   /*LREVERSE */
   {NORMAL + HIGH_INT, c_att(WHITE + LIGHT, BLUE)},                 /*LHIGHLITE*/
   {NORMAL + HIGH_INT + BLINK, c_att(RED + BLINK, BLACK)},          /*LURGENT  */
   {NORMAL + HIGH_INT, c_att(BLUE, WHITE)},                         /*LHELP    */
   {REVERSE, c_att(RED, BLACK)},                                    /*LERROR   */
   {NORMAL + HIGH_INT, c_att(WHITE + LIGHT, BLUE)},                 /*MESSAGE  */
   {NORMAL + HIGH_INT, c_att(CYAN, BLUE)},                          /*LFIELDI  */
   {REVERSE, c_att(BLACK, CYAN)},                                   /*LFIELDA  */
   {REVERSE, c_att(BLUE, WHITE)},                                   /*LMARK    */
```

att_glob.h (continued)

```
   {0, c_att(BLUE, BLUE)},                                          /*LNODISPLA*/
   {REVERSE, c_att(BLACK, BLUE)},                                   /*LBLACK   */
   {REVERSE, c_att(BLUE, WHITE)},                                   /*LBLUE    */
   {NORMAL,  c_att(GREEN, BLUE)},                                   /*LGREEN   */
   {NORMAL, c_att(CYAN, BLUE)},                                     /*LCYAN    */
   {REVERSE, c_att(RED, BLUE)},                                     /*LRED     */
   {REVERSE, c_att(MAGENTA, BLUE)},                                 /*LMAGENTA */
   {NORMAL, c_att(BROWN, BLUE)},                                    /*LBROWN   */
   {NORMAL, c_att(WHITE, BLUE)},                                    /*LWHITE   */
   {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},                          /*RESERVED */
   {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},                          /*RESERVED */
   {0, 0}, {0, 0}                                                   /*RESERVED */
#endif

#ifdef UNIX
   {NORMAL, COLOR9},                                                /*LDOS     */
   {NORMAL, COLOR10},                                               /*LNORMAL  */
   {REVERSE, COLOR11},                                              /*LREVERSE */
   {HIGH_INT, COLOR12},                                             /*LHIGHLITE*/
   {BLINK, COLOR13},                                                /*LURGENT  */
   {HIGH_INT, COLOR11},                                             /*LHELP    */
   {REVERSE, COLOR14},                                              /*LERROR   */
   {HIGH_INT, COLOR12},                                             /*MESSAGE  */
   {HIGH_INT, COLOR15},                                             /*LFIELDI  */
   {REVERSE, COLOR16},                                              /*LFIELDA  */
   {REVERSE, COLOR11},                                              /*LMARK    */
   {INVISIBLE, INVISIBLE},                                          /*LNODISPLA*/
   {REVERSE, COLOR1},                                               /*LBLACK   */
   {REVERSE, COLOR2},                                               /*LBLUE    */
   {NORMAL, COLOR3},                                                /*LGREEN   */
   {NORMAL, COLOR4},                                                /*LCYAN    */
   {REVERSE, COLOR5},                                               /*LRED     */
   {REVERSE, COLOR6},                                               /*LMAGENTA */
   {NORMAL, COLOR7},                                                /*LBROWN   */
   {NORMAL, COLOR8},                                                /*LWHITE   */
   {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},                          /*RESERVED */
   {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},                          /*RESERVED */
   {0, 0}, {0, 0}                                                   /*RESERVED */
#endif
};
int _attrowq = LATTQ;

#else                                   /*PHYSICAL ATTRIBUTES in use       */
int _lattsw = 0;
char latt[1];                           /*array of logical attributes      */
char datt_tbl [1][PATTQ];
int _attrowq = 1;
#endif

int _attcolq = PATTQ;
```

This page intentionally left blank.

```
/* wfc_stru.h -- Windows for C structure declarations

    ************* Copyright 1985 by Vermont Creative Software **************

*/

typedef struct bord                     /*BORDER STRUCTURE                  */
{
    char batt;                          /*attribute of border characters    */
    char h_ch, v_ch;                    /*horizontal and vertical characters*/
    char c1, c2, c3, c4;                /*corner char, starting in upper left*/
                                        /*and continuing clockwise.         */
} BORDER, *BORDERPTR;


typedef struct file_line                /*ASCII FILE LINE STRUCTURE         */
{
    int line_len;                       /*length of a line in the file      */
    char *line_st;                      /*pointer to line string            */
}FLINE, *FLINEPTR;


typedef struct file_record              /*ASCII FILE RECORD STRUCTURE       */
{
    char *fn;                           /*filename                          */
    int ib;                             /*line-pointer array index of       */
                                        /*beginning line of file            */
    char *lbp;                          /*pointer to beginning line of file */
    char *lep;                          /*pointer to end line of file       */
    int c_q;                            /*max number of columns, equal to   */
                                        /*to maximum length line of file,   */
                                        /*including newline.                */
    int ln_q;                           /*number of lines in file, including*/
                                        /*EOF (NULL) line                   */
    int wfr;                            /*row of file to appear in wn.r = 0 */
    int wfc;                            /*col. of file to appear in wn.c = 0*/
    FLINEPTR *farray;                   /*pointer to a pointer to a FLINE   */
    int fmaxline;                       /*max number of lines allowed       */
    int ftabq;                          /*number of spaces per tab character*/
    int fmaxcol;                        /*maximum number of columns in file */
} FREC, *FRECPTR;


typedef struct wnd                      /*WINDOW STRUCTURE                  */
{
    int rb;                             /*top row of window                 */
    int re;                             /*bottom row of window              */
    int cb;                             /*left hand column of window        */
    int ce;                             /*right hand column of window       */
    int r;                              /*virtual cursor row-position       */
    int c;                              /*virtual cursor column-position    */
    char att;                           /*window video attribute            */
    char page;                          /*graphics-card alpha mode page #   */
    int  wrap;                          /*word-wrap switch                  */
    int  location;                      /*window location parameter         */
    int  scr_q;                         /*max number of lines to scroll     */
    int l_mg, r_mg;                     /*left and right spaces (margins) --*/
                                        /*to border                         */
    BORDERPTR bdrp;                     /*pointer to border struct          */
```

```
    char setsw;                     /*window dimensions: FULL = 0, INSIDE */
                                    /* = 1; initialize to 0               */
#ifdef MS
    char wn_dummy;                  /*dummy byte for alignment            */
#endif
#ifdef LATTICE
    char wn_dummy;                  /*dummy byte for alignment            */
#endif
    FRECPTR frp;                    /*pointer to file-record struct       */
    char *storp;                    /*pointer to window storage           */
    char *userp[2];                 /*2 pointers reserved for users       */
    char *wname;                    /*pointer to window name              */
    char *larray;                   /*pointer to logical attribute array  */
    char *pu_storp;                 /*pointer to pop-up window storage    */
    char bdratt;                    /*video attribute attribute of border */
    char popup;                     /*popup switch: 0 = no popup          */
                                    /*              1 = popup             */
/*  char *reserv3[4];                 four reserved pointers              */
/*  int reserv2[6];                   six reserved words, not now used    */
} WINDOW, *WINDOWPTR;


typedef struct krec                 /*MULTIPLE KEYSTROKE STRUCTURE        */
{
    int kv;                         /*key code value                      */
    int kq;                         /*quantity of identical codes found   */
    int kmax;                       /*limit on number of codes to be re-  */
} KEYR, *KEYRPTR;                   /*moved from buffer                   */


#ifdef MSDOS
typedef struct addr_struct          /*8086 ADDRESS STRUCTURE              */
{
    int off;                        /*address offset                      */
    int seg;                        /*address segment                     */
} ADDR, *ADDRPTR;
#endif


typedef struct vidioregs            /*8086 REGISTER STRUCTURE             */
{
    unsigned int ax, bx, cx, dx, si, di, es, ds;
} VIDIO, *VIDIOPTR;

#ifdef UNIX
typedef struct kdef                 /*KEYSTROKE DEFINITION STRUCTURE      */
{
    char *key_st;                   /*pointer to keystroke string         */
    int key_len;                    /*length of keystroke string          */
    int key_xlat;                   /*translated value of string          */
} KEYDEF, *KEYDEFPTR;
#endif
```

APPENDIX 2

WINDOWS FOR C -- LIBRARY FUNCTIONS

ALPHABETICAL LISTING OF LIBRARY FUNCTIONS

adj_cs(&wn) -- adjusts cs row and col values and checks limits

bell() -- sound the bell (beep) on the IBM PC

cls() -- clears screen with clear-screen attribute

cl_wn(&wn) -- clears windows

color_sc(bground) -- sets a color attribute for use in clearing the screen

color_wn(fground, bground, &wn) -- sets the foreground and background colors in a window

copy_wc(dimen, filename, dmode, &wn) -- copies the contents of a window image to a file

csr_hide() -- hides the screen cursor

csr_show() -- restores the screen cursor to its previous location

csr_type(type) -- controls size and style of screen cursor

c_att(fground, bground) -- returns the attribute value for the specified colors

defs_wn(&wn, rb, cb, rw_q, co_q, &bdr) -- alternate define window function, defines size

def_fr(&fr, fname, fmaxline, fmaxcol) -- assigns values to the FREC structure

def_wn(&wn, rb, re, cb, ce, l_mg, r_mg, &bdr) -- assigns initial values to the members of a window structure

dim_wn(dimen, &wn) -- adjusts the working dimensions of a window

di_file(&fr) -- read disk file into memory

di_st(fp, stp, q, brk_ch, tab_q) -- general purpose routine for disk input (di) to string (st)

dup_wn(&dwn, &swn) -- duplicates a window structure

errout(st1, st2) -- prints fatal error messages and exits

char *file_lnp(frow, &mfile) -- returns a pointer to the string associated with the i-th line in the file

free_file(&fr) -- frees FLINE structures associated with a file

free_mem(p) -- frees memory obtained from get_mem() or malloc()

(Continued)

ALPHABETICAL LISTING OF LIBRARY FUNCTIONS (continued)

char *get_mem(size) -- calls malloc() with error checking

init_wfc() -- initializes Windows for C system

ki() -- obtains key code from keyboard

ki_chk() -- checks buffer for key value; returns value if available.

ki_cum(&keyr) -- obtains identical codes from buffer

k_vcom(kv, kq, &wn) -- translates special key codes into video movement commands

void lower_st(st) -- converts each position in string to lowercase

menu2(&wn, qitems, itemlength, item_row, def_pos) -- pop-up menu display and selection routine

mode_col() -- obtains number of video columns in current mode

mod_wn(rb, cb, rowq, colq, &wn) -- modifies coordinates of a window

mv_cs(rw, co, &wn) -- changes value of virtual cursor; does not move screen cursor

mv_csr(rw, co, &wn) -- changes value of virtual cursor and moves screen cursor to virtual cursor location

mv_rws(nlines, dir, &wn) -- scrolls designated window by nlines in given direction

mv_wi(rw, co, &wn) -- moves a window image to a new location on the screen

pl_csr(&wn) -- places screen cursor at window virtual cursor

pl_mfwn(frow, fcol, &wn) -- places window origin at specified location in file

pl_wn(rw, co, &wn) -- places window at specified location

prt_wn(&wn) -- copies contents of a window to a printer

rd_csr(&row, &col, page) -- reads the current location of the screen cursor

rd_mode() -- returns current video mode of the IBM PC

repl_wi(&wn) -- replaces onto the screen a saved window image

sav_wi(&wn) -- saves a window image in memory

scrl_file(nrows, dir, beg_row, &mfile) -- scrolls a memory file

(Continued)

ALPHABETICAL LISTING OF LIBRARY FUNCTIONS (continued)

set_wn(&wn) -- sets borders and margins on designated window

size_wn(dimen, part, &wn) -- calculates size of video string needed for window saves

char *skip_wh(st) -- skips leading whitespace in string

sti_file(st, frow, &mfile) -- loads strings into WFC data structure used for files

char *stblank(len) -- allocates memory for string and initializes to blanks

char *strcpyp(dest, src) -- copies source string to destination string

void strip_wh(st) -- strips trailing whitespace from string

sw_att(att, &wn) -- sets wn.att to the specified attribute

sw_bdratt(att, &wn) -- sets wn.bdratt to the specified attribute

sw_border(&bdr, &wn) -- sets wn.bdrp to the specified border

sw_cleor(state, &wn) -- sets the clear to end of row bit-switch in wn.wrap

sw_csadv(state, &wn) -- sets the cursor advance bit-switch in wn.wrap

sw_latt(att_arry, &wn) -- sets wn.larray to point to the specified logical attribute array

sw_margin(l_mg, r_mg, &wn) -- sets the wn.l_mg and wn.r_mg to the specified values

sw_mfile(&mfile, &wn) -- sets wn.frp to point to the specified memory file structure

sw_name(name, &wn) -- set wn.wname to the specified name

sw_plcsr(state, &wn) -- sets the place cursor bit-switch in wn.wrap

sw_popup(state, &wn) -- set wn.popup to the specified state

sw_scroll(state, &wn) -- sets wn.scr_q to the specified state

sw_wwrap(state, &wn) -- sets the wrap bit-switch in wn.wrap

s_latt(col, rowq, colq, att_tbl, att_arry) -- copies appropriate column from the attribute table into the logical attribute array

unsav_wi(&wn) -- replaces a saved window image, frees memory

(Continued)

ALPHABETICAL LISTING OF LIBRARY FUNCTIONS (continued)

unset_wn(&wn) -- removes a window from the screen

void upper_st(st) -- converts each position in the string to uppercase

u_init() -- allows initialization of globally known variables

vid_bdr(color) -- sets color border around screen

vid_int(&vri, &vro) -- General video interrupt (INT 10H) routine

vid_mode(mode) -- sets the video mode, switches display adapters

vo_att(&wn) -- reads an attribute from the window

vo_ch(&wn) -- reads a character from the window

void vs_file(exit_key, wnp) -- view and scroll a file

v_att(att, &wn) -- writes the specified attribute to the current virtual cursor position

v_axes(r_origin, c_origin, height, width, &wn) -- draws axes for graphs

v_bar(row_size, col_size, r_begin, c_begin, ch, attrib, &wn, &bdr) -- draws a horizontal or vertical bar

v_border(&wn, &bdr) -- draws specified border on specified window

v_ch(ch, &wn) -- writes specified character to window at the virtual cursor position

v_co(ch, q, &wn) -- puts column of attribute-char's to a window

v_file(&wn) -- puts a file to a window

v_fst(st, &wn) -- puts full character string to a window

v_mov(vst, &wn, part, direction) -- moves info between video and a "video string"

v_mova(st, &wn, part, direction) -- moves info between video and a standard ASCII string

v_natt(att, part, &wn) -- gives a new attribute to a section of a window

v_plst(rw, co, st, &wn) -- writes string to window starting at specified location

v_printf(&wn, fmt, args....) -- performs formatted output to window

(Continued)

ALPHABETICAL LISTING OF LIBRARY FUNCTIONS (continued)

v_qch(ch, q, &wn) -- writes q character-attribute pairs to window

v_rw(ch, q, &wn) -- puts a row of attribute-char's to a video window

char *v_st(st, &wn) -- puts character string to video window

void v_st_nop(st, q, &wn) -- variation of v_st_rw( ) for use by v_file( )

char *v_st_rw(st, q, &wn) -- puts character string to a row of a video window

v_tv(r0, r1, c0, c1, &wn) -- issues a request to TopView to update video display

LIBRARY FUNCTIONS BY CATEGORY OF USE

BIOS Video Routines, Access to

vid_int(&vri, &vro) -- General video interrupt (INT 10H) Routine

v_tv(r0, r1, c0, c1, &wn) -- issues a request to TopView to update video display

Bit-Switch Utilities

sw_cleor(state, &wn) -- sets the clear to end of row bit-switch in wn.wrap

sw_csadv(state, &wn) -- sets the virtual cursor advance bit-switch in wn.wrap

sw_plcsr(state, &wn) -- sets the place cursor bit-switch in wn.wrap

sw_wwrap(state, &wn) -- sets the word wrap bit-switch in wn.wrap

Color Window Control

color_sc(bground) -- sets a color attribute for use in clearing the screen

color_wn(fground, bground, &wn) -- sets the foreground and background colors in a window

c_att(fground, bground) -- returns the attribute value for the specified fground and bground colors

rd_mode() -- returns current video mode of the IBM PC

vid_bdr(color) -- sets color border around screen

vid_mode(mode) -- sets the video mode, switches display adapters

Cursor Control and Scrolling

adj_cs(&wn) -- adjusts cs row and col values and checks limits

csr_hide() -- hides the screen cursor

csr_show() -- restores the screen cursor to its previous location

csr_type(type) -- controls size and style of cursor

mv_cs(rw, co, &wn) -- changes value of virtual cursor; does not move screen cursor

mv_csr(rw, co, &wn) -- changes value of virtual cursor and moves screen cursor to new location

(Continued)

LIBRARY FUNCTIONS BY CATEGORY OF USE (continued)

mv_rws(nlines, dir, &wn) -- scrolls designated window by nlines in given direction

pl_csr(&wn) -- places screen cursor at location of window virtual cursor

rd_csr(&row, &col, page) -- reads the current location of the screen cursor

Defining and Adjusting Values of Windows

def_wn(&wn, rb, re, cb, ce, l_mg, r_mg, &bdr) -- assigns initial values to the members of a window structure

defs_wn(&wn, rb, cb, rw_q, co_q, &bdr) -- alternate define window function, defines size

dup_wn(&dwn, &swn) -- duplicates a window structure

dim_wn(dimen, &wn) -- adjusts the working dimensions of a window

mod_wn(rb, cb, rowq, colq, &wn) -- modifies coordinates of a window

Error Messages

errout(st1, st2) -- writes error messages and calls exit(1)

File Viewing and Management

def_fr(&fr, fname, fmaxline, fmaxcol) -- assigns values to the FREC structure

di_file(&fr) -- read disk file into memory

di_st(fp, stp, q, brk_ch, tab_q) -- general purpose routine for disk input (di) to string (st)

char *file_lnp(frow, &mfile) -- returns a pointer to the string associated with the i-th line in the file

free_file(&fr) -- frees FLINE structures associated with a file

k_vcom(kv, kq, &wn) -- translates special key codes into video movement commands

pl_mfwn(frow, fcol, &wn) -- places window origin at specified location in file

scrl_file(nrows, dir, beg_row, &mfile) -- scrolls a memory file

sti_file(st, frow, &mfile) -- loads strings into WFC data structure used for files

(Continued)

LIBRARY FUNCTIONS BY CATEGORY OF USE (continued)

vs_file(exit_key, wnp) -- view and scroll a file

v_file(&wn) -- puts a file to a window

Initialization

init_wfc() -- initializes the Windows for C system

u_init() -- allows initialization of globally known variables

Keyboard Functions

ki() -- obtains key code from keyboard

ki_chk() -- checks buffer for key value; returns value if available.

ki_cum(&keyr) -- obtains identical codes from buffer

Graphing Functions

v_axes(r_origin, c_origin, height, width, &wn) -- draws axes for graphs

v_bar(row_size, col_size, r_begin, c_begin, ch, attrib, &wn, &bdr) -- draws a horizontal or vertical bar

Logical Attribute Support Functions

s_latt(col, rowq, colq, att_tbl, att_arry) -- copies appropriate column from attribute table into logical attribute array

Memory Management

free_mem(p) -- calls free() with error checking

char *get_mem(size) -- calls malloc() with error checking

Menu Management

menu2(&wn, qitems, itemlength, item_row, def_pos) -- a pop-up menu display and selection routine

Reading Information from Screen

vo_att(&wn) -- reads attribute from window at current virtual cursor position

vo_ch(&wn) -- reads character from window at current virtual cursor position

v_mova(st, &wn, part, direction) -- reads specified section from window to standard ASCII string

(Continued)

LIBRARY FUNCTIONS BY CATEGORY OF USE (continued)

Set Window Structure Member Functions

sw_att(att, &wn) -- sets wn.att to the specified attribute

sw_bdratt(att, &wn) -- sets wn.bdratt to the specified attribute

sw_border(&bdr, &wn) -- sets wn.bdrp to the specified border

sw_latt(att_arry, &wn) -- sets wn.larray to point to the specified logical attribute array

sw_margin(l_mg, r_mg, &wn) -- sets the wn.l_mg and wn.r_mg to the specified values

sw_mfile(&mfile, &wn) -- sets wn.frp to point to the specified memory file structure

sw_name(name, &wn) -- set wn.wname to the specified name

sw_popup(state, &wn) -- set wn.popup to the specified state

sw_scroll(state, &wn) -- sets wn.scr_q to the specified state

Sound

bell() -- sound the bell (beep) on the IBM PC

String Utility Functions

void lower_st(st) -- converts each position in string to lowercase

char *skip_wh(st) -- skips leading whitespace in string

char *stblank(st) -- allocates memory for string and initializes to blanks

char *strcpyp(dest, src) -- copies source string to destination string

void strip_wh(st) -- strips trailing whitespace from string

void upper_st(st) -- converts each position in the string to uppercase

Window Screen Images: Moving, Changing, Saving, and Restoring

copy_wc(dimen, filename, dmode, &wn) -- copies the character contents of a window image to a file

mv_wi(rw, co, &wn) -- moves a window image to a new location on the screen

prt_wn(&wn) -- copies the character contents of a window to a printer

(Continued)

LIBRARY FUNCTIONS BY CATEGORY OF USE (continued)

repl_wi(&wn) -- replaces onto the screen a saved window image

sav_wi(&wn) -- saves a window image in memory

size_wn(dimen, part, &wn) -- calculates size of video string needed for window saves

unsav_wi(&wn) - places onto the screen a saved window image, frees memory

v_natt(att, part, &wn) - gives a new attribute to a section of a window

v_mov(vst, &wn, part, direction) -- moves info between video and a "video string"

v_mova(st, &wn, part, direction) -- moves info between video and standard ASCII string

Window Management

cls( ) -- clears screen with screen background attribute

cl_wn(&wn) -- clears windows

mode_col( ) -- obtains number of video columns in current mode

pl_wn(rw, co, &wn) -- places window at specified location

set_wn(&wn) -- sets borders and margins on designated window

unset_wn(&wn) -- removes windows from the screen, resets dimensions

v_border(&wn, &bdr) -- draws border on designated window

Writing Text to Video

v_att(att, &wn) -- writes specified video attribute to window at current virtual cursor position

v_ch(ch, &wn) -- writes single character to window at current virtual cursor position

v_co(ch, q, &wn) -- puts column of attribute-char's to a window

v_fst(st, &wn) -- puts full character string to a window

v_plst(rw, co, st, &wn) -- writes string to window beginning at specified row and column

v_printf(&wn, fmt, args...) -- performs formatted output to window

(Continued)

LIBRARY FUNCTIONS BY CATEGORY OF USE (continued)

v_qch(ch, q, &wn) -- writes q character-attribute pairs to window

v_rw(ch, q, &wn) -- puts a row of attribute-char's to a video window

char *v_st(st, &wn) -- puts character string to video window

void v_st_nop(st, q, &wn) -- puts character string to a row of a video window, no options

char *v_st_rw(st, q, &wn) -- puts character string to a row of a video window

NAME

adj_cs.c -- adjusts cs row and col values and checks limits

DATE: October 23, 1985

USAGE

To check and adjust the position of the virtual cursor in a window. The virtual cursor is the position defined by the values of wn.r and wn.c contained in the structure WINDOW wn.

To check whether cs is beyond the window bottom, that is, if the window is full.

To adjust wn.r and wn.c after incrementing wn.c, to ensure that cs is kept within the window boundaries.

FUNCTION

If wn.c is greater than wn.ce, wn.r is incremented and wn.c is set to zero.

If cs is beyond the lower right corner of a window, wn.r is set to the first row beyond the window and wn.c is set to zero.

If wn.r or wn.c is less than zero, the negative value is set to zero.

CALL

```
adj_cs(&wn)
WINDOW wn;                    window structure
```

RETURNS

= Ø if beyond window lower right corner;

= 1 otherwise

CAUTIONS

None

NAME

bell.c -- sounds the bell on the IBM PC

DATE: January 30, 1986

USAGE

To sound the bell (generally as a warning).

FUNCTION

For MSDOS, calls BIOS interrupt 10H, function 14 (via vid_int()) to sound the bell (beep).

For UNIX, sends the terminal command string that rings the bell.

CALL

void bell()

RETURNS

None

CAUTIONS

None

NAME

cls.c -- clear screen

DATE: January 30, 1986

USAGE

To clear the entire video screen.

FUNCTION

MSDOS Systems:

The screen is cleared with spaces with the attribute cl_att. Variable cl_att is a globally known variable defined in window.h. It is assigned an initial value of NORMAL if physical attributes are used and LDOS if logical attributes are used. These values can be changed within the program.

UNIX Systems:

If a clear screen escape sequence has been defined, it will be used to clear the screen; otherwise the screen will be filled with blanks. The escape sequence approach is much faster and is the preferred method.

If the escape sequence is used to clear the screen and cl_att is not NORMAL, the current video attribute for the terminal is set to cl_att. The escape sequence is then sent to the terminal to clear the screen.

The virtual video buffer is cleared and with the cl_att attribute.

Variable cl_att is a globally known variable defined in window.h. It is assigned an initial value of NORMAL if physical attributes are used and LDOS if logical attributes are used. These values can be changed within the program.

CALL

```
void cls();
```

RETURNS

None

CAUTIONS

None

NAME

cl_wn.c -- clears windows

DATE: October 23, 1985

USAGE

Clear window and set background attribute to wn.att.

FUNCTION

Fills window with spaces of attribute wn.att, using v_qch().

The virtual cursor is set to Ø,Ø.

CALL

```
void cl_wn(&wn)
WINDOW wn;                          window structure
```

RETURNS

None

CAUTIONS

None

NAME

color_sc.c -- sets a color attribute for use in clearing the screen

DATE: January 3Ø, 1986

USAGE

When using physical attributes, to set the background attribute used by functions that clear to the screen background (cls( ) and unset_wn( )).

FUNCTION

Assigns a color value to the public attribute byte (cl_att) according to codes used for colors. For a definition of the colors, see color_wn( ).

Variable cl_att is the attribute byte used by cls( ) and unset_wn( ) when clearing the screen area with spaces.

Variable cl_att is defined in window.h. Because it is defined externally before main( ), it is available for use or modification in all subroutines.

The initial value of cl_att is NORMAL (BLACK background and WHITE foreground). Generally, this value should be restored by direct assignment before returning from a color to a black and white mode.

A WHITE background assigned by color_sc( ) will result in the clearing to REVERSE in non_color modes. This same result can be obtained by direct assignment of REVERSE to cl_att.

MSDOS Systems:

Note: This function only assigns a value to the background 3 bits of cl_att. The BLINK bit is turned off. The foreground color is set to WHITE (Ø). A color value can be directly added to cl_att after calling color_sc( ) to provide a non-WHITE foreground color. The foreground is never used by cls( ) and unset_wn( ), but you may wish to use it for another purpose in your own functions.

UNIX Systems:

Invalid attribute values are mapped to NORMAL.

CALL

```
void color_sc(bground)
char bground;                    background color value
```

RETURNS

None

CAUTIONS

Use only when physical attributes are implemented.

For UNIX systems, invalid attributes are mapped to NORMAL.

NAME

color_wn.c -- sets the foreground and background colors in a window.

DATE: January 30, 1986

USAGE

When using physical attributes, to set the colors displayed in a window when using the Color/Graphics Adapter or Enhanced Graphics Adapter under MSDOS or color terminals under UNIX.

FUNCTION

Assigns values to the attribute byte (wn.att) in a window structure according to codes used for colors on the color adapters or terminal.

MSDOS Systems:

To simplify use, the numeric codes are #defined into color words in computer.h (which is nested within bios.h). Eight basic color values are #defined:

BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, WHITE

Adding the #defined value LIGHT to the basic colors yields the second eight of the 16 colors available as foreground colors in the text modes, e.g. specifying LIGHT + RED as the foreground parameter will provide light red characters. For convenience, LIGHT + BROWN is #defined as YELLOW (see the caution below on use of LIGHT colors). On some monitors BROWN will also show up as yellow.

The LIGHT colors are not implemented for background colors when blink is enabled (because the color values would then overlap the attribute bit that creates blinking).

When blink is enabled adding BLINK to the foreground parameter value will cause characters to blink.

UNIX Systems:

To simplify use, the numeric codes are #defined into color words in terminal.h (which is nested within bios.h). Generic colors are #defined for 16 colors:

COLOR1 COLOR2 ... COLOR16

These definitions refer to the specific colors (foreground/background combinations) that were loaded in from the WFCTERMCAP file. There is no limit on the number of color combinations that can be read in from the WFCTERMCAP file. However, only COLOR1 - COLOR16 are #defined in terminal.h. If you wish to use more than 16 color combinations, you should add the appropriate definitions. The global array (va_cmd[][]) must be large enough to hold the desired number of video attribute commands.

(Continued)

color_wn (continued)

For convenience, more specific definitions may be made to reference the appropriate colors. For example, suppose that the first color (scl0, rcl0) defined in WFCTERMCAP is blue on a black background. You could make the following definition

```
#define BLUE COLOR1
```

You could then use BLUE when referring to this color combination.

For compatibility with the PC/MSDOS version of Windows for C, there are foreground and background arguments in the call. In the Unix/Xenix version, wn.att is set the foreground argument. The background argument is ignored.

CALL

```
void color_wn(fground, bground, &wn)
char fground;                    foreground color value
char bground;                    background color value
WINDOW wn;                       window structure
```

RETURNS

None

CAUTIONS

Use only when physical attributes are being implemented.

NAME

copy_wc.c -- copy character contents of a window image to a file

DATE: October 23, 1985

USAGE

To copy character information only (not attributes) from a window to a file.

Copies contents within the working dimensions of the window. If the dimensions are set to INSIDE, the border will not be copied. To copy the full window including the border, the dimensions need to be FULL. If necessary, use dim_wn( ) to alter the working dimensions.

FUNCTION

The "dmode" parameter specifies whether the file is to be opened in the FWRITE mode, which erases contents of the file specified, or FAPPEND mode, which appends the copied information to existing information in the file. If the specified file does not exist, it is created. FWRITE and FAPPEND are #defined in wfc_defs.h

Whether the full window or only window contents are copied is specified by the value of the "dimen" parameter. The values FULL (0) and INSIDE (1) are #defined in wfc_defs.h.

Allocates sufficient memory (via get_mem( )) to store one row of the window in a video string and calls v_mov( ) (ROW,OUT) to fill the string.

Copies the characters in the video string to a specified file using putc( ). Appends a newline to the end of each window row copied to the file.

CALL

```
copy_wc(dimen, filename, dmode, &wn)
char dimen;                    INSIDE or FULL window copy
char *filename;                name of the file to copy to
char *dmode;                   mode of disk operation
WINDOW wn;                     window structure
```

RETURNS

= 1 if the copy is successful.

= 0 if copy unsuccessful because file can not be opened or there is insufficient memory for the video-string buffer; check _wn_err code for more information.

= -1 if dimen specified was not equal to FULL (0) or INSIDE (1)

CAUTIONS FOR PROGRAMMING

Uses "stdio.h".

NAME

csr_hide( ) -- hides the screen cursor
csr_show( ) -- restores the screen cursor to its previous location

DATE: January 30, 1985

USAGE

To hide and restore the screen cursor.

FUNCTION

The present location of the screen cursor is read, via rd_csr( ), and stored. A call to csr_show( ) will restore the cursor to its previous location.

MSDOS Systems:

The cursor is moved off the screen so that it does not show.

UNIX Systems:

The cursor is turned off (if supported) by csr_hide( ) and turned on by csr_show( ).

CALL

```
void csr_hide( );
void csr_show( );
```

RETURNS

None

CAUTIONS

MSDOS Systems:

Do not make two calls to csr_hide( ) without an intervening call to csr_show( ). If you do, the stored location of the cursor will be offscreen. The next call to csr_show( ) will simply leave the cursor offscreen. If this is a possibility, use pl_csr( ) or mv_csr( ) to restore the screen cursor to a known location.

If the Color/Graphics Adapter is in use, it is possible to have more than one display page of information stored in the video buffer. Only one of these can be active and show on the screen. The default page is page 0. This function assumes that page 0 is the active page; thus it stores the location of the cursor in page 0, and csr_show will restore the cursor to this location.

UNIX Systems:

If the terminal does not support turning the cursor on/off, nothing will happen.

NAME

csr_type.c -- controls size and style of screen cursor

DATE: January 30, 1986

USAGE

To change the size and style of the screen cursor.

FUNCTION

MSDOS Systems:

Calls vid_int() to set the size of the cursor via PC INT 10H, function 1.

Provides for four cursor types. The cursor type is specified in the function call by the following parameter values #defined in wfc_defs.h:

LINE	(0)	a line at the bottom
BLOCK	(1)	a full block
BOT_BLK	(2)	a block in the bottom half of the character box
TOP_BLK	(3)	a block in the top half of the character box

UNIX Systems:

This is a dummy function in the Unix version of Windows for C. It is provided for compatibility with the MSDOS version.

CALL

```
void csr_type(type)
int type;                          type parameter
```

RETURNS

None

CAUTIONS

For UNIX systems, this function performs no action.

NAME

c_att.c -- returns attribute value for specified colors

DATE: October 23, 1985

USAGE

To set the foreground and background sections of an attribute byte. For a definition of the colors, see color_wn().

FUNCTION

Shifts the background value left by 4 bits and adds the foreground value.

CALL

```
c_att(fground, bground)
char fground;                    foreground color value
char bground;                    background color value
```

RETURNS

= attribute value for specified colors

CAUTIONS

Implemented as a macro. Be aware of side effects.

NAME

defs_wn.c -- define-window function, defines size

DATE: October 23, 1985

USAGE

To define a window in a standard manner. This alternate to def_wn( ) defines the length and width of a window rather than the beginning and ending row and column numbers.

FUNCTION

The beginning coordinates, the size of the window and the border are specified in the call. The margins are set to 1 by this function.

The minimum value allowed for wn.rb and wn.cb is Ø.

The maximum value allowed for wn.re and wn.cb are the last row and column on the screen. If the window size specified in the call would make the window lie outside of the screen, the origin will be as specified, but the size reduced to make it lie within the screen.

If logical attributes are being used (_lattsw == ON), the window and border attributes are set to LNORMAL, else they are set to NORMAL.

Elements of the window not specified in the call are set within the function as follows:

```
int wn.c = Ø;
int wn.r = Ø;
char wn.att = NORMAL or LNORMAL;
char wn.page = Ø;
int wn.wrap = WRAP;
int wn.location = Ø;
char wn.scr_q = 1;
int wn.l_mg = 1;
int wn.r_mg = 1;
char wn.setsw = Ø;
FRECPTR wn.frp = NULLP;
char *wn.storp = NULLP;
char *wn.userp[Ø] = NULLP;
char *wn.userp[1] = NULLP;
char *wname = NULLP;
char *larray = NULLP;
char *pu_storp = NULLP;
char bdratt = NORMAL or LNORMAL;
char popup = NO;
```

See def_wn( ) for further discussion.

(Continued)

defs_wn (continued)

CALL

```
void defs_wn(&wn, rb, cb, rw_q, co_q, &bdr)
int rb;                          See WINDOW typedef in wfc_stru.h for
int cb;                          definitions of these variables.
int rw_q;
int co_q;
BORDER bdr;
```

RETURNS

None

RELATED FUNCTIONS

def_wn( )

CAUTIONS

The WINDOW structure must be declared before the call. This function only sets the element values, it does not declare the window.

NAME

def_fr.c -- initializes a memory file

DATE: October 23, 1985

USAGE

To initialize a memory file by assigning initial values to a memory-file record structure (an FREC structure) and allocating an array to hold information on each line of the file. Members of the structure not specified in the call are assigned default values.

FUNCTION

Before calling this function, a memory file record (mfile) of type FREC must be declared. A pointer to this file record is passed as an argument. Additional arguments specify the file name, the maximum number of lines in the file, and the maximum number of columns in a single line. These values are assigned to the memory file record as follows:

mfile.fn	filename, including drive (and path, if supported by compiler and DOS and file not in default directory)
mfile.fmaxlines	the number of lines in the memory file (not including a position for an end of file marker)
mfile.fmaxcol	the maximum number of columns allowed in a file line

Storage is allocated for a memory file array, which is an array of pointers to FLINE structures. FLINE structures are used to hold information on each line in the memory file. The size of the memory file array equals the specified maximum number of lines in the memory file, plus one additional space for an end-of-file marker. All elements of the array are initialized to NULLP. A pointer to this array is assigned to the FREC member:

mfile.farray	pointer to an allocated array of FLINEPTRS

Structure members not specified in the call are assigned the following default values:

```
mfile.c_q = 0;          will be set by di_file or sti_file
mfile.ln_q = 0;         will be set by di_file or sti_file
mfile.wfr = 0;          origin of the window in file
mfile.wfc = 0;          origin of the window in file
mfile.ftabq = 8;        tab spacing, for di_file()
mfile.ib = 0;           not currently used
mfile.lbp = NULLP;      not currently used
mfile.lep = NULLP;      not currently used
```

If values other than those assigned internally are desired, assign them after calling def_fr.

(Continued)

def_fr (continued)

CALL

```
def_fr(&mfile, fname, fmaxline, fmaxcol)
FREC mfile                      FREC structure to be initialized
char *fname                     pointer to filename string
int  fmaxline                   maximum number of lines in file
int  fmaxcol;                   maximum number of columns in file
```

RETURNS

= Ø if unable to allocate storage for array of FLINE pointers

= 1 otherwise

RELATED FUNCTIONS

di_file()
sti_file()
free_file()
v_file()
vs_file()

CAUTIONS

The FREC structure must be declared before the call. This function only sets the member values, it does not declare the frec.

NAME

def_wn.c -- assigns values to WINDOW-structure elements

DATE: October 23, 1985

USAGE

To define a window in a standard manner. After declaring a structure, this function can be used to simplify assigning initial values to the structure.

FUNCTION

Dimensions, margins, and the border are specified in the call. Remaining elements of the WINDOW structure are assigned values internally.

Pointers to the standard borders are #defined in wfc_defs.h The borders and their definitions are:

BORDER	#define bdrp	BORDER STYLE
bdr_Ø	BDR_ØP	No border
bdr_ln	BDR_LNP	Single line border
bdr_dln	BDR_DLNP	Double line border
bdr_rev	BDR_REVP	Reverse border
bdr_dot	BDR_DOTP	Dot border

If values other than those assigned internally are desired, assign them after calling def_wn.

The minimum value allowed for wn.rb and wn.cb is Ø.

If logical attributes are being used (_lattsw == ON), the window and border attributes are set to LNORMAL, else they are set to NORMAL.

The maximum value allowed for wn.re and wn.cb are the last row and column of the current screen-size.

(Continued)

def_wn (continued)

Elements of the window not specified in the call are set within the function as follows:

```
int wn.c = 0;
int wn.r = 0;
char wn.att = NORMAL or LNORMAL;
char wn.page = 0;
int wn.wrap = WRAP;
int wn.location = 0;
char wn.scr_q = 1;
char wn.setsw = 0;
FRECPTR wn.frp = NULLP;
char *wn.storp = NULLP;
char *wn.userp[0] = NULLP;
char *wn.userp[1] = NULLP;
char *wname = NULLP;
char *larray = NULLP;
char *pu_storp = NULLP;
char bdratt = NORMAL or LNORMAL;
char popup = NO;
```

CALL

```
void def_wn(&wn, rb, re, cb, ce, l_mg, r_mg, &bdr)
int rb;                          ¦ See WINDOW typedef in bios.h for
int cb;                          ¦ definitions of these variables.
int re;                          ¦
int ce;                          ¦
int l_mg;                        ¦
int r_mg;                        ¦
BORDER bdr;                      BORDER structure
```

RETURNS

None

RELATED FUNCTIONS

defs_wn()

CAUTIONS

The WINDOW structure must be declared before the call. This function only sets the element values, it does not declare the window.

NAME

dim_wn.c -- adjusts the working dimensions of a window

DATE: October 23, 1985

USAGE

To adjust the "working dimensions" of a window to either the full dimensions including the border or to the inside dimensions, which are smaller than the full dimensions by the amount of the border and margins.

Working dimensions are defined by the values of wn.rb, wn.re, wn.cb, and wn.ce (the beginning and ending row and column values of the wn structure). When a window is first initialized, these members reflect the full dimensions of the window, to the outside of the borders. After a window is "set," using set_wn(), the values of these members are reduced by the width of the border and the margins. The inside of the window now represents the working dimensions.

Full dimensions are needed for routines that affect an entire window such as sav_wi(), whereas string output functions generally expect inside dimensions.

FUNCTION

A parameter in the call specifies whether the working dimensions of a window are to be set to the full or inside dimensions of a window. The values FULL (0) and INSIDE (1) are #defined in wfc_defs.h.

The value of wn.setsw is checked to determine the present status of the window dimensions, and the values of wn.rb, wn.re, wn.cb, and wn.ce are adjusted appropriately.

The value of wn.setsw is set to reflect the status of the working dimensions of the window: 0 for FULL and 1 for INSIDE.

CALL

```
dim_wn(dimen, &wn)
char dimen                      FULL (0) or INSIDE (1)
WINDOW wn;                      window structure
```

RETURNS

= -1 if dimen is not equal to FULL (0) or INSIDE (1)

= 1 otherwise

CAUTIONS

Does not perform consistency checks on the window dimensions. Use set_wn() initially to perform this check.

NAME

di_file.c -- to transfer an ASCII file from disk into a memory file

DATE: October 23, 1985

USAGE

To read an ASCII file from disk and place file contents in a memory file.

FUNCTION

This function draws on information set in a passed FREC structure, mfile, to perform the file operations. The FREC must be initialized by a call to def_fr( ) prior to calling this function. Function def_fr( ) will allocate and initialize a memory file array, which is an array of pointers to FLINE structures.

The memory file consists of FLINE structures. Function di_file( ) allocates a separate FLINE structure for each line of the ASCII file that is read into the memory file and place a pointer to this structure in the memory file array.

The structure FLINE fline contains two members:

fline.line_len is an integer and holds the length of the file line (excluding the terminal newline and null).

fline.line_st is a character pointer and points to the location where the file line-string is stored.

Function di_st( ) is called to read a file-line string terminated by newline into a string buffer. Tab expansion is performed. Function di_file( ) allocates the memory necessary for the FLINE structure and the file line string and then fills the FLINE structure with the appropriate information. A pointer to the FLINE structure is then placed in the memory file array and the array is indexed to prepare for the next entry.

The above functions are continued until end-of-file is reached or the limit on number of lines (mfile.fmaxlines) is reached.

If EOF is reached or the limit on number of lines is exceeded, a NULLP is placed in the last position in the array of FLINEPTR's.

The member mfile.ln_q is set to the number of lines read into the memory file. Member mfile.c_q is set to the length of the longest line (not including the terminal newline null on each file line). The values of mfile.c_q and mfile.ln_q are used by the functions (v_file( ), k_vcom( ), and vs_file( )) that provide capability for displaying and scrolling through a memory file.

Source code for this function is provided on the system diskette.

(Continued)

di_file (continued)

CALL

```
di_file(&mfile)
FREC mfile;                          memory file record
```

RETURNS

= 1 if the copy is successful

= Ø if error, check _wn_err for more information

ERROR HANDLING

The global variable _wn_err is set to different error codes to indicate that cause of an error return:

_wn_err =	ERR_OPEN	unable to open file
=	READERR	error reading file
=	MEMLACK	insufficient memory to copy file
=	FILETOOBIG	number of lines in file exceeds mfile.fmaxline
=	ERR_CLOSE	unable to close file
=	BADHEAP	error trying to free temporary buffer

You might wish to proceed when an error is returned (for example if the ASCII file contained more lines than would fit in the memory file), therefore no memory that is allocated by this function is released prior to an error return. Call free_file() to release the memory allocated by this function if you do not want to proceed.

RELATED FUNCTIONS

def_fr()
free_file()
sti_file()
v_file()
vs_file()
scrl_file()

CAUTIONS

Before using this function, the memory file must be initialized by calling def_fr().

NAME

di_st.c -- general purpose routine for disk input (di) to string (st)

DATE: October 23, 1985

USAGE

To read input from a disk file to a string, with capabilities to limit the number of characters read, to expand tab characters, and to specify the break character that indicates end of string within the disk file.

The limit specified for the limit on the number of characters in the string should include the break character and the null terminator (if it is not the break character).

The break character may differ from the standard null terminator that is used internally within C programs to indicate an end of string.

By specifying newline as a break character, one line of a standard ASCII file will be copied to a string. The newline will be included and a null byte will be appended.

The number of spaces (tab_q) to be inserted in the string for a tab character is a function parameter. If tab_q = Ø, no tab expansion occurs.

FUNCTION

The specified file is read until the specified break character is detected, the number of characters read equals one less than the limit specified, or the EOF is reached. Characters are placed sequentially into a specified string.

If the limit on number of characters is reached before the break character, the remaining characters in the record to the break character are discarded. The break character is placed at the end and a null terminator appended.

A null terminator ('\Ø') is placed in the string after the last character read. If the first character read is EOF, a null terminator is placed in the first position of the string. If the limit on the string size (q) equals Ø, a return is made without placing any characters in the string.

CALL

```
di_st(fp, stp, q, brk_ch, tab_q)
FILE *fp;                       file pointer
char *stp;                      output string pointer
int   q;                        limit on the string size
char  brk_ch;                   denotes end of string in disk file
int   tab_q;                    number of spaces between tabs
```

(Continued)

di_st (continued)

RETURNS

= -1 if EOF is first character read. A null terminator is placed in the first string position.

= Ø if q = Ø. Nothing is placed in the string.

= number of characters placed in string, including the null terminator

CAUTIONS

Functions included in stdio.h are used.

NAME

dup_wn.c -- duplicates a window structure

DATE: October 23, 1985

USAGE

To duplicate a window structure by copying the elements of a source window to a destination window. The destination window must be defined (have storage allocated) before calling this function.

FUNCTION

All elements of the source window are copied to the destination window.

The elements copied are:

```
int wn.rb
int wn.re
int wn.cb
int wn.ce
int wn.c
int wn.r
char wn.att
char wn.page
int wn.wrap
int wn.location
char wn.scr_q
int wn.l_mg
int wn.r_mg
BORDERPTR wn.bdrp
char wn.setsw
FRECPTR wn.frp
char *wn.storp
char *wn.userp[0]
char *wn.userp[1]
char *wname
char *larray
char *pu_storp
char bdratt
char popup
```

CALL

```
void dup_wn(&dwn, &swn)
WINDOW dwn                    destination window structure
WINDOW swn                    source window structure
```

(Continued)

dup_wn (continued)

RETURNS

None

CAUTIONS

The WINDOW structure must be declared before the call. This function only sets the element values, it does not declare the window.

NAME

errout.c -- prints fatal error messages and exits

DATE: October 23, 1985

USAGE

To exit from programs when a function returns an error signal.

FUNCTION

Prints error message using v_st() and calls exit(1). Exit(1) is the conventional exit indicating an error.

CALL

```
void errout(s1, s2)
char *s1, *s2;                    error message strings
```

RETURNS

None

CAUTIONS

This function does not return. It causes the application program to terminate.

NAME

file_lnp.c -- returns a pointer to the string associated with the i-th line in the file

DATE: October 24, 1985

USAGE

Used to access the string associated with the i-th line in the file.

FUNCTION

Accesses the array of pointers to FLINE structures for the specified line in the file and returns the line_st member found in the FLINE structure.

CALL

```
char *file_lnp(frow, &mfile)
int frow;                       line number to access
FREC mfile;                     FREC structure
```

RETURNS

= NULLP if requested line number is negative or greater than the actual number of lines in the file

= pointer to character string otherwise

CAUTIONS

The first line in the file is line 0.

NAME

get_mem.c -- call to malloc() with error checking

DATE: October 23, 1985

USAGE

Used to call malloc() and perform standard error procedure

FUNCTION

Calls malloc(). If insufficient memory, _wn_err is set to MEMLACK.

CALL

```
char *get_mem(size)
int size;                          number of bytes of memory
```

RETURNS

= pointer to allocated memory if successful

= NULLP if insufficient memory.

ERROR HANDLING

If insufficient memory, _wn_err is set to MEMLACK and an error return is made.

RELATED FUNCTIONS

free_mem()

CAUTIONS

None

NAME

init_wfc.c -- initializes Windows for C system

DATE: February 5, 1986

USAGE

Initializes the global variables for the Windows for C system.

FUNCTION

Initializes the global variables declared in window.h.

Initializes wn0 to cover the full screen.

Initializes internal global variables.

CALL

void init_wfc()

RETURNS

None

CAUTIONS

MSDOS Systems:

Should be called before any Windows for C functions. Many WFC functions, but not all, check if the initialization has been performed and if not, will invoke the initialization routine. To be absolutely safe, the main application program should call this function as its first task. In the future it may be required that the main application program call init_wfc().

UNIX Systems:

Must be called before any Windows for C functions.

NAME

free_file.c -- frees FLINE structures associated with a file

DATE: October 23, 1985

USAGE

To clear a memory file of its contents, so it can be filled again. Frees FLINE structures and string pointers allocated by di_file(). This function does not free the array that holds the pointers to the FLINE structures.

FUNCTION

Calls free_mem() to free each non-NULL string pointer and FLINE structure in the FLINEPTR array pointed to by fr.farray.

free_mem() will return a 0 on errors for those compilers that support this error return. Otherwise a 1 is returned even on errors.

CALL

```
free_file(&fr);
FREC fr;                          pointer to FREC structure
```

RETURNS

= 1 if no error or if compiler does not provide error return

= 0 if attempt to free a pointer not allocated from the heap (only for those compilers supporting an error return).

RELATED FUNCTIONS

def_fr()
di_file()
v_file()

CAUTIONS

Does not free the array that holds the pointers to the FLINE structures. If this is desired, give the call

```
free_mem((char *)mfile.farray);
```

NAME

free_mem.c -- call to free() with error code setting

DATE: October 23, 1985

USAGE

Used to call free() and set error code.

FUNCTION

Calls free() if pointer is not NULLP. If free() makes an error return, _wn_err is set to BADHEAP.

CALL

```
free_mem(p)
char *p;                          pointer to allocated memory
```

RETURNS

= 1 if successful

= Ø if error on free() (only for those compilers that support an error return on free())

ERROR HANDLING

If error return on call to free(), _wn_err is set to BADHEAP and an error return is made. Not all compilers provide an error return from free().

RELATED FUNCTIONS

get_mem()

CAUTIONS

The pointer to the memory block must be cast to a character pointer.

Check your compiler documentation for free(). Many compilers do not make an error return from free() but will exit to the operating system for this error.

NAME

ki.c -- obtains key code from keyboard

DATE: January 31, 1986

USAGE

Waits until a key is pressed, then returns ASCII or extended codes.

FUNCTION

MSDOS Systems:

Calls BIOS int 16H, function 1, and returns code value.

Extended codes are returned as negative values. See the IBM Technical Reference Manual, Section 2, "Keyboard Encoding and Usage" for more information on extended codes.

UNIX Systems:

In general, this routine will read a character (or characters in the case of multiple character escape sequences generated by a single key), check the character (or characters) against the keystroke definition table and return an extended code in case of a match or the raw keystroke in the case of no match.

Reads a character from the keyboard using stdio.h routines. The keystroke definition table is then checked to see if the received keystroke matches the first character of any of the keystroke strings in the table. If no match is found, the keystoke is returned. If a match is found and the length of the string is 1, then the extended code for that string is returned.

If a match is found but the number of keystrokes received is less than the length of the string, another character is read from the keyboard. The keystroke definition table is checked to see if this keystroke matches the second character of any of the keystroke strings in the table. If no match is found and the first keystroke equals the second keystroke, then the keystroke is returned. This permits a <lead-in> character to be entered from the keyboard by pressing that key twice. (For example, ESC on ANSI terminals). If no match is found and the first and second keystrokes are different, then the bell is sounded and the process starts over. If a match is found and length of the string is 2, then the extended code for that string is returned.

If a match is found but the number of keystrokes received is still less than the length of the string, the process is repeated again.

If we process as many keystrokes as in the longest string without finding a match, then the bell is sounded an the process starts over.

CALL

```
ki();
```

(Continued)

ki (continued)

RETURNS

= ASCII code if normal key.

= - (Extended Code) if special key.

RELATED FUNCTIONS

ki_cum( )
ki_chk( )

CAUTIONS

Return must be checked for negative value (which indicates extended code).

MSDOS Systems:

Key definitions in computer.h #define positive values for keys returning extended codes. Compare the negative of these #defined values against ki( ).

CTRL-BREAK is mapped to -1.

UNIX Systems:

The maximum length of any keystroke sequence is 10.

Functionally the same as the PCDOS version; however the actual implementation is different to handle multiple keystroke escape sequences.

NAME

ki_chk.c -- checks buffer for key value; returns value if available.

DATE: October 23, 1985

USAGE

To check if a key code is available in buffer. If available, the key code is returned, but the keystroke is not removed from the buffer.

FUNCTION

MSDOS Systems:

Calls BIOS int 16H, function 2; returns code value if found in buffer. Extended codes are returned as negative values.

UNIX Systems:

Checks the structure associated with stdin to determine if any keystrokes are in the buffer. If there are no keystrokes here it does not mean that there are no keystrokes waiting in the terminal's keyboard buffer.

The value of the keystroke is returned. Extended key codes are returned as negative values.

CALL

```
ki_chk();
```

RETURNS

= ASCII code if normal key.

> For UNIX Systems, this key may be first keystroke of an escape sequence or may be an extended code if a translated keystroke was pushed back by a previous call to ki_cum().

= - (Extended Code) if special key.

= Ø if no code in buffer

RELATED FUNCTIONS

ki()
ki_cum()

CAUTION

Return must be checked for negative value (which indicates extended code).

Key definitions in computer.h #define positive values for keys returning extended codes. Compare the negative of these #defined values against ki().

CTRL-BREAK is mapped to -1.

NAME

ki_cum.c -- obtains identical codes from buffer

DATE: January 31, 1986

USAGE

To obtain at least one key code and to check keyboard buffer for additional identical keycodes. Identical keycodes are removed and the number returned as an element of a structure.

May be used to minimize over-scrolling on systems with slow screen updating (see k_vcom()).

FUNCTION

Waits until at least one key code is available (via ki()) and then calls ki_chk() to check if an additional identical code is still in the buffer. Identical codes are removed from buffer until either a specified maximum number is reached or a non-identical code encountered.

A typedef KEYR structure is used to return the value of the keycode and the number of identical keystrokes found in the buffer.

CALL

```
ki_cum(&keyrec)
KEYR keyrec;                          key record structure

typedef struct krec         defined in bios.h
{
    int kv;                 key code value
    int kq;                 quantity of identical codes found
    int kmax;               limit on number of codes to be returned
} KEYR, *KEYRPTR;
```

RETURNS

= key code value (Extended Codes are returned as negative values).

Values are also returned in keyrec structure:

keyrec.kv = code value (negative value for extended codes).

keyrec.kq = quantity of identical keycodes found.

RELATED FUNCTIONS

ki()
ki_chk()
k_vcom()
vs_file()

(Continued)

ki_cum (continued)

CAUTIONS

keyrec.kmax must be specified before call.

Key definitions in computer.h (terminal.h for UNIX systems) #define positive values for keys returning extended codes. Compare the negative of these #defined values against ki().

MSDOS Systems:

CTRL-BREAK is mapped to -1.

UNIX Systems:

Because ki_cum() calls ki_chk() and the Unix version of ki_chk() is slightly different from the MSDOS version of ki_chk(), the Unix version of ki_cum() will behave slightly differently than the MSDOS version.

In actual practice, if 6 identical keystrokes are in the buffer the MSDOS version of ki_cum() will return all 6 at once. The Unix version of ki_cum() will probably return the 6 keystrokes in two groups. This is due to the nature of ki_chk() and is dependent on how quickly the keystrokes are entered and when the terminal's keyboard is actually read.

NAME

k_vcom.c -- translates special key codes into video movement commands

DATE: April 17, 1986

USAGE

To translate Extended Codes (obtained via ki()) into video movement commands. This function is used by vs_file(). It provides for moving the virtual and screen cursor within a window and also for moving the window over a file that is larger than the window dimensions.

FUNCTION

This function operates on a memory file. It requires that a FREC structure for the file be established, and that a pointer to the FREC be installed in the wn.frp member of the window.

Translates various keys on the cursor pad into movements of the virtual and screen cursors within a window.

The cursor-movement commands will move the cursor to the edge of the window and, if text remains beyond the edge, additional cursor movements will shift the window origin in the file (wn.frp->wfr and wn.frp->wfc will be altered).

This function does not redraw a window. If the window location in the memory file is moved, it is necessary to redraw the window contents upon return. A non-zero return indicates the window origin has been shifted. Use v_file() to redraw the window.

Function vs_file() incorporates k_vcom() and automatically redraws the window when its location in the memory file moves.

Provision is made for handling more than one identical keystroke. For example, if q up-cursor strokes are collected before calling k_vcom, and kq is set equal to q, k_vcom will process q up-cursor moves in one call.

The following keys are translated:

cursor arrow keys:	one space in direction of the arrow; except that left and right arrows cause horizontal scrolling of five spaces when cursor is at window edge.
Home:	top of file
End:	end of file
PgUp:	page up
PgDn:	page down

(Continued)

k_vcom (continued)

MSDOS Systems:

Ctrl-left-arrow:	five spaces left
Ctrl-right-arrow:	five spaces right
Ctrl-PgUp:	five spaces up
Ctrl-PgDn:	five spaces down

UNIX Systems:

Page left:	five spaces left
Page right:	five spaces right

If a keystroke is passed to this function that is not recognized the bell is rung.

Source is provided to allow you to change this function.

CALL

```
k_vcom(kv, kq, &wn)
int kv;                    Extended key code value.
int kq;                    quantity of keystrokes to implement
WINDOW wn;                 window struct
```

RETURNS

= Ø if the window origin in the file-record is unchanged

= 1 if the window origin in the file-record is shifted

RELATED FUNCTIONS

vs_file()
v_file()

CAUTIONS

If k_vcom() causes a shift in file origin (the value returned by k_vcom != Ø), the window contents must be redrawn (using v_file() or alternative function).

NAME

menu2.c -- a menu display and selection routine

DATE: October 23, 1985

USAGE

To display a pop-up menu on a screen, allow menu selection, and then restore the original screen content.

FUNCTION

This function displays a menu that resides in a memory file. Prior to calling this function, the memory file must be created and a pointer to the file placed in member wn.frp of the menu display window. See the text chapters for more information.

Menus have the following format:

Menus can consist of any number of items.

Menu items can be arranged in one or more rows, and each row can have one or more items. The same number of items must be on each row except the last one.

The same space must be allocated to each item in the menu. Use spaces to pad items to make them all the same length.

The window in which the menu is to be displayed is passed as an argument of menu2(). Additional parameters specify the number of items in the menu, number of characters per item, the number of items per row and the default position of the bar cursor.

The area of the screen where the menu is to appear is stored using sav_wi().

The menu-window is placed on the screen and the menu text moved to the window.

The default item is highlighted.

Highlighting is accomplished by using a sub-window that is one-menu item in size. The sub-window is placed over the menu item, and v_natt is called to change the text attribute within the window to REVERSE or LREVERSE.

Cursor pad keys are used to move the highlight area to other menu items.

(Continued)

menu2 (continued)

Pressing the enter key "selects" the menu item highlighted, which causes the following sequence:

The original screen is restored over the menu window.

The code number of the menu item is returned on exit.

The cursor is restored to its original position before this function returns.

Source code for this function is included on the system diskette.

CALL

```
menu2(wnp, qitems, itemlength, item_row, def_pos)
WINDOWPTR wnp;                  menu window struct
int qitems;                     no. of menu items
int itemlength;                 no. of char space per item
int item_row;                   no. of items per row
int def_pos;                    default position in menu
```

origin 1

RETURNS

= the number of the menu item, counting from left to right and starting at one in the upper left corner.

= -1 if error; check _wn_err for more information.

LTS: 0 4 ESC

RELATED FUNCTIONS

def_fr()
di_file()
sti_file()

CAUTIONS

This function displays a menu that resides in a memory file. Prior to calling this function, the memory file must be created and a pointer to the file placed in member wn.frp of the menu display window. See the text chapters for more information.

NAME

mode_col.c -- obtains number of video columns in current mode

DATE: January 31, 1986

USAGE

To obtain the number of character columns (40 or 80) in the current (text) mode.

FUNCTION

MSDOS Systems:

Calls VID_INT to obtain the mode and column quantity via PC INT 10H, with AH = 0FH.

UNIX Systems:

Returns the global variable v_coq. This function is included in the UNIX version for compatibility with the MSDOS version of WFC.

CALL

mode_col()

RETURNS

= number of character columns on screen in current mode.

CAUTIONS

None

NAME

mod_wn.c -- modifies the coordinates of a window

DATE: October 23, 1985

USAGE

Modifies the coordinates of a previously defined window.

FUNCTION

Modifies the coordinates of a previously defined window. The dimension of the window is checked. If the dimension is INSIDE, the dimension is changed to FULL before the modifications are made. The dimension will be restored to its original condition before return.

CALL

```
void mod_wn(rb, cb, rowq, colq, &wn)
int rb;                          new beginning row window
int cb;                          new beginning column of window
int rowq;                        number of rows in window
int colq;                        number of columns in window
WINDOW wn;                       window structure
```

RETURNS

None

CAUTIONS

Changing window dimensions between calls to sav_wi() and unsav_wi() or repl_wi() can result in errors. For popup windows changing the window dimensions between calls to set_wn() and unset_wn() can result in errors. In these situations, the window coordinates may be changed after a call to sav_wi() or set_wn() but should be restored to the original values before a call to unsav_wi(), repl_wi() or unset_wn().

NAME

mv_cs.c -- changes the location of the virtual cursor; does not move screen cursor

DATE: October 23, 1985

USAGE

To change location of virtual cursor. Does not place screen cursor at new virtual cursor position.

FUNCTION

Moves the virtual cursor to specified row and column on designated window.

CALL

```
void mv_cs(rw, co, &wn)
int rw, co;                    row and column
WINDOW wn;                     window structure
```

RETURNS

None

CAUTIONS

None

NAME

mv_csr.c -- changes the location of the virtual cursor and moves the screen cursor to new location

DATE: January 31, 1986

USAGE

To move the virtual and screen cursor to a new location.

FUNCTION

Moves the virtual and screen cursor to the specified row and column on the designated window.

CALL

```
void mv_csr(rw, co, &wn)
int rw, co;                    row and column
WINDOW wn;                     window structure
```

RETURNS

None

CAUTIONS

MSDOS Systems:

wn.page, the active video page, must be set correctly before this call. Unless you are using multiple pages of the color graphics display cards (not usual), this will be done by defs_wn() and def_wn().

NAME

mv_rws.c -- scrolls designated window by nlines in given direction (nlines = Ø, clears window).

DATE: October 23, 1985

USAGE

To scroll lines of text up or down on wn; also to clear window. This function is used to create space for new output at the top or bottom of a screen window. It is not used for scrolling within memory files using the cursor control keys. Function vs_file() is provided for moving a window over a text file contained in memory.

FUNCTION

Uses v_mov() to move nlines of text up or down in a screen window. All of the rows within the window are affected. Information scrolled off the top or the bottom of the window is lost. Blank rows will be created at the bottom of the window (scroll direction = UP) or at the top of the window (scroll direction = DOWN). Decrements (scroll up) or increments (scroll down) virtual cursor row value by nlines, so it is correctly valued.

The maximum allowed value of the virtual cursor after scrolling is one row below the working dimensions of the window (with wn.c = Ø).

The minimum allowed value of the virtual cursor after scrolling is Ø, Ø.

If nlines = Ø, the window is cleared and cs is set at Ø,Ø. The cleared area is filled with spaces of attribute wn.att.

If nlines is greater than the window size, the window is cleared. The virtual cursor is set to the maximum (direction = DOWN) or the minimum (direction = UP).

CALL

```
mv_rws(nlines, dir, &wn);
int nlines;                    number of lines to scroll
char dir;                      direction 6=UP, 7=DOWN (defined in wfc_defs.h)
WINDOW wn;                     window structure
```

RETURNS

= 1 if successful

= -1 if error; check _wn_err code for more information

CAUTIONS

None

NAME

mv_wi.c -- moves a window image to a new screen location

DATE: October 23, 1985

USAGE

To move a screen window, including borders and contents, to a new location on the screen. The location of the window structure and the window image on the screen are both moved.

FUNCTION

Calls sav_wi( ) to save window image. If there was already a saved window image in wn.storp, it will be freed. The window is then removed from the screen with a call to unset_wn( ). If this was a pop-up window, the underlying information will be restored. The window location members are changed to the new values by a call to pl_wn( ). If the window is a pop-up window, the underlying information is saved and the screen image is placed in the new location with unsav_wi( ).

Parameters in the call specify the origin of the FULL dimension of the window. Working dimensions are not altered. See dim_wn( ) for definitions.

Because pl_wn( ) keeps windows within the screen boundaries, this function will not permit window images to be moved off the screen.

Memory is allocated temporarily for the saved image but is released at the end of the call.

If window is a pop-up window, the contents of wn.pu_storp will be changed.

Member wn.storp will always be NULLP on return.

CALL

```
mv_wi(rw, co, &wn)
int rw;                          screen row of new origin
int co;                          screen column of new origin
WINDOW wn;                       window structure
```

RETURNS

= Ø if move is successful

= -1 if insufficient memory for move

CAUTIONS

None

NAME

pl_csr.c -- Sets screen cursor at virtual cursor in a window

DATE: February 3, 1986

USAGE

To set screen cursor at the location of virtual cursor in a designated window.

FUNCTION

MSDOS Systems:

Calls VID_INT to set the cursor via PC INT 10H, function 3.

UNIX Systems:

Uses the appropriate command string defined in wfctermcap to position the cursor.

CALL

```
void pl_csr(&wn)
WINDOW wn;                          pointer to window struct
```

RETURNS

None

CAUTIONS

MSDOS Systems:

If virtual cursor is off video screen, cursor will disappear.

UNIX Systems:

Functionally the same as the PCDOS version; however the actual implementation is different to handle terminals.

If virtual cursor is off of the video screen, an attempt will be made to turn the cursor off. The cursor will not be moved.

NAME

pl_mfwn.c -- places window origin at specified location in file

DATE: October 24, 1985

USAGE

Used to set the origin of the window within the file. v_file() has the capability of displaying different sections of the file within the window based on wnp->frp->wfr and wnp->frp->wfc. pl_mfwn() provides a convenient method of setting fr.wfr and fr.wfc.

FUNCTION

Sets wnp->frp->wfr and wnp->frp->wfc to the specified values.

CALL

```
pl_mfwn(frow, fcol, &wn)
int frow;          row of file to appear in first row of window
int fcol;          column of file to appear in first column of
                   window
WINDOW wn;         WINDOW structure
```

RETURNS

= 0 if wn.frp is NULLP

= 1 otherwise

CAUTIONS

Pointer to FREC must be installed in WINDOW structure.

NAME

pl_wn.c -- sets location members of a window structure to a specified position on the screen

DATE: October 23, 1985

USAGE

To set the location members of a window to a desired location on screen.

FUNCTION

Sets the FULL dimension origin of a window (wn.rb and wn.cb) to the specified row and column values. Working dimensions not altered. See dim_wn( ) for definitions.

Will not locate a window so that any part of it is outside of the borders of the screen.

CALL

```
void pl_wn(rw,co,wnp)
int rw;                          screen row
int co;                          screen column
WINDOWPTR wnp;                   window structure
```

RETURNS

None

CAUTIONS

Do not use this function to move a window that has already been placed on the screen. Use mv_wi( ).

This function only changes the member values of a window structure. It does not set the window on the screen.

NAME

prt_wn.c -- copy character contents of window to printer

DATE: February 3, 1986

USAGE

Copies the character contents within the working dimensions of a window to the printer. If the dimensions are set to INSIDE (wn.setsw = 1), the border will not be copied. To copy the full window including the border, the dimensions need to be FULL (wn.setsw = Ø). If necessary, use dim_wn() to alter the working dimensions.

FUNCTION

The type of copy (FULL or INSIDE) is determined by the value of wn.setsw.

MSDOS Systems:

Calls copy_wc() to transfer window contents to the printer "file".

UNIX Systems:

Opens a pipe to the shell command "lpr" and transfers the contents of the window through the pipe.

CALL

```
prt_wn(&wn)
WINDOW wn;                          window structure
```

RETURNS

= 1 if the copy is successful.

= Ø if copy unsuccessful because file can not be opened or there is insufficient memory for the video-string buffer.

= -1 if wn.setsw not equal to zero or one.

CAUTIONS

None

NAME

rd_csr.c -- reads cursor position

DATE: February 3, 1986

USAGE

To read x and y coordinates of the cursor on the screen.

FUNCTION

MSDOS Systems:

Calls vid_int to read the screen cursor position function via PC INT 10H, with AH = 3.

"page" is a parameter in the call. The page applies only to the Color/Graphics Adapter(CGA) or the Enhanced Graphics Adapter(EGA). The normal setting for this is 0 (zero), unless you are using the additional memory pages available on the CGA or EGA.

UNIX Systems:

Places the contents of the internal global variables used to hold the row and column of the screen cursor in the locations specified by row and col.

If the user makes use of non-WFC output routines (such as putchar(), puts(), printf(), etc), the values returned by this function will be incorrect until an intervening pl_csr() call is made.

The 'page' argument in the call is retained for compatibility with the PC/MSDOS WFC version. It has no meaning at the present time.

CALL

```
void rd_csr(&row, &col, page)
int row;                        variable to contain row
int col;                        variable to contain column
int page;                       page number (graphics card)
```

RETURNS

None. Values are placed in variables pointed to in the call.

(Continued)

rd_csr (continued)

CAUTIONS

Pointers to the row and column positions must be passed in call, not the variables themselves.

MSDOS Systems:

"page" number must be set if the graphics display adapter is being used.

UNIX Systems:

Use of non-WFC output routines (such as putchar(), puts(), printf(), etc), will result in incorrect values being returned by this function until an intervening pl_csr() call is made.

NAME

rd_mode.c -- returns current video mode

DATE: February 3, 1986

USAGE

To obtain the current video mode.

FUNCTION

MSDOS Systems:

Calls vid_int( ) to read the current video mode via PC INT 1ØH, with AH = 15. The mode values returned are the same as those used by vid_mode( ).

UNIX Systems:

Simply returns the value of the global variable v_mode. This global variable indicates if the terminal is operating in a monochrome mode or color mode.

CALL

rd_mode( )

RETURNS

= current video mode

RELATED FUNCTIONS

vid_mode( )

CAUTIONS

None

NAME

repl_wi.c -- places a saved window on the screen

DATE: October 23, 1985

USAGE

To place on the screen a window previously saved by sav_wi(). Image stored in memory is retained. Use unsav_wi() to free memory at the same time.

FUNCTION

The contents of the window image stored in the video string pointed to by wn.storp are transferred to the screen to the current location of wn, via v_mov(). See sav_wi().

If wn.storp = NULLP, no transfer is made.

CALL

```
repl_wi(&wn)
WINDOW wn;                          window structure
```

RETURNS

= 0 if replace is successful

= -1 if wn.storp = NULL

RELATED FUNCTIONS

sav_wi()
unsav_wi()

CAUTIONS

None

NAME

sav_wi.c -- saves a window image in memory

DATE: October 23, 1985

USAGE

To save a window image currently on the screen in order to replace it later, with repl_wi( ) or unsav_wi( ).

FUNCTION

The contents of the window image in the regen buffer (attributes and characters) are transferred to an internal video string via v_mov( ).

Space for the video string is allocated via get_mem( ) and a pointer to the string is placed in the window structure (in wn.storp) for use by other functions that operate on saved or stored windows.

The FULL window is saved. Working dimensions are not altered.

CALL

```
sav_wi(&wn)
WINDOW wn;                          window structure
```

RETURNS

= Ø if save is successful

= -1 if insufficient memory for save

RELATED FUNCTIONS

repl_wi( )
unsav_wi( )

CAUTIONS

Memory is used by this function. When the saved window is no longer needed, release the memory by a call to unsav_wi( ) or to free_mem(wn.storp).

NAME

scrl_file.c -- scrolls a memory file

DATE: October 23, 1985

USAGE

To scroll a group of lines in a memory file up or down within the file structure. Lines scrolled off the top or bottom of the file structure are lost.

All rows in a file may be scrolled by specifying the special #defined constant ALL_ROWS as the beginning row.

FUNCTION

This function operates on a memory file. Prior to calling this function, a memory file record (of type FREC) must have been initialized, using def_fr(), and information placed in it using di_file() or sti_file().

The scrolling direction is checked. If it is not UP or DOWN, an error return is made.

If the specified beginning row of the scrolling region is not within the limits of the file, an immediate return of Ø is made.

The number of rows to scroll is adjusted, if necessary, so that it does not exceed the number of rows between the beginning row and the limit of the memory file.

If the specified beginning row is ALL_ROWS, the actual beginning row is set according to the scrolling direction. For UP, the beginning row is set to the last row in the file. For DOWN, the beginning row is set to the first row. Setting the number of rows to scroll to a value larger than the file will essentially cause all rows in the file to be replaced with blank lines.

If the scrolling direction is UP, all of the rows between and including the first row of the file and the specified beginning row (beg_row) are scrolled up the specified number of rows (nrows). The first nrows of the file are lost. Starting at the row where the scrolling initiated (beg_row), nrows of blank lines are added.

If the scrolling direction is DOWN, all of the rows between and including the specified beginning row (beg_row) and the last row in the file are scrolled down the specified number of rows (nrows). The last nrows of the file are lost. Starting at the row where the scrolling initiated (beg_row), nrows of blank lines are added.

All scrolling is accomplished by moving pointers within the memory file array pointed to by mfile.farray.

(Continued)

scrl_file (continued)

If the current row is to be scrolled off the top of the file (dir = UP) or the bottom of the file (dir = DOWN), the FLINE structure associated with this row is freed.

If a blank line is to be placed in a row, the existing FLINE structure is freed and a new FLINE structure is allocated and initialized to a blank line. A pointer to the new FLINE structure is placed in the appropriate element of the memory file array.

If an error occurred when trying to free an existing FLINE structure or allocating a new FLINE structure, an error return is made.

This function does not alter the total number of rows in a memory file. It will only change the relative position of the lines in the file.

CALL

```
scrl_file(nrows, dir, beg_row, &mfile)
int nrows;                      number of rows to scroll
int dir;                        direction to scroll
int beg_row;                    starting row of scroll region
FREC mfile;                     FREC structure
```

RETURNS

= 0 if direction not UP or DOWN, beginning line outside file limits, or memory allocation error; check _wn_err code for more information

= 1 if success

ERROR HANDLING

One possible error would result from insufficient memory to allocate the necessary structures for the new blank lines. In this case, the global error variable _wn_err will be set to MEMLACK. Insufficient memory is unlikely since in almost all cases we will free more memory than we are likely to allocate.

If the error return is due to parameters specified that are outside file limits, _wn_err will equal 0.

(Continued)

scrl_file (continued)

RELATED FUNCTIONS

def_fr()
di_file()
sti_file()
v_file()
vs_file()

CAUTIONS

Rows are numbered starting with Ø.

Prior to using this function, a memory file record must be established and lines placed in the memory file.

NAME

set_wn.c -- sets a window on the screen

DATE: October 23, 1985

USAGE

Use to place a window on the screen.

FUNCTION

For given window:

1) Checks defining parameters for consistency: after allowing for margins and border, window must be at least one character in size; all dimensions must be non-negative.

2) Checks the window dimensions to insure that it does not lie outside of the screen. When necessary, the values of wn.rb and wn.ce are reduced to make the window lie entirely within the screen.

3) If wn.popup = 1, the information underlying the window is saved so that it may be restored when the window is unset. A pointer to the saved image is placed in wn.pu_storp.

4) Clears window with "space" character and attribute wn.att.

5) Draws 1-character wide border (using v_border), unless a null border (BDR_ØP) is specified.

6) Adjusts the working dimensions of the window to inside dimensions, which are smaller than the full dimensions of the window by the amount of the border and margins.

7) Sets wn.setsw = INSIDE (= 1) to indicate inside dimensions have been set. See dim_wn() for further explanation. The value of wn.setsw can be used to tell if a window is on the screen. All Windows for C functions maintain this value consistently. It will always be Ø when the window is not on the screen.

8) Places virtual cursor at Ø, Ø.

CALL

```
set_wn(&wn)
WINDOWPTR &wn;                    pointer to window structure
```

(Continued)

set_wn (continued)

RETURNS

= Ø if fails consistency check, if any wn dimension is negative or if unable to allocate storage for underlying window in case wn.popup is set

= 1 otherwise.

CAUTIONS

Initial values of window structure must be defined before call.

Repeated calls can be made to this function without creating errors in the working dimensions of a window, as long as the BORDER structure associated with the window has not been changed. If called more than once for the same window without an intervening call to un_set(), the border must have the same dimensions as in the prior call; otherwise screen dimensions will be in error.

NAME

size_wn.c -- calculates size of video string needed for window saves

DATE: October 15, 1985

USAGE

To determine the number of bytes that must be allocated to a video string to store a window or window section. Useful for calls to get_mem() used in conjunction with v_mov().

FUNCTION

An argument in the call specifies whether size is calculated for the FULL or INSIDE dimensions of the window (see dim_wn()). The working dimensions of the window are not altered.

A second argument specifies the part of the window to be sized. Five window parts are defined:

CH -- the character at the location of the virtual cursor

ENDROW -- from present location of virtual cursor (cs) to last column of this row

ROW -- row on which cs is located

COL -- column on which cs is located

ENDCOL -- from the present location of virtual cursor to the last row in this column

ENDWIND -- row on which cs is located to end of window

WIND -- entire window

The values of the window "parts", ENDROW, etc., are #defined in wfc_defs.h.

The calculated size equals two bytes for each character space in the window section specified (one for the character and one for the attribute byte).

CALL

```
size_wn(dimen, part, &wn)
char dimen;                    FULL (Ø) or INSIDE (1) dimensions
int part;                      portion of the window to size
WINDOW wn;                     window structure
```

RETURNS

= size (in bytes) of video string needed to hold window part

NAME

sti_file.c -- places a string in a memory file

DATE: October 23, 1985

USAGE

To place a string in a specified row of a memory file.

FUNCTION

This function provides a means of directly placing strings in a memory file. It provides an alternative to di_file() for placing information in a memory file. Function di_file() transfers lines from a disk file to a memory file.

This function draws on information set in a passed FREC structure, mfile, to perform the file operations. The FREC must be initialized by a call to def_fr() prior to calling this function. Function def_fr() will allocate and initialize a memory file array, which is an array of pointers to FLINE structures.

A memory file consists of a number of FLINE structures. The structure FLINE, fline, contains two members:

> fline.line_len is an integer and holds the length of the file line (excluding the terminal newline and null).
>
> fline.line_st is a character pointer and points to the location where the file line-string is stored.

Function sti_file() allocates memory for a FLINE structure and for a copy of the string that is to be placed in the memory file. It fills the FLINE structure with the appropriate information and places a pointer to the structure in the specified row of the memory file array.

If there is already a pointer for the specified element in the array, the pointer is freed.

Function sti_file() performs the following checks and actions:

> If the row specified as the location of the string in the memory file is less than Ø or greater than the length of the memory file (mfile.fmaxlines), an immediate return of Ø is made.
>
> The length of the specified string is checked. The string length must be less than mfile.fmaxcol. If the string is longer than mfile.maxcol, the string is truncated. A return of -1 will be made to indicate that the string was truncated.

(Continued)

sti_file (continued)

Enough space is allocated to hold a copy of the specified string plus a newline which will be appended to the string. The specified string is copied to the new location and the newline appended. A pointer to the allocated string location is placed in fline.line_st. The length of the string (excluding the newline ('\n') and null terminator ('\Ø')) is placed in the fline.line_len.

The structure members mfile.ln_q and mfile.c_q are updated. See def_fr() for the definition of these members.

CALL

```
sti_file(st, frow, &mfile)
char *st;                   pointer to source string
int frow;                   row in file to place new string
FREC mfile;                 FREC structure
```

RETURNS

= -1 if string is truncated

= Ø if unable to allocate sufficient memory or frow outside file limits

= 1 if success

ERROR HANDLING

If unable to allocate sufficient memory, the global variable _wn_err is set to MEMLACK. If frow is outside limits, _wn_err will equal Ø.

RELATED FUNCTIONS

def_fr()
di_file()
v_file()
vs__file()
scrl_file()

CAUTIONS

Before using this function, the memory file must be initialized by calling def_fr().

NAME

stringf.c -- string utility functions

DATE: October 24, 1985

USAGE & FUNCTION

lower_st()

Each position in the string is converted to lowercase.

skip_wh()

Skips the leading white space in the string and returns a pointer to the first non-whitespace character.

stblank()

Allocates memory to hold the string and the terminating '\Ø' and initializes the string to blanks. Returns a pointer to the string.

strcpyp()

Copies the contents of the source string to the destination string. Returns a pointer to the terminating '\Ø' in the destination string.

strip_wh()

Strips the trailing white space from the string and repositions the terminating '\Ø'.

upper_st()

Converts the string to uppercase.

CALLS

```
void lower_st(st)
char *st;                          pointer to string

char *skip_wh(st)
char *st;                          pointer to string

char *stblank(len)
int len;                           length of string to create

char *strcpyp(dest, src)
char *dest;                        pointer to destination string
char *src;                         pointer to source string
```

(Continued)

stringf (continued)

```
void strip_wh(st)
char *st;                              pointer to string

void upper_st(st)
char *st;                              pointer to string
```

RETURNS

lower_st()

    None

skip_wh()

    = NULLP if string is all white space
    = pointer to first non-white space character in string

stblank()

    = NULLP if unable to allocate memory
    = pointer to string

strcpyp()

    = pointer to terminating '\Ø' in destination string

strip_wh()

    None

upper_st()

    None

CAUTIONS

Little or no error checking is done by any of these routines. It is your responsibility to pass correct arguments.

Any routine that adjusts the size of a string assumes that the string buffer is large enough to hold the longer string.

NAME

```
sw_att()    - sets wn.att to the specified value
sw_bdratt() - sets wn.bdratt to the specified value
sw_border() - sets wn.bdrp to the specified border
sw_cleor()  - sets the clear to end of row bit-switch ON/OFF
sw_csadv()  - sets the virtual cursor advance bit-switch ON/OFF
sw_latt()   - sets wn.larray to point to the specified logical attribute array
sw_margin() - sets wn.l_mg and wn.r_mg to the specified values
sw_mfile()  - sets wn.frp to point to the specified memory file structure
sw_name()   - sets wn.wname to the specified name
sw_plcsr()  - sets the place cursor bit-switch ON/OFF
sw_popup()  - sets wn.popup to the specified state
sw_scroll() - sets wn.scr_q to the specified state
sw_wwrap()  - sets the wrap bit-switch ON/OFF
```

DATE: October 23, 1985

USAGE

These functions control the setting of the bit-switches controlling the string output functions and will set various window structure members.

FUNCTION

The functions will modify wn.wrap accordingly.

```
sw_wwrap()  - will set the wrap bit-switch to the specified state
sw_cleor()  - will set the clear to end of row bit-switch to the specified
              state
sw_csadv()  - will set the cursor advance bit-switch to the specified state
sw_plcsr()  - will set the place cursor bit-switch to the specified state
```

The following functions set various members in the window structure.

```
sw_att()    - will set wn.att to the specified attribute
sw_bdratt() - will set wn.bdratt to the specified attribute
sw_border() - will set wn.bdrp to point to the specified border
sw_latt()   - sets wn.larray to point to the specified logical attribute array
sw_margin() - sets wn.l_mg and wn.r_mg to the specified values
sw_mfile()  - sets wn.frp to point to the specified memory file structure
sw_name()   - will set wn.wname to point to the specified window name
sw_popup()  - will set wn.popup to the specified state
sw_scroll() - will set wn.scr_q to the specified state
```

CALL

```
void sw_att(att, &wn)
int att;                          video attribute
WINDOW wn;                        window structure
```

(Continued)

Set Window Member Macros (continued)

```
void sw_bdratt(att, &wn)
int att;                        video attribute
WINDOW wn;                      window structure

void sw_border(&bdr, &wn)
BORDER bdr;                     border structure
WINDOW wn;                      window structure

void sw_cleor(state, &wn)
int state;                      state of clear to end of row bit-switch
WINDOW wn;                      window structure

void sw_csadv(state, &wn)
int state;                      state of cursor advance bit-switch
WINDOW wn;                      window structure

void sw_latt(att_arry, &wn)
char att_arry[];                logical attribute array
WINDOW wn;                      window structure

void sw_margin(l_mg, r_mg, &wn)
int l_mg, r_mg;                 values for left and right margins
WINDOW wn;                      window structure

void sw_mfile(&mfile, &wn)
FREC mfile;                     memory file FREC structure
WINDOW wn;                      window structure

void sw_name(name, &wn)
char *name;                     pointer to window name string
WINDOW wn;                      window structure

void sw_plcsr(state, &wn)
int state;                      state of place cursor bit-switch
WINDOW wn;                      window structure

void sw_popup(state, &wn)
int state;                      state of pop-up member; YES or NO
WINDOW wn;                      window structure

void sw_scroll(state, &wn)
int state;                      value to set wn.scr_q
WINDOW wn;                      window structure

void sw_wwrap(state, &wn)
int state;                      state of wrap bit-switch
WINDOW wn;                      window structure
```

(Continued)

Set Window Member Macros (continued)

RETURNS

None - for all functions

CAUTIONS

All of these functions are implemented as macros; beware of side effects.

NAME

s_latt -- copies appropriate column from the attribute table into the logical attribute array

USAGE

Used by initialization routines to put correct physical attributes for the current mode into a logical attribute array.

FUNCTION

If the global variable _lattsw is 0, physical attributes rather than logical attributes are to be implemented. A return of 0 is made immediately in this case.

This function is passed five arguments. The first argument specifies the column to be copied from the multiple column table of logical attribute information. The second argument specifies the number of rows in the logical attribute table. The third argument specifies the number of columns in the logical attribute table. The fourth argument is a pointer to the multiple-column table of logical attribute information. The fifth argument is a pointer to an single-index array that will contain the correct logical attribute information for the current mode and display adapter.

The specified column in the multi-dimensional table is copied to the single - index logical-attribute-array.

Note that graphics modes are not supported; thus the logical attribute tables and arrays only are relevant to the alpha/numeric modes.

CALL

_attrows _attcols datt_tbl latt

```
s_latt(col, rowq, colq, att_tbl, att_arry)
int col;                        column to be copied
int rowq;                       number of rows in att table
int colq;                       number of cols in att table
char att_tbl[][];               pointer to multi-dimensional table
char att_arry[];                pointer to single-dimensional array
```

RETURNS

= 1 if logical attributes are in use (_lattsw = 1)

= 0 if physical attributes are in use (_lattsw = 0)

CAUTIONS

The variable att_arry must be dimensioned large enough to receive a single column from att_tbl.

NAME

unsav_wi.c -- places a saved window on the screen and frees memory

DATE: October 23, 1985

USAGE

To place on the screen a window previously saved by sav_wi() and to free the memory space used for storage. Use repl_wi() to replace the window and retain the memory image.

FUNCTION

The contents of the window image stored in the video string pointed to by wn.storp are transferred to the screen to the current location of wn, via v_mov(). See sav_wi().

If wn.storp = NULLP initially, no transfer is made. Otherwise, memory allocated to storing the window image is freed after the transfer has been made, and wn.storp is set to NULLP.

CALL

```
unsav_wi(&wn)
WINDOW wn;                          window structure
```

RETURNS

= Ø if unsave is successful

= -1 if wn.storp = NULLP or if free_mem() is unsuccessful

ERROR HANDLING

On an error return by free_mem(), _wn_err is set to BADHEAP. This only applies to compilers that support an error return by free(). Check the documentation for your compiler.

RELATED FUNCTIONS

sav_wi()
repl_wi()

CAUTIONS

None

NAME

unset_wn.c -- removes a window from the screen, resets dimensions

DATE: October 23, 1985

USAGE

Use to remove a window from screen. Will restore underlying information if wn.popup was set.

FUNCTION

If wn.popup is set, the underlying information is restored, otherwise the window is cleared with spaces with attribute in global variable cl_att. Variable cl_att is declared in window.h with the initial value NORMAL or LDOS. The value of cl_att can be changed to provide a different background for the screen.

Sets working dimensions to FULL. (set_wn()) sets working dimensions to INSIDE.) See dim_wn() for definitions.

Sets cs location to origin (Ø,Ø).

CALL

```
unset_wn(&wn)
WINDOW wn;                          window structure
```

RETURNS

= -1 if unable to restore underlying information for a pop-up window

= Ø otherwise

CAUTIONS

If bdrp is not the same as the one used in the call to set_wn, an error in dimensions may be created (if one had a NULL border but not the other).

If this was a popup window, the underlying information where the window existed will be restored. Otherwise, the screen where the window existed will be filled with cl_att attribute spaces. If the underlying screen should have a different attribute, change the value of cl_att before calling unset_wn().

NAME

u_init -- allows users to initialize globally known variables

DATE: October 23, 1985

USAGE

To alter the values assigned by the initialization routine, init_wfc(), to the globally known variables that reflect specific computer and mode display information. The user supplies code in this routine.

FUNCTION

This function is called by init_wfc() just prior to returning; thus any values assigned in u_init() will take precedence over those assigned previously by init_wfc().

init_wfc() uses values supplied in computer.h to initialize the values of globally known variables declared in window.h. See these files for specific information.

As supplied in the Windows for C library, u_init() is a dummy routine, returning immediately upon call.

CALL

```
void u_init();
```

RETURNS

None

CAUTIONS

None

NAME

vid_bdr.c -- sets color of the screen border (IBM PC Color/Graphics Adapter only)

DATE: February 3, 1986

USAGE

When using physical attributes, to set the color of the screen border in the text modes on the IBM PC Color/Graphics Adapter.

FUNCTION

MSDOS Systems:

Uses physical color attributes defined in computer.h to describe the desired border color. Any one of sixteen colors may be specified. See the discussion under color_wn() for a description of the physical color attributes.

Calls BIOS interrupt 1ØH, function 11 (via vid_int()) to set the border color.

UNIX Systems:

This is a dummy function. It does nothing and returns nothing.

CALL

```
void vid_bdr(color)
int color;
```

RETURNS

None

CAUTIONS

MSDOS Systems:

Works with physical video attributes only. Will not work correctly if passed a logical video attribute.

UNIX Systems:

Provided for compatibility with MSDOS versions of Windows for C. This function has no effect in the terminal environment.

NAME

vid_int.c -- General video interrupt (INT 10H) Routine

DATE: February 3, 1986

USAGE

To access any of the video-IO interrupt functions (INT 10H) in the IBM PC ROM BIOS from C programs.

FUNCTION

MSDOS Systems:

This function is passed pointers to VIDIO data structures that contain values for the 8088/8086/80186/80286 registers: ax, bx, cx, dx, si, di, es, and ds. The values in the input structure are transferred to the 80xxx registers and INT 10H is called. The values returned by the interrupt are placed in the data structure for the output values. See wfc_stru.h for the definition of the structure.

For information on the input and output values for INT 10H, see the IBM Technical Reference Manual -- 6025005 , Appendix A, pp. A-47 to A-49.

Many of the available INT 10H functions have been #defined in computer.h for your convenience. These constants would be loaded into vri.ax to invoke the appropriate INT 10H function.

UNIX Systems:

This is a dummy function. It does nothing and returns nothing. It is provided for compatibility with PC/MSDOS versions of Windows for C.

CALL

```
void vid_int(&vri, &vro)
VIDIO vri                    input data structure
VIDIO vro                    output data structure
```

RETURNS

None

CAUTIONS

MSDOS Systems:

None

UNIX Systems:

Provided for compatibility with MSDOS version of Windows for C. This function has no effect in the terminal environment.

NAME

vid_mode.c -- sets video mode, switches display adapters

DATE: April 17, 1986

USAGE

To set the video mode on the IBM PC and to switch between display adapters when both the Color/Graphics and Monochrome Adapters are installed.

Can be called from within a program to allow displays to be switched dynamically. Useful for program development as well as for allowing end users to select a video mode.

Also used to check for the IBM Enhanced Graphics Adapter (EGA) being present and active without switching modes.

FUNCTION

MSDOS Systems:

The code numbers for video mode as defined in the ROM BIOS are used. The code specified in the call is used to set the proper equipment flag value in the BIOS data area, and then INT 1ØH, function Ø is called to reset the mode.

If the IBM EGA is present and active, the global variable _ibmega is set to 1, otherwise it is set to Ø.

If the "mode number" passed as an argument in the call is < Ø, this function will set the value of _ibmega and return without switching modes.

The modes and corresponding mode numbers are:

Mode	Mode Number	Symbolic Mode Value
Text Modes		
40x25 B&W	Ø	BW4Ø
40x25 Color	1	BW4Ø
80x25 B&W	2	CO8Ø
80x25 Color	3	CO8Ø
Graphics Modes		
320x200 Color	4	
320x200 B&W	5	
640x200 B&W	6	
Monochrome Board	7	MONO

NOTE: Windows for C does not support any graphics modes. If vid_mode is called with mode numbers 4-6, all video display routines of Windows for C will be inoperative.

(Continued)

vid_mode (continued)

The symbolic mode values can be used instead of the mode numbers in vid_mode(). The symbolic values are the same as those used in the DOS 2.0 MODE command. They are #defined in the computer.h (which is nested within bios.h).

Mode numbers greater than 7 are treated as graphics modes.

UNIX Systems:

This is a dummy function. It does nothing and returns nothing. It is provided for compatibility with MSDOS version of Windows for C.

CALL

```
void vid_mode(mode)
int mode;                          video mode number
```

RETURNS

None

CAUTIONS

MSDOS Systems:

None

UNIX Systems:

Provided for compatibility with PC/MSDOS version of Windows for C. This function has no effect in a terminal environment.

NAME

vo_att.c -- reads a attribute from a window

USAGE

To read an attribute from a window.

FUNCTION

Calls v_mov( ) to obtain the physical attribute at the current virtual cursor position and returns its value.

The attribute returned will be the physical attribute for the character at the virtual cursor position. There is no way to map this physical attribute to its logical attribute counterpart.

The virtual cursor is not advanced.

CALL

```
vo_att(&wn)
WINDOW wn;                          window structure
```

RETURNS

= The attribute read.

CAUTIONS

Returns physical attribute only.

NAME

vo_ch.c -- reads a character from a window

USAGE

To read a character from a window.

FUNCTION

Calls v_mova( ) to obtain the character at the current virtual cursor position and returns its value.

The virtual cursor is not advanced.

CALL

```
vo_ch(&wn)
WINDOW wn;                          window structure
```

RETURNS

The ASCII code for the character read. The valid codes range from Ø to 255.

RELATED FUNCTIONS

v_mova( )

CAUTIONS

For multiple characters, v_mova( ) is much faster. Use the predefined parts of a window available for v_mova( ) or use a small one-row window to read part of a line in a larger window.

NAME

vs_file.c -- view and scroll a file

DATE: October 23, 1985

USAGE

To allow a user to view a file through a window and scroll through it.

FUNCTION

This function opens a display window on a memory file and then allows the user to scroll through it.

Prior to calling this function, a memory file record (of type FREC) must have been initialized, using def_fr(), and information placed in it using di_file() or sti_file(). The window member wn.frp must have been set to point to the memory file record.

Uses v_file() to display the file and calls k_vcom() to allow the user to scroll through the file by using the cursor pad keys.

Purges identical keystrokes from the keyboard buffer to prevent continued scrolling after a cursor key is released.

Calls unset_wn() prior to returning.

Returns when the "exit-key" specified in the call is pressed. Note that the keycode for keys that return "extended codes" must be negative (see Caution).

Top and bottom of file messages can optionally be displayed. The default is to have the messages displayed. The display is controlled by the global variable tbf_msg. You can turn off the message by direct assignment

```
tbf_msg = OFF;
```

or use the system macro provided:

```
s_tbfmsg(OFF);
```

Source for this file is provided on the system diskette.

CALL

```
void vs_file(exit_key, &wn)
int exit_key;                    key code for exit key
WINDOW wn;                       pointer to window structure
```

RETURNS

None

(Continued)

vs_file (continued)

RELATED FUNCTIONS

def_fr()
di_file()
sti_file()
v_file()
scrl_file()

CAUTIONS

The values #defined for "extended code" keys (which includes the function keys) in computer.h are positive. If one of these is used as the exit key, place a minus sign in front of it in the call.

Prior to using this function, a memory file record must be established, lines placed in the file, and window member wn.frp must be pointed to the memory file record.

NAME

v_att.c -- writes the specified video attribute to the current virtual cursor position

DATE: October 23, 1985

USAGE

To write the specified video attribute to the current virtual cursor position.

FUNCTION

Calls v_natt(att, CH, wnp) to change the attribute associated with the character at the current virtual cursor position.

If virtual cursor advance is turned off (see v_st_rw()), the virtual cursor is unchanged.

If virtual cursor advance is turned on:

> The virtual cursor position is advanced. If the virtual cursor is moved beyond the end of a row, it will be placed at the first column of the next row.
>
> If the current virtual cursor position is the last position in the window, the virtual cursor will be placed in the row below the window so that adj_cs() can detect that the window is full.

If the present virtual cursor position is not within the window, the attribute will not be written.

This routine supports both logical and physical video attributes.

CALL

```
v_att(att, &wn)
int att;                        video attribute
WINDOW wn;                      window structure
```

RETURNS

= Ø if window is filled

= 1 otherwise

CAUTIONS

For changing the attribute of more than one character at a time, v_natt() is much faster. Use the predefined parts of a window available for v_natt() or use a small one-row window to read part of a line in a larger window.

NAME

v_axes.c -- draws axes for graphs

DATE: October 23, 1985

USAGE

To draw horizontal and vertical axes for graphs

FUNCTION

Calls v_rw and v_co to draw axes of specified length from a specified origin.

The bars are drawn within window "wn", specified as an argument in the call. The bar location is measured relative to the window origin.

Axes are drawn with REVERSE or LREVERSE video spaces, so they entirely fill the line and columns designated as the origins of the axes. Because of the nature of text graphics, the horizontal axis is relatively wide.

If non-error return, wn.r and wn.c are set at origin of axes.

CALL

```
v_axes(r_origin, c_origin, height, width, &wn)
int r_origin;                      row value of origin
int c_origin;                      column value of origin
int height;                        height of vertical axes (in rows)
int width;                         width of horiz. axis (in columns)
WINDOW wn;                         window structure
```

RETURNS

= Ø if the origin of the axes does not fall within window or the height of the axes does not fit within window

= 1 otherwise.

RELATED FUNCTIONS

v_bar()

CAUTIONS

In constructing graphs, the beginning locations of bars should not be in the same row or column as the axes, but in the adjacent locations.

NAME

v_bar.c -- draws a horizontal or vertical bar

DATE: April 17, 1986

USAGE

Draw bars for bar graphs.

FUNCTION

Calls set_wn to define and fill a window of specified dimensions and location with a specified attribute-character pair.

The bars are drawn within window "wn", specified as a parameter in the call. The bar location is measured relative to the window origin.

By properly defining row_size and col_size, either horizontal or vertical bars can be drawn.

Borders can be drawn around bars by defining bdr struct. If no borders are desired, make call with BDR_ØP (NULL border).

This routine supports both logical and physical video attributes.

MSDOS Systems:

The block-graphic shading characters useful for filling in bars are #defined in "computer.h". The available shades are: LIGHT_SHADE, MEDIUM_SHADE, DARK_SHADE, SOLID.

CALL

```
v_bar(row_size, col_size, r_begin, c_begin, ch, attrib, &wn, &bdr)
int row_size;                      height of bar measured in rows
int col_size;                      width of bar measured in columns
int r_begin;                       bottom row of bar
int c_begin;                       left hand column of row
char ch;                           character with which to fill bar
char attrib;                       attribute with which to fill bar
WINDOW wn;                         window structure
BORDER bdr;                        border structure
```

RETURNS

= Ø if bar dimensions are inconsistent or location is off the window;

= 1 otherwise.

RELATED FUNCTIONS

v_axes( )

(Continued)

v_bar (continued)

CAUTIONS

In constructing graphs, the beginning locations of bars should not be in the same row or column as the axes, but in the adjacent locations. Axes are drawn with REVERSE or LREVERSE video spaces, so they entirely fill the line and columns designated as the origins of the axes.

Accuracy of bar graphs is limited by the number of discrete steps from the origin to the end of the bar. Because there are three times as many columns as rows on the video, greater accuracy can be achieved with horizontal bar graphs.

NAME

v_border.c -- draws border on designated window

DATE: October 23, 1985

USAGE

Used to put borders on windows.

FUNCTION

Borders defined by a specified BORDER structure are used to place borders on windows. Standard borders are defined in "window.h" include file.

NOTE WELL: the border attribute specified in the BORDER structure is NOT used. This attribute was used in earlier functions of Windows for C, but v_border( ) uses the attribute defined in the WINDOW structure, wn.bdratt. This allows all information on the window and border to be stored in the WINDOW structure. The same border characters can be conveniently used in different windows with different attributes.

Pointers to the standard borders are #defined in wfc_defs.h The borders and their definitions are:

BORDER	#define bdrp	BORDER STYLE
bdr_Ø	BDR_ØP	No border
bdr_ln	BDR_LNP	Single line border
bdr_dln	BDR_DLNP	Double line border
bdr_rev	BDR_REVP	Reverse border
bdr_dot	BDR_DOTP	Dot border

If the wn.name element of the referenced window structure is not NULL, the name string pointed to by this element is written at the top left corner of the border, beginning in column 1 of the FULL dimension window.

If the passed pointer to a border structure is NULLP, a return is made without drawing a border.

A return without drawing a border is also made if border pointer BDR_ØP is specified. Border bdr_Ø is a null border, used if no border is desired. Routine checks for a NULL (Ø) character in bdr.h_ch. If found, routine returns without drawing a border.

If no border is drawn, no window name is written, even if wn.wname is not NULL.

The border is drawn around the FULL dimension of the window, regardless of the current dimension.

All the values in the window structure are preserved.

(Continued)

v_border (continued)

CALL

```
void v_border(&wn, &bdr)
WINDOW wn;                          window structure
BORDER bdr;                         pointer to border structure
```

RETURNS

None

CAUTIONS

Include "window.h" in main program if standard borders or windows are to be used.

The border attribute is contained in wn.bdratt. bdr.batt is not used.

NAME

v_ch.c -- writes a single character to a window

DATE: October 24, 1985

USAGE

To write a character to a window.

FUNCTION

Writes the specified character to the virtual cursor position in the window using the attribute specified in the window structure.

If virtual cursor advance is turned off (see v_st_rw()), the virtual cursor is unchanged.

If virtual cursor advance is turned on:

> The virtual cursor position is advanced. If the virtual cursor is moved beyond the end of a row, it will be placed at the first column of the next row.
>
> If the character is written to the last position in the window, the virtual cursor will be placed in the row below the window so that adj_cs() can detect that the window is full. A return of Ø will be made.

If the present virtual cursor position is not within the window, the character will not be written.

CALL

```
v_ch(ch, &wn)
int ch;                    character to be written
WINDOW wn;                 pointer to window structure
```

RETURNS

= Ø if window is filled

= 1 otherwise

CAUTIONS

None

NAME

v_co.c -- puts a column of attribute-char's to a window

DATE: October 24, 1985

USAGE

To draw borders on windows, to fill windows used as bars, to draw vertical lines.

FUNCTION

Prints q characters in column format on wn, starting at the position of the virtual cursor. Advances the virtual cursor. Will move to the next column if necessary to fit q char.

If q exceeds the space remaining in the window, output will stop when the bottom right position of the window is filled, and the virtual cursor will be placed in the row below the window so that adj_cs( ) can detect that the window is full.

Uses function v_qch.

Operation of this function is not affected by the setting of the options parameter, wn.wrap.

CALL

```
v_co(ch, q, &wn)
char ch;                    character to use
int q;                      number of characters to put
WINDOW wn;                  window structure
```

RETURNS

= Ø if window filled;

= 1 otherwise.

RELATED FUNCTIONS

v_rw( )

CAUTIONS

None

NAME

v_file.c -- displays a memory file in a window.

DATE: October 15, 1985

USAGE

Used to view a memory file through a window.

FUNCTION

This function displays a memory file in a window. Information on the memory file is contained in an FREC structure. This structure, called a memory file record, must have been initialized, using def_fr(), and information placed in it using di_file() or sti_file(). The window member wn.frp must be set to point to the memory file record before calling v_file(). See the text chapters for more information.

This function:

Calls set_wn() if the window is not set on the screen.

Puts lines to the window from the designated file, starting from the location in the file indicated by fr.wfr and fr.wfc (the row and column in the file where the upper left-hand corner of the window is located).

Calls v_st_nop() to put the lines to the window until the window is full or the end of file is reached.

Top and bottom of file messages can optionally be displayed. The default is to have the messages displayed. The display is controlled by the global variable tbf_msg. You can turn off the message by direct assignment

```
tbf_msg = OFF;
```

or use the system macro provided:

```
s_tbfmsg(OFF);
```

Source for this file is provided on the system diskette.

DISCUSSION

v_file() has the capability of starting the display of file lines in the window from different points in the file, effectively allowing the origin of the window in the file to be changed. The starting point of the transfer is determined by the values of wfr and wfc in the memory file record pointed to by wn.frp. Thus, for example, the file can be scrolled down one line by incrementing wn.frp->wfr and calling v_file(). This capability is used in vs_file(), which provides for displaying and scrolling of a file.

(Continued)

v_file (continued)

CALL

```
void v_file(&wn)
WINDOW wn;                          window structure
```

RETURNS

None

RELATED FUNCTIONS

```
def_fr()
di_file()
sti_file()
vs__file()
scrl_file()
```

CAUTIONS

Prior to using this function, a memory file record must be established, lines placed in the file, and window member wn.frp must be pointed to the memory file record.

NAME

v_fst.c -- puts full character string to video window

DATE: October 24, 1985

USAGE

A general purpose routine for putting full strings to video windows. This function differs from v_st() in that it will write to a window until the last character has been written, even if this involves scrolling the beginning of the string off the top of the window. This is useful to insure that a full string will be written in circumstances where the bottom of the window might be reached in the middle of the string.

FUNCTION

Scrolling is enabled for the duration of this call, regardless of the setting of wn.scr_q. v_st() is called repeatedly (if necessary) until the end of string is reached.

SEE CAUTIONS BELOW.

CALL

```
void v_fst(st, &wn)
char *st;                       string to be put
WINDOW wn;                      window structure
```

RETURNS

None (end of string will always be reached).

CAUTIONS

Will scroll the beginning of the string off the top of the window if this is required to provide space in the window for the last character.

Does not handle tabs, backspaces, or any special characters other than newlines (ØAH). These must be processed by a separate subroutine (not provided).

The window can be set to disable cursor advance and disable automatic clearing to end of row. These options are primarily intended to improve the usefulness of v_st() for maintaining status lines and other situations where output is written to the same place repetitively. They should not be invoked when calling this function.

NAME

v_mov.c -- moves info between video buffer and "video" string

DATE: October 24, 1985

USAGE

To transfer character-attribute information from a window to a string in user memory and also to restore previously stored information to a window. This routine is intended primarily as a subroutine for higher-level window transferring routines, such as sav_wi(), unsav_wi(), etc. Memory management is not automatic with this routine.

FUNCTION

This function operates within the working dimensions of a window (see dim_wn()).

Seven different "parts" of a window can be moved with this function: CH, ENDROW, ROW, ENDCOL, COL, ENDWIND, and WIND (see size_wn() for definitions). The values of the window "parts", ENDROW, etc., are #defined in wfc_defs.h

The direction of move parameter in the call determines whether the data is moved out from the screen to a string (direct = OUT) or in to the screen from a string (direct = IN).

The values of IN and OUT are #defined in wfc_defs.h.

CALL

```
v_mov(vst, &wn, part, direct)
char vst[];                     string for video screen data
WINDOW wn;                      window structure
int part;                       part-of-window parameter
int direct;                     direction-of-move parameter
```

RETURNS

= Ø if invalid direction specified

= number of bytes transferred if successful.

(Continued)

v_mov (continued)

CAUTIONS

A "video" string consists of character-attribute pairs and does not have a terminating '\0'. It cannot be manipulated using standard library string functions (strlen(), etc.).

Memory management is not automatic. The size of vst must be sufficient to hold the amount of information to be moved from the screen. No check is made on the size of the string. Use size_wn() to determine the required size of vst.

Changing the window between IN and OUT moves can cause problems. On IN moves, the length of the string to move is calculated from the dimensions of the window and the window part specified. If no changes are made in the values of the elements of the window structure between the OUT and IN movements, and the part parameter is kept fixed, no problems will arise. Changing the window dimensions or the location of the virtual cursor between IN and OUT moves may cause problems. For instance if a move OUT for part ENDWIND is made and then wn.r is decremented, a move IN for part ENDWIND will move more bytes of data than were placed in vst by the OUT move (because the number of bytes to the end of the window was increased when wn.r was decreased).

NAME

v_mova.c -- moves character info between the screen and a standard ASCII string

DATE: October 15, 1985

USAGE

To transfer character information from a window to a standard ASCII string in user memory and also to restore previously stored information to a window.

FUNCTION

The character information is read from the screen and stored in the specified string buffer. A terminating '\Ø' is placed at the end of the string.

This function operates within the working dimensions of a window (see dim_wn()).

Seven different "parts" of a window can be moved with this function: CH, ENDROW, ROW, ENDCOL, COL, ENDWIND, and WIND (see size_wn() for definitions). The values of the window "parts", ENDROW, etc., are #defined in wfc_defs.h.

The direction of move parameter in the call determines whether the data is moved out from the screen to a string (direct = OUT) or in to the screen from a string (direct = IN).

The values of IN and OUT are #defined in wfc_defs.h.

Memory for a working buffer is allocated by this function and will be released on exit.

CALL

```
v_mova(st, &wn, part, direct)
char st[];                      standard ASCII string
WINDOW wn;                      window structure
int part;                       part-of-window parameter
int direct;                     direction-of-move parameter
```

RETURNS

= -1 if error; check _wn_err code for more information

= number of bytes transferred if successful.

ERROR HANDLING

If insufficient memory is available to allocate a working buffer, _wn_err is set to MEMLACK.

(Continued)

v_mova (continued)

CAUTIONS

Memory management is not automatic. The size of st must be sufficient to hold the amount of information to be moved from the screen regen buffer. No check is made on the size of the string. Use size_wn( ) to determine the required size of st.

Changing the window between IN and OUT moves can cause problems. On IN moves, the length of the string to move is calculated from the dimensions of the window and the window part specified. If no changes are made in the values of the elements of the window structure between the OUT and IN movements, and the part parameter is kept fixed, no problems will arise. Changing the window dimensions or the location of the virtual cursor between IN and OUT moves may cause problems. For instance if a move OUT for part ENDWIND is made and then wn.r is decremented, a move IN for part ENDWIND will move more bytes of data than were placed in st by the OUT move (because the number of bytes to the end of the window was increased when wn.r was decreased).

NAME

v_natt.c -- sets a new attribute to a part of a window

DATE: October 24, 1985

USAGE

To change the attribute value of a section of a window without affecting the text.

FUNCTION

Calls v_mov() to move a section of a window to a "video string" (vst), change the attribute, and move it back to the screen.

This function operates within the working dimensions of a window (see dim_wn()).

Seven different "parts" of a window can be changed with this function: CH, ENDROW, ROW, COL, ENDCOL, ENDWIND, and WIND (see size_wn() for definitions). The values of the window "parts", ENDROW, etc., are #defined in wfc_defs.h.

CALL

```
v_natt(att, part, &wn)
char att;                       attribute byte
int part;                       part-of-window parameter
WINDOW wn;                      window structure
```

RETURNS

= 0 if successful

= -1 if insufficient memory to save window part; _wn_err will be set to MEMLACK.

CAUTIONS

None

NAME

v_plst.c -- puts character string to window beginning at the specified row and column.

DATE: October 24, 1985

USAGE

A general purpose routine for putting strings to video windows at the specified row and column. See cautions for limitations.

FUNCTION

The virtual cursor is moved to the specified row and column and v_st() is called. The virtual cursor remains where v_st() leaves it.

If the column is specified as CENTER_TXT, the string will be centered within the window if possible. If the string is too long to fit within the window, the string will start in the first column. CENTER_TXT is #defined in wfc_defs.h.

If the column is specified as LEFT_TXT, the string will be left justified within the window. LEFT_TXT is #defined in wfc_defs.h.

If the column is specified as RIGHT_TXT, the string will be right justified within the window. If the string is too long to fit within the window, the string will start in the first column. RIGHT_TXT is #defined in wfc_defs.h.

All of the bit switch options controlling string output are observed. See v_st_rw() for more information.

See v_st() for more detailed information.

CALL

```
char *v_plst(row, col, st, &wn)
int row;                        row where string is to begin
int col;                        column where string is to begin
char *st;                       string to be put
WINDOW wn;                      window structure
```

RETURNS

= Pointer to next character in string if writing stops before end of string is reached.

= NULLP if end of string is reached.

(Continued)

v_plst (continued)

CAUTIONS

Does not handle tabs, backspaces, or any special characters other than newlines (ØAH). These must be processed by a separate subroutine (not provided).

Disabling of cursor advance and disabling automatic clearing to end of row are primarily intended to improve the usefulness of v_plst(), which calls v_st(), for maintaining status lines and other situations where output is written to the same place repetitively. Disabling these functions will cause problems in many normal output situations. Exercise care.

NAME

v_printf.c -- performs formatted output to windows

DATE: October 24, 1985

USAGE

A general purpose routine for performing formatted output conversions and displaying the resulting string in the window. See cautions.

FUNCTION

Allocates a buffer of _vpstlen bytes to hold the formatted string. _vpstlen is a global variable that is initialized to 133. It may be changed by the user at any time.

Calls sprintf( ) to format the variables according to the format control string. The result is placed in the previously allocated buffer.

Calls v_st( ) to display the string in the specified window. If the string will not fit in the window, the remainder of the string is lost.

The buffer is freed.

All of the bit switch options controlling string output are observed. See v_st_rw( ) for more information.

CALL

```
void v_printf(&wn, fmt, args...)
WINDOW wn;                      window structure
char *fmt;                      format control string
---- args;                      list of arguments to be formatted
```

RETURNS

None

(Continued)

v_printf (continued)

CAUTIONS

A maximum a 20 integer arguments, or 10 long integer arguments, or 5 single precision or double precision floats are allowed in the call.

The allocated buffer must be large enough to hold the resulting formatted string. It may be necessary to change the value of _vpstlen.

Use of this routine will link in sprintf() and all of its underlying routines. This may result in significantly larger executable files.

Printing floating numbers may require that you link in the floating point math library or some floating point object module. Consult your compiler manual for the specifics.

Disabling of cursor advance and disabling automatic clearing to end of row are primarily intended to improve the usefulness of v_printf(), which calls v_st(), for maintaining status lines and other situations where output is written to the same place repetitively. Disabling these functions will cause problems in many normal output situations. Exercise care.

NAME

v_qch.c -- writes q character attribute pairs to a window

DATE: October 24, 1985

USAGE

To write q character-attribute pairs to a window.

FUNCTION

The value of q is checked. If negative, a return is made without writing any characters.

Obtains address of current virtual cursor position and puts q characters to that address. Does not observe window boundaries.

The attribute of the displayed characters will be the value in wn.att.

Does not advance the virtual cursor.

CALL

```
v_qch(ch, q, &wn)
char ch;                    character to write
int q;                      number of characters to write
WINDOW wn;                  window structure
```

RETURNS

= 1 if q >=0

= -1 if q < 0

CAUTIONS

The value of q must be checked to insure that it does not exceed the remaining columns in the window, else writing will occur beyond the window boundaries.

Does not handle any special characters as command characters; does not advance cursor.

NAME

v_rw.c -- puts rows of attribute-char's to a video window

DATE: October 24, 1985

USAGE

To place a row of characters to a window.

FUNCTION

Prints q characters in row format on wn, starting at present position. Advances virtual cursor. Will move to next row if necessary to fit q char.

If q exceeds the space remaining in the window, output will stop when bottom right position of the window is filled, and the virtual cursor will be placed in the row below the window so that adj_cs( ) can detect that the window is full.

CALL

```
v_rw(ch, q, &wn)
char ch;                    character to write
int q;                      quantity of char to write
WINDOW wn;                  window structure
```

RETURNS

= Ø if window filled;

= 1 otherwise.

CAUTIONS

None

uses v_qch

NAME

v_st.c -- puts character string to video window

DATE: October 24, 1985

USAGE

A general purpose routine for putting strings to video windows. See cautions for limitations.

FUNCTION

This function writes the specified string within the working dimensions of the window, using v_st_rw( ).

AUTO SCROLLING

Scrolling behavior is controlled by window member wn.scr_q, which specifies the number of lines by which text is to be scrolled when v_st( ) attempts to write to the bottom of a full window. Scrolling can only occur if automatic updating of the virtual cursor is enabled (see below).

The function v_st( ) will stop writing when it reaches the end of the window, assuming it started writing prior to the end. The virtual cursor will be placed in the first column position of the first row below the bottom of the window. The window is defined as "full" when the virtual cursor is in this position. The function adj_cs( ) returns -1 when a window is full, allowing this condition to be determined.

If v_st( ) is called when the window is full, the window text and the virtual cursor are scrolled upward by the number of lines specified by wn.scr_q, and then v_st( ) writes the specified string until the end of the string is reached, or the window is full.

When v_st( ) reaches the end of a window without writing an entire string, it returns a pointer to the next character in the string. If the end of the string is reached, NULLP is returned. Together with v_st( )'s ability to scroll text up automatically, this feature permits you to control what happens when a string cannot be fully written before filling the window.

(Continued)

\n works

v_st (continued)

The following code fragment shows how a string can be written so that the window line will be scrolled just one line upward in an attempt to write a string. If it will not fit in the second line, additional scrolling will not occur.

```
write_st(st, wnp)
char *st;
WINDOWPTR wnp;
{
    char *st1, *v_st();
    int qt;

    qt = wnp->scr_q;                        /save current value          /
    wnp->scr_q = 1;                         /scroll only 1 line         /
    if((st1 = v_st(st, wnp)) != NULLP)      /write string               /
        v_st(st1, wnp);                     /write overflow             /
    wnp->scr_q = qt;                        /restore original value     /
    return;
}
```

If v_st() reaches the end of the string on the first try, the condition test will fail; otherwise v_st() will be called again for st1, which starts where v_st() left off writing on the first attempt.

BIT-SWITCH OPTIONS

The member wn.wrap, which controlled only word wrap (and thereby got its name) in early versions of Windows for C, now controls a number of options via bit switches. Values are #defined in wfc_defs.h to allow the switches to be set.

Option	Switch Value	
Word wrap enabled	WRAP	(1)
Auto clear to row-end disabled	NO_CLEAR	(2)
Virtual cursor advance disabled	NO_CS_ADV	(4)
Place screen cursor at v. cursor	PL_CSR	(8)

When these switches are not set, the opposite of the actions is carried out by v_st(). The switches may be set in any combination. Functions defs_wn() and def_wn() set word wrap ON and all other options OFF.

Macros are #defined in wfc_defs.h to aid in setting these bit-switches.

```
sw_wwrap(state, &wn) -- sets word wrap switch to the specified state
sw_cleor(state, &wn) -- sets auto clear to row-end switch to the specified
                        state
sw_csadv(state, &wn) -- sets the virtual cursor advance switch to the
                        specified state
sw_plcsr(state, &wn) -- sets the cursor placement switch to the specified
                        state
```

(Continued)

v_st (continued)

CALL

```
char *v_st(st, &wn)
char *st;                       string to be put
WINDOW wn;                      window structure
```

RETURNS

= Pointer to next character in string if writing stops before end of string is reached.

= NULLP if end of string is reached.

CAUTIONS

Does not handle tabs, backspaces, or any special characters other than newlines (ØAH). These must be processed by a separate subroutine (not provided).

Disabling of cursor advance and disabling automatic clearing to end of row are primarily intended to improve the usefulness of v_st(), which calls v_st_rw(), for maintaining status lines and other situations where output is written to the same place repetitively. Disabling these functions will cause problems in many normal output situations. Exercise care.

NAME

v_st_nop.c -- puts string to window in specific format, no options

DATE: February 4, 1986

USAGE

To copy q characters of a string to one row of window. Intended primarily as a subroutine in v_file(). No options are allowed. No interpretation of special character is done.

FUNCTION

This function makes no checks at all. It writes q characters from the specified string to a row in the specified window, beginning at the current position of the virtual cursor, fills rest of the row with spaces, and advances the virtual cursor to next line.

CALL

```
void v_st_nop(st, q, &wn)
char *st;                    string to be written
int q;                       number of char to be written
WINDOW wn;                   window structure
```

RETURNS

None

CAUTIONS

No checks of any kind are made. If q exceeds the remaining spaces in a window row, the window boundaries will be exceeded.

The bit-switch settings in wn.wrap are not observed.

When running on UNIX systems, does not restore screen cursor to its original position.

NAME

v_st_rw.c -- Puts string to a row of a window

DATE: October 24, 1985

USAGE

To copy a string to one row of video window, with word wrap if specified. Intended primarily as a subroutine to more general string-to-window functions, such as v_st(). This function handles newlines and detects the NULL terminator, but does not handle tabs or other special characters.

Logical attributes can be used.

FUNCTION

Transfers q characters of a string to the video regen buffer, beginning at the address corresponding to the location of the virtual cursor in the specified window.

The attribute of the displayed string will be the value in wn.att. Logical attributes are implemented. If the globally known switch, _lattsw, is set, wn_att is interpreted as a logical attribute. See the text discussion of logical attributes for further details.

Generally, q will be the number of columns to the right margin of the window. If the number of characters put to a row is less than q (because of a newline, end of string, or word wrap), the remaining screen positions can be automatically cleared with spaces. This has the effect of erasing the previous contents of the row before writing the current string. If this is not desired, a bit switch in wn.wrap can be set to disable automatic clearing (see below).

BIT-SWITCH OPTIONS:

The parameter wn.wrap, which controlled only word wrap (and thereby got its name) in early versions of Windows for C, now controls a number of options via bit switches in the parameter. Values are #defined in wfc_defs.h to allow the switches to be set.

Option	Switch Value
Word wrap enabled	WRAP (1)
Auto clear to row-end disabled	NO_CLEAR (2)
Virtual cursor advance disabled	NO_CS_ADV (4)
Place screen cursor at vir. cursor	PL_CSR (8)

When none of these switches are set (wn.wrap = Ø or NO_WRAP), the opposite of the actions listed above is carried out by v_st_rw().

(Continued)

v_st_rw (continued)

Macros are #defined in wfc_defs.h to aid in setting these bit-switches.

```
sw_wwrap(state, &wn) -- sets word wrap switch to the specified state
sw_cleor(state, &wn) -- sets auto clear to row-end switch to the specified
                        state
sw_csadv(state, &wn) -- sets the virtual cursor advance switch to the
                        specified state
sw_plcsr(state, &wn) -- sets the cursor placement switch to the specified
                        state
```

See CAUTIONS about improper uses of disabling switches.

GENERAL:

If word wrap is specified, v_st_rw( ) insures that next character in the string after the last one printed is a space or a NULL (indicating end of string).

If a string fills an entire row without any spaces, word-wrap is disabled for that row.

If the virtual cursor advance is enabled (default):

If a linefeed is detected, it is not printed, but wn.r is incremented and wn.c set = Ø. No further characters are printed. The next character is checked for end of string indicator (NULL); if found, return is set to so indicate.

If end of string is not reached, wn.r is incremented and wn.c set = Ø. The next character is checked for end of string indicator (NULL); if found, return is set to so indicate.

If end of string is reached (NULL detected), wn.c is incremented by the number of characters put from string to video.

If no characters can be put to video, or q = Ø, a linefeed/carriage-return is executed.

If the virtual cursor advance is disabled:

If a newline or end of string indicator (NULL) is detected, no further characters are printed. By definition, the virtual cursor is not moved.

The function always returns NULLP.

(Continued)

v_st_rw (continued)

CALL

```
char *v_st_rw(st, q, &wn)
char *st;                       string to be put
int q;                          number of char in string, maximum equals co_q
                                remaining in row.
WINDOW wn;                      window structure
```

RETURNS

If virtual cursor advance is disabled

= NULLP regardless of whether or not the end of string was reached.

If virtual cursor advance enabled (default):

= pointer to next character in string if terminal '\Ø' is not detected.

= NULLP if terminal '\Ø' is detected

CAUTIONS

Value of q must be checked prior to call to insure that string will fit on current row, otherwise string will be written beyond window boundaries.

Does not provide for tabs, backspaces, or any special characters, other than newlines (ØA).

If the last line of a window is filled on a call, the newline executed at the end of the line will result in wn.r = wn.re + 1 and wn.c = Ø, that is the virtual cursor will lie in the first row beyond the window. If v_st_rw is called in this circumstance, the row will be written below the window. The full-window condition can be determined by a call to adj_cs(wnp), which will return Ø when wn.r exceeds wn.re. Room for the next line of text can be obtained by a call to mv_rws.

Disabling of cursor advance and disabling automatic clearing to end of row are primarily intended to improve the usefulness of v_st(), which calls v_st_rw(), for maintaining status lines and other situations where output is written to the same place repetitively. Disabling these functions will cause problems in many normal output situations. Exercise care.

NAME

v_tv.c -- issues a request to Topview to update video display

DATE: February 4, 1986

USAGE

In WFC/MSDOS, to update the video display under the Topview environment. In WFC/UX, issues a call to upd_vdisp().

FUNCTION

Accepts as inputs the two positions in a window between which the screen is to be updated. The positions are specified by row and column numbers.

MSDOS Systems:

The starting and ending positions to be updated in the TopView-assigned video buffer are calculated and passed in prescribed form to the TopView Update Video Display function (INT 10H, function AH = 0FFH).

UNIX Systems:

The WFC/UX version of this function merely calls upd_vdisp().

CALL

```
void v_tv(r0, r1, c0, c1, &wn)
int r0, r1, c0, c1;          window row and column begin and end
                             positions for update
WINDOW wn;                   WINDOW structure
```

RETURNS

None

CAUTIONS

MSDOS Systems:

None

UNIX Systems:

Provided for compatibility with MSDOS version of Windows for C. The function upd_vdisp() should be used by Unix/Xenix programs.

This page intentionally left blank.

APPENDIX 3

SOURCE CODE FILES

SOURCE CODE FILES

Source code is provided on the accompanying diskette for the following library functions:

di_file( ) -- reads a disk file into memory into a sequential array.

k_vcom( ) -- translates keyboard "extended codes" into video movement commands.

menu2( ) -- pop-up menu display and selection routine

v_file( ) -- puts a file read by di_file( ) into a window for viewing.

vs_file( ) -- puts a file read by di_file( ) into a window for viewing and permits the user to scroll through it.

These functions may be modified to accommodate your information structures.

Code for other library functions may be provided on updated disks. Check your diskette.

Source code is provided on the accompanying diskette for the following demonstration programs:

demo_wn.c -- A demo program that views multiple files in several different overlapping windows.

dem_cmov.c -- A demo program that illustrates the use of windows in animation and movement.

dem_menu.c -- A demo program that illustrates the use of pop-up menus.

dem_grph.c -- A demo program that illustrates the use of windows in drawing block graphics mode bar charts.

prt_labl.c -- A demo program that illustrates the use of windows in printing texts in uncommon formats.

Source code is provided on the accompanying diskette for the following tutorials and examples:

hello_wc.c -- A tutorial for using physical color attributes.

loop.c -- A tutorial for implementing the keyboard loop function.

sti_buf.c -- An example function that can be used to add lines to an off-screen memory file.

tutor.c -- A tutorial program that illustrates some text editing capabilities using windows.

tut_help.c -- A tutorial that illustrates pop-up help files.

vmenu.c -- A tutorial program for using pop-up menus. A much simpler program than dem_menu.c.

APPENDIX 4

DEFINITIONS AND ABBREVIATIONS

DEFINITIONS AND ABBREVIATIONS

bdr = a type BORDER data structure. The typedef type-specifier BORDER is defined in "wfc_stru.h". "bdr" contains the information necessary for drawing a border on a window.

bdrp = a pointer to a type BORDER data structure BORDERPTR, a typedef type-specifier defined in "wfc_stru.h", can be used to declare "bdrp". One of the elements of "wn" is a pointer to a BORDER structure, that is, "wn.bdrp" is an element of "wn."

c = the column position of the virtual cursor in window "wn", that is "wn.c" specifies the window column-number of the virtual cursor in a window. Window column-numbers begin at the left-hand side of the window. Zero is the column-number of the first column available for output.

After a window has been "set" using function "set_wn", which draws a border and sets left and right margins for the window, the columns occupied by the border and the left margin are no longer counted in measuring the row-number, as they are no longer accessible until the window is "unset".

cl = clear, as in clear window (cl_wn).

cls = clear screen.

co = a general abbreviation for column, distinct from the column-number of the virtual cursor.

cs = the virtual cursor in a window. The virtual cursor is the position in a window at which the library functions will begin writing output. It differs from the screen cursor (abbreviated "csr", see below) that appears on the screen. It is not necessary for the screen cursor to be located at the virtual cursor position to write at that point.

csr = the screen cursor; the blinking cursor that appears on the screen. The screen cursor is distinct from the virtual cursor of a window (see above).

cwn = color window. Used to indicate functions intended primarily for use in managing window structures for color displays.

d = disk, used in naming functions that access disks.

dim = dimension.

f = file of data, which may be either on disk or in memory.

fr = a type FREC data structure. The typedef type-specifier FREC is defined in "wfc_stru.h". "fr" refers to "file-record" and contains the basic information required for **Window** library functions to access ASCII text files that have been read into memory.

i = input, used as a suffix to denote an input operation; thus "di" refers to input from disk.

k = key data, read from keyboard; sometimes combined with "i" to emphasize its input nature, as in "ki" (key input).

mv = move, as in "mv_rws" (move rows).

o = output, used a suffix to denote an output operation.

p = In general, "p" is appended to a variable to refer to a pointer to the variable in question.

pl = place, as in "pl_csr" (place cursor).

q = quantity, used as a suffix to denote the quantity of certain variables. For example, "co_q" stands for the quantity of columns in window. "q" is used in preference to "n", which is commonly used as an varying index number rather than as an indicator of fixed quantity.

r = the row position of the virtual cursor in window "wn", that is "wn.r" specifies the window row-number of the virtual cursor in a window. The window row-number is measured relative to the top of the window, not from the top of the screen. Zero is the row-number of the first row of a window available for output.

After a window has been "set" using function "set_wn", which draws a border on the window, the row occupied by the border is not counted in measuring the row-number, as it is no longer accessible until the window is "unset".

rw = a general abbreviation for row, distinct from the row-number of the virtual cursor.

sc = screen.

st = string (of characters), as in "v_st" (video output of string).

v = video. "v" refers to writing to a window on the video screen. For example, the function "v_ch" refers to writing a character to a window.

vid = video interrupt. Indicates functions that obtain information by calling the IBM video interrupt function (INT 1ØH).

vo = video out. "vo" refers to reading a character/attribute out of the video buffer. For example, the function "vo_ch" refers to reading a character from a window.

wc = window contents. Used to indicate functions that operate on the character contents only (not attributes) of window images.

wi = window image. The image on the screen within the area defined by a WINDOW structure.

wn = a type WINDOW data structure. In library functions and in text discussion, "wn" is used to refer to a window data structure declared using the typedef type-specifier WINDOW.

Members of the WINDOW structure are addressed in the form "wn.x," where "x" is one of the members. For example, "wn.rb" refers to the screen row-number (zero is the top row of the screen) of the top row of window "wn". (See the discussion in Appendix 5, **WINDOW STRUCTURES**.)

wnp = pointer to a type WINDOW data structure. "wnp" can be declared using the typedef type-specifier WINDOWPTR (defined in "wfc_stru.h").

APPENDIX 5

WINDOW AND MEMORY FILE STRUCTURES

This appendix provides a detailed guide to the structures that are used to manage windows and memory files within **Windows for C**.

WINDOW STRUCTURES

Those who are unfamiliar with the use of pointers and structures in C will need to read Chapters 5 and 6 of K&R to fully understand this section.

All **Window** functions make use of information contained in a C structure that is defined for each window. This structure contains (or refers to) all of the information needed for a window's management. Reference to the structure substitutes for long paramenter lists in parameter call. This simplifies coding, saves time, and reduces errors.

The structures that contain the information for a window are of type WINDOW. WINDOW is defined using the typedef facility of C (see K&R, pp. 140 and 200) in the wfc_stru.h #include file. This definition is reproduced here:

```
typedef struct wnd
{
   int rb;                  /*top row of window                    */
   int re;                  /*bottom row of window                 */
   int cb;                  /*left hand column of window           */
   int ce;                  /*right hand column of window          */
   int r;                   /*virtual cursor row-position          */
   int c;                   /*virtual cursor column-position       */
   char att;                /*character attribute                  */
   char page;               /*graphics-card alpha mode page #      */
   int  wrap;               /*word-wrap switch                     */
   int  location;           /*must be initialized to Ø             */
   int  scr_q;              /*max number of lines to scroll        */
   int l_mg, r_mg;          /*left and right spaces (margins) --   */
                                 /*to border                       */
   BORDERPTR bdrp;          /*pointer to border structure          */
   char setsw;              /* = 1 for INSIDE, = Ø for FULL dimen;*/
                                 /*initialize to Ø                 */
   FRECPTR frp;             /*pointer to file-record structure     */
   char *storp;             /*pointer to window storage            */
   char *userp[2];          /*2 pointers reserved for users        */
   char *wname;             /*pointer to window name               */
   char *larray;            /*pointer to logical attribute array   */
   char *pu_storp;          /*pointer to pop-up window storage     */
   char bdratt;             /*video attribute of border            */
   char popup;              /*popup switch: Ø = no popup           */
                            /*               1 = popup             */
/*  int reserv2[6];           six reserved words, not now used     */
/*  char *reserv3[4];         four reserved pointers               */
} WINDOW, *WINDOWPTR;
```

DESCRIPTION OF STRUCTURE MEMBERS

Each member of the structure WINDOW wn is described briefly below:

* The first four entries (rb, re, cb, ce) define the location and size of a window. The values of these parameters are measured in terms of screen row-numbers and column-numbers (which begin at zero in the upper left-hand corner of the screen).

* **r** and **c** define the **virtual cursor** position in the window. The values of wn.r and wn.c are measured relative to the origin of the window (its upper left-hand corner).

  The virtual cursor is the position in a window at which the library functions will begin writing output. A separate virtual cursor is defined for each window. The virtual cursor differs from the actual cursor that appears on the screen. It is not necessary for the actual cursor to be located at the virtual cursor position to write at that point.

* **page** applies only to the Color/Graphics Adapter. It refers to the memory page within the Color/Graphics Adapter card that the window is addressing. A page must be the "active" page before text written to it will appear on the screen. Upon initialization of the computer, page Ø (zero) is made the active page; thus the normal setting for wn.att will be Ø (zero).

  By setting wn.page to another page number, information can be stored for later output to the screen (remember, though, that this only applies to the Color/Graphics Adapter). The active page must be changed (using IBM BIOS interupt 1ØH, function 5 -- which can be done with **Window** function vid_int()) to display the contents of other memory pages.

* **wrap**, which controlled only word wrap (and thereby got its name) in early versions of **Windows for C** now controls a number of options via bit switches in the member. Values are #defined in wfc_defs.h to allow the switches to be set.

Option	Switch Value
Word wrap enabled	WRAP (1)
Auto clear to row end disabled	NO_CLEAR (2)
Virtual cursor advance disabled	NO_CS_ADV (4)
Place screen cursor at virtual cursor	PL_CSR (8)

  The word wrap switch specifies whether or not word wrap should be implemented for strings put to the window by the library function v_st() (video_string). If word wrap is implemented, only entire words will appear on a line. If no-wrap is implemented, contents of a string will fill entire rows of a window, without regard to whether a word is split with a part on each line.

  The auto clear to row end switch controls whether or not the row should be cleared from the last character written to the end of the row. This has the effect of erasing the previous contents of the row before writing the current string. For most applications this is the preferred setting. However, when setting up status lines where only a small section of the row is updated by a video output call, the rest of the row should not be deleted because it contains relevant information.

  The virtual cursor advance switch controls whether the virtual cursor is updated to reflect the ending location of the string just written to the window. For most applications the virtual cursor should be advanced. This ensures that subsequent writes to the window will start where the current string ended. However, in some applications this may be undesirable and can be inhibited by properly setting this switch.

The place screen cursor at virtual cursor switch controls whether the physical screen cursor is placed at the virtual cursor location after the string is written to the window. In most situations this is not necessary and may actually distract the user. However in some instances it may be advantageous to place the cursor at the virtual cursor to draw the user's attention to some important piece of information.

* **location** is reserved for use in later versions of **Windows for C**. Code is not operable for this parameter in the present version. It must be set to zero before calling video output functions, or the application program will abort with an appropriate error message.

* **scr_q** specifies whether or not upward scrolling of text is to be automatically implemented by the library function v_st(). If scr_q is greater than zero, a call to v_st() when the window is full will automatically scroll up the text by the number of rows specified in the scr_q variable.

* **l_mg** and **r_mg** specify the number of spaces of margin between text and border on the left and right sides of the window.

* **bdrp** is a pointer to a (typedef) BORDER structure, which contains the information necessary for drawing borders on a window. The BORDER structure is defined in wfc_stru.h, and a number of standard BORDER structure declarations are contained in window.h. If the standard borders (or the standard full-screen window) defined in window.h are to be used, this file should be #included following bios.h in the main program.

* **setsw** indicates whether the "working dimensions" reflect the FULL dimensions of the window, before adjustment for left and right margins and the border, or INSIDE dimensions, after these adjustments have been made. FULL and INSIDE are #defined in wfc_defs.h and have the values Ø and 1, respectively.

  The value of wn.setsw ("set switch") is modified by set_wn(), unset_wn(), and dim_wn(). Setting a window with set_wn() includes adjusting the dimensions of the window (rb, re, cb, ce) to INSIDE to reflect the space allocated to a border and to l_mg and r_mg. Function unset_wn() restores FULL dimensions. Function dim_wn() allows the working dimensions to be specified directly.

  Window functions involving movement of information to or from windows operate within the working dimensions of a window. For certain purposes FULL dimensions will be preferred to INSIDE dimensions.

* **frp** is a pointer to a (typedef) FREC (File-RECord) structure, which contains information on a data file in memory. This information is used by function v_file() to allow viewing of the file through a window. FREC is defined in wfc_stru.h. Programmers may wish to redefine FREC to contain the information necessary for their specific choice of list-handling methodologies. For more information see Chapter 6.

* **storp** is a pointer to storage for a window image. This pointer is assigned by functions that move window images to storage, such as sav_wi(), and is used to determine storage location by functions that retrieve images, such as repl_wi().

* **char *userp[2]** reserves two pointers for users. You can use these pointers to refer to information you want associated with a specific window. They can point to other structures, so there is no limit on the number of variables you can tie to a window structure. These pointers will be maintained in future revisions.

* **wname** is a pointer to the string which contains the name for the window. If this pointer is not NULL, the string containing the window name will be placed in the top border of the window starting at the left hand corner when the window is set on the screen using set_wn( ) or when a new border is drawn using v_border( ). If no border is specified, the window name will not be written to the window.

* **larray** is a pointer to a character array which maps the logical attributes for the active adapter and mode into physical attributes. Although there is a default logical attribute array, latt[], for the **Windows for C** system, a unique logical attribute array may be associated with each window. If a unique logical attribute array is specified for a window, it will be used instead of the default logical attribute array. This feature provides the flexibility to map the same logical attribute to different physical attributes for different windows.

* **pu_storp** is a pointer to storage where the underlying window image will be stored for pop-up windows. Before pop-up windows can be displayed on the screen, the underlying screen information must be saved so that it can be restored when the pop-up window is removed. This pointer is manipulated by the set_wn( ) and unset_wn( ) functions and should not be changed by the application program.

* **bdratt** defines the video attribute of the border for the window. wn.bdratt may be interpreted as a physical attribute value or a logical attibute value depending if logical or physical video attributes are in use. In earlier versions of **Windows for C** the border attribute was stored in the border structure. Although an attribute still exists in the border structure it is no longer used.

* **popup** is a switch which identifies if the window is a pop-up window or a standard window. If the window is a pop-up window, before the window can be set on the screen, the underlying information must be saved.

Additionally, the system reserves storage at the end of the present WINDOW structure for future expansion. The reserved storage is:

* **char *reserv3[4]** reserves four pointers.

* **int reserv2[1Ø]** reserves ten words.

The reserved pointers and words are not now currently used, nor is storage allocated for them in WINDOW structures.

Before using a window, values must be assigned to the members in the window structure. The preferred method of assigning values to the WINDOW structure members is to use the functions def_wn( ), defs_wn( ), or a function of your own creation. In this way future changes to the window structure are isolated from the rest of your program. Only the window assignment function need be changed and recompiled. Another method (not recommended) of initialization would be to place external or static initializations of WINDOW structures in a

single include file and #include it prior to main(). You should definitely avoid external or static initializations that are buried in subroutines.

Not all members of a window structure will be utilized in all uses. For example, if only individual strings are put to a window and the window is not used to view an entire ASCII file, the frp member will not be addressed. For those members not used in the program, the values are immaterial and initial values need not be assigned.

MEMORY FILE STRUCTURES

The basic structure used to manage memory files is a **memory file record** of type defined by the typedef specifier **FREC**. The members of this **mfile** structure are:

mfile. fn	filename (drive and path)
mfile.fmaxlines	maximum number of lines in memory file
mfile.fmaxcol	maximum number of colums in a line
mfile.farray	pointer to an array of FLINEPTRS
mfile.ln_q	will be set by di_file or sti_file
mfile.c_q	will be set by di_file or sti_file
mfile.wfr	origin of the window in file
mfile.wfc	origin of the window in file
mfile.ftabq	tab spacing, for di_file()
mfile.ib	not currently used
mfile.lbp	not currently used
mfile.lep	not currently used

DESCRIPTION OF STRUCTURE MEMBERS

mfile.fn: filename, including drive (and path, if supported by compiler and DOS and file not in default directory) of an ASCII file. The filename is specified as an argument in **def_fr().**

The filename is used by **di_file()** to read a file from disk to a memory file. If you are creating a memory file, using **sti_file()**, and intend to write the file to disk, store the intended file name in this member. (Note, a function for writing a memory file to disk is not supplied).

If you are not intending to do any disk operations on the memory file, the filename argument in **def_fr()** can be specified as **NULLP** (a null pointer).

mfile.fmaxlines: the maximum number of lines in the memory file (not including a position for an end of file marker). The value of this member is specified as an argument in **def_fr()**.

mfile.fmaxcol: the maximum number of columns allowed in a file line (not including the terminal newline and null that will be appended to each line by **di_file()** and **sti_file()**. When **di_file()** or **sti_file()** copy a string to the memory file, they will truncate the copy at **fmaxcol**; thus you can control the length of strings that will be placed in the file.

If you want to control the length of file lines but do not want to lose information by truncation, use **sti_file()** (and **di_st()**, if you are going to obtain the strings from a disk file) to place lines in the memory file. Function **sti_file()** will return a -1 when a line is truncated. You can

call **sti_file()** again to copy the following part of the string to the following line. Repeat this until the string is fully copied.

mfile.farray: A pointer to an array of pointers to FLINE structures. Each FLINE structure will contain information for one line of the memory file.

Function **def_fr()** allocates storage for the array and places a pointer to this array into **mfile.farray**. The size of the array is equal to **mfile.fmaxline** plus one additional element for an end-of-file marker. All elements of the array are initialized to NULLP. The initialization to NULLP is done because **sti_file()**, which places copies of strings into the file will attempt to free the contents of any FLINE element that does not contain a NULLP.

mfile.ln_q: the number of lines placed in the memory file by di_file() or sti_file(). This number will always be less than or equal to the **mfile.fmaxlines**. If the actual number of lines is less than **fmaxlines**, the end of file will be marked by a NULLP in the element of **farray** that follows the pointer to the last line of the file. You can check if **frow** of the memory file is the end of file by:

```
if(mfile.farray[frow] == NULLP)     /*end of file                    */
```

mfile.c_q: the maximum number of characters (excluding the newline and terminal null on each line) in any line in the memory file. This value is maintained by **di_file()** and **sti_file()**.

The values of **mfile.c_q** and **mfile.ln_q** are used by the functions (v_file(), k_vcom(), and vs_file()) that provide capability for displaying and scrolling through a memory file. They tell these functions where the row and column limits of text occur in the memory file.

mfile.wfr and **mfile.wfc**: the row and colum in the memory file where the origin of the display window will be placed by **v_file()** and **vs_file()**. Consider an example where window **wn** points to memory file **mfile**, that is **wn.frp = &mfile**. When **v_file(&wn)** is called, the window origin will be at **mfile.wfr, mfile.wfc**. To scroll the file down one line in the window, all that needs to be done is to increment **mfile.wfr** and call **v_file(&wn)**. This capability is used in **vs_file()**, which provides for displaying and scrolling of a file.

mfile.ftabq: the spacing of tab stops in a file stored on disk to be read into a memory file by **di_file()**. Some editors and MS/PCDOS expect tabs to be spaced every 8 spaces and will correctly expand tab characters (ASCII 9) embedded in file lines. The subroutine called by **di_file()** to read in file lines, **di_st()**, has the capability to expand tabs to specified tab spacing. Function **di_file()** passes **di_st()** the value of **mfile.ftabq** for the tab spacing to be used. The default value set by **def_fr()** is 8, the same as that assumed by DOS; thus standard DOS files with embedded tabs will be expanded properly.

The remaining members of the FREC structure are not used in the present implementation.

HOW LINE INFORMATION IS STORED IN A MEMORY FILE

Function **def_fr()** allocates memory for **memory file array**. This is an array of pointers to FLINE structures. A separate **FLINE** structure is allocated for each line of a memory file. The structure **FLINE fline** contains two members:

fline.line_len is an integer and holds the length of the file line (excluding the terminal newline and null).

fline.line_st is a character pointer and points to the location where the file line-string is stored.

Functions **di_file()** and **sti_file()** fill the array of FLINE pointers as they add lines to the file.

DIRECTLY ACCESSING THE STRINGS IN A MEMORY FILE

Once lines have been placed in a file, you can access them by using the system function:

```
char *file_lnp(frow, &mfile)
int frow;
FREC mfile;
```

which returns a pointer to the string that is in **frow** of the memory file. The string will be be terminated by a newline and a null.

Function **file_lnp()** is convenient for many uses, but if you intend to work intensively with memory files, you may wish to reference the file strings more directly.

For example, if you are going to reference file strings a number of times in a module, you will save code if you first set a variable to point to the array of FLINE pointers and then use this variable as the base for accessing the strings. Assume that a pointer to **FREC mfile**, **mfp**, is passed to your function, then this code fragment illustrates the approach:

```
FLINEPTR *farray;
char *string;

farray = mfp->frp;
string = farray[frow]->line_st;
```

Variable **string** will point to the string that is in **frow** of the memory file.

CODING EXAMPLE

The following code reads in a file from disk named "junk.txt". The maximum number of lines in the file is 100 and the maximum number of characters in a line is 80. After the memory file is established, lines of the file are displayed in the full-screen window, using v_st(), until the window is full or the end of file is reached. Before returning, memory allocated for the contents of the file is freed.

```
void view_file(mfp)
FRECPTR mfp;
{
    FLINEPTR *linep;
    int line_len;
    char *line_st;
    int i = 0;

    def_fr(mfp, "junk.dat", 100, 80);     /*initialize FREC                */
    di_file(mfp);                         /*read in file                   */
    linep = mfp->farray;                  /*avoid indirection              */
    while (linep[i] != NULLP)
    {
        v_st(linep[i]->line_st, &wn0);
        if( ! adj_cs(&wn0))               /*window is full --              */
            break;                        /*so break out of loop           */
        i++;                              /*advance to next pointer        */
    }
    free_file(mfp);                       /*free allocated memory          */
    return;
}
```

This page intentionally left blank.

APPENDIX 6

ERROR CODES AND ERROR HANDLING

ERROR CODES AND ERROR HANDLING

ERROR HANDLING

All of the high level functions of **Windows for C** will signal an error condition on return, generally by returning a Ø or a -1. This allows you to test for an error by, for example:

```
if(set_wn(&wn) == Ø)
     err_proc();                          /*process the error      */
```

The Global Error Variable: _wn_err

Whenever an error return is made, you can check the global error code variable, **_wn_err**, to identify the root cause of the error. Any function of **Windows for C** that detects an error will place an error code in **_wn_err**. See Table A6.1 for a listing of the error codes.

At the start of the program, **_wn_err** is initialized to zero. Functions detecting an error overwrite the current value of **_wn_err** with the appropriate error code. If you want **_wn_err** to equal zero after you have processed your error, you must make this assignment.

Memory Management Functions: get_mem() and free_mem()

Special memory management functions are provided, **get_mem()** and **free_mem()**, which will set the global error code, **_wn_err**, when an error is detected.

The function **get_mem()** calls the function malloc() in the C library supplied with your compiler. If there is insufficient memory to fulfill the request, **get_mem()** returns a NULLP and sets **_wn_err** to MEMLACK.

The function **free_mem()** calls the function free() in the C library supplied with your compiler. If your version of free() supports an error return and an error is detected, **free_mem()** returns Ø and sets **_wn_err** to BADHEAP. Not all compiler versions of free() support an error return. Check your compiler documentation. If free() does not support an error return, **free_mem()** always returns a 1.

FATAL ERRORS

Windows for C has one error that will cause the application program to write an error message and return to the operating system level.

Error Message

The window element wn.location must be Ø

Cause

Attempting to write to or read from a window whose structure member **wn.location** is not equal to zero.

All window structures must have the member **wn.location** set equal to zero. All functions of **Windows for C** will initialize window structures in this manner. If **wn.location** becomes non-zero it will be as a result of improper explicit initialization by the programmer or overwriting memory as a result

of using a stray pointer or accessing an array beyond its bounds. No routine of **Windows for C** will change **wn.location**.

NON-FATAL ERRORS

Handling of non-fatal errors is your responsibility. Based upon program context and the code in **_wn_err**, you must determine whether to proceed with the program, invoke an error handling routine, or abort the program. Table A6.1 lists all error codes assigned to **_wn_err**.

Table A6.1: **Error Codes**[1]

Error Condition	Error Code Value	Comments
MEMLACK	1	Insufficient memory
BADHEAP	2	Memory corrupted
READERR	3	Error reading file
BAD_EOF	4	Unexpected End of File
BADPARM	5	Bad parameter value
ERR_OPEN	6	Error opening file
ERR_CLOSE	7	Error closing file
WRITEERR	8	Error writing to file
DISKFULL	9	Full disk
FILETOOBIG	10	Too many lines in file
BADWNCOORD	11	Invalid window coordinates

(1) Error codes 1 - 9 are #defined in wn_errd.h.
Error codes 10 - 19 are reserved for future use by **Windows for C**
Error codes 20 - 33 are reserved for use by **Windows for Data**

APPENDIX 7

SYSTEM DISKETTE FILES

The following files are on the system diskette for **Windows for C**.

READ.AAA	Contains information on current version. Read.
TEST.ADR	Names and addresses for prt_labl demo program.
CL.BAT	Example batch file for compiling and linking a program.
DEMO_WN.C	Source code for multiple files in multiple windows demo program.
DEM_CMOV.C	Source code for block graphics animation demo program.
DEM_GRPH.C	Source code for bar graph demo program.
DEM_MENU.C	Source code for the pop-up menu demo program.
DI_FILE.C	Source code for the **WFC** library function **di_file()**.
HELLO_WC.C	Source code for the color version of "hello world".
K_VCOM.C	Source code for the **WFC** library function **k_vcom()**.
LOOP.C	Source code for the keyboard loop tutorial program.
MENU2.C	Source code for the **WFC** library function **menu2()**.
PRT_LABL.C	Source code for the label printing demo program.
SET_MODE.C	Source code for the video mode switching tutorial program.
STI_BUF.C	Source code for the off-screen buffer tutorial function.
TUTOR.C	Source code for the "typing tutor" demo program.
TUT_HELP.C	Source code for the pop-up help file tutorial program.
U_INIT.C	Source code for the **WFC** library function **u_init()**.
VMENU.C	Source code for the pop-up menu tutorial program.
VS_FILE.C	Source code for the **WFC** library function **vs_file()**.
V_FILE.C	Source code for the **WFC** library function **v_file()**.
MENU.DEM	ASCII file for menu choices used by dem_menu.
GRPH_HLP.DOC	ASCII file used by dem_grph.
MENU.DOC	ASCII file used by dem_menu.
MENU_HLP.DOC	ASCII file used by dem_menu.
VERSION.DOC	Contains **WFC** version history information.
WFC_FLY.DOC	ASCII file used by demo_wn.
WN_HELP.DOC	ASCII file used by demo_wn.
ATT_GLOB.H	Include file for globally assigned logical video attributes.
BIOS.H	Top level include file for **Windows for C**.
COMPUTER.H	Include file of computer specific definitions.
DEF_ATT.H	Include file of logical video attribute definitions.
LABELS.H	Include file for prt_labl. Sets placement of labels.
LABELS2.H	Include file for 2 across labels.
LABELS3.H	Include file for 3 across labels.
LLIST.H	Include file of linked list definitions.
VEXTERN.H	Include file of external declarations.
WFC_COMP.H	Include file of compiler specific definitions.
WFC_DEFS.H	Include file of **WFC** specific definitions.
WINDOW.H	Include file of **WFC** global variable declarations.
WN_ERRD.H	Include file of **WFC** error definitions.
DEMO_WN.PIF	TopView/MS Windows Program Interface File for demo_wn.
DEM_MENU.PIF	TopView/MS Windows Program Interface File for dem_menu.
HELP.TXT	ASCII file for pop-up help tutorial program, tut_help.
VMENU.TXT	ASCII file for menu choices used by vmenu.

Library files may be included on the system diskette or be provided on a separate diskette. The names and extensions depend upon the conventions of the individual compiler. Each library name begins with **wn**, and has the extension assigned by the compiler to its libraries. For those compilers having more than one memory model, following **wn** will be a letter or letters identifying the memory model. For example, the library for the large program model of Lattice C is wnp.lib.

A

Abbreviations 3-7, A4-2 thru A4-4
ADDR 8-13
ADDR structure 8-13
ADDRPTR 8-13
Advanced topics 8-3 thru 8-16
Allocating memory 8-6
ALL_ROWS 6-10, 7-8
Alternative display adapters 8-13
Animation 8-10
    demonstration 8-10
Applications,
    v_mova() 8-7
    WFC 8-15
 attcolq 5-7
Attribute 3-11
    base state 3-11
    changing 8-11
    highlighting 8-11
    highlighting specified number of
        characters 8-11
    logical 3-24
    physical 3-24
    monochrome 5-9
Attribute byte 3-11, 5-3
_attrowq 5-7
att_glob.h 5-4, 5-5, 5-6, T-4,
    A1-14 thru A1-15
ATT_LOGIC 3-12, 5-8
Auto-clear-to-end-of-row 3-10, 8-7
Auto-scroll 3-9
Automatic virtual cursor advance 3-10

B

Background color 5-3, 5-10
Bar graph 8-12
    horizontal 8-12
    vertical 8-12
Batch file for compiling and
    linking 2-5
BDR_0P 3-5
BDR_DLNP 3-5
BDR_LNP 3-5
BDR_REVP 3-5
bell() 3-20, A2-14
Bell, ring 3-20
bios.h 2-5, 3-4, 3-12, 3-26, 5-8,
    5-10, 5-12, A1-2 thru A1-3
BLACK 5-10
BLINK 3-11, 3-13, 5-9, 5-10
Block graphics character 8-10
BLUE 5-4, 5-10
BORDER 3-8
Border
    changing attributes 3-13
    changing color 5-11
    color 5-3
    structure 3-8
    types 3-5
BROWN 5-10
Building new functions 8-16
BW40 5-12
BW80 5-12

C

Calling a menu 7-5, 7-6
CENTER_TXT 3-19
CGA See Color Graphics Adapter
CH 8-5, 8-11
Changes
    version 2.0 to 2.1 C-7
    version 2.2 to 3.1 C-5 thru C-7
    version 3.1 to 4.0 C-1 thru C-5
Changing
    attributes 8-11
    border attributes 3-13
    character attributes 3-20
    special options 3-10
    window attributes 3-13
    window defaults 3-7
    window margins 3-9
    working dimensions 3-14
Character
    block graphics 8-10
    changing attribute 3-20
    delete 8-7
    insert 8-7
    functions 8-14
Character output 3-19
    columns 3-20
    low level 8-14
    rows 3-20
Character-graphics animation 8-10
cl.bat 2-5
Clearing
    memory file 6-11
    screen 3-24
    windows 3-24
cls() 3-24, 5-11, A2-15
cl_att 3-24, 5-11
cl_wn() 3-24, 8-10, A2-16
CO40 5-12
CO80 5-12
COL 8-5
Color
    border 5-3
    controlling 1-4, 5-3
    foreground 5-10
    background 5-10
    screen background 3-24
    window background 5-3

window foreground 5-3
Color attributes 3-11, 5-3, 5-9
background 5-10
border 5-11
foreground 5-10
Color graphics adapter 5-9, 5-11
color_sc( ) 5-11, A2-17
color_wn( ) 5-10, 5-11, A2-18, A2-19
Column quantity macro 3-16, 8-15
col_qty( ) 3-16, 8-15
Communication display buffer 6-3
Compiling programs 2-5
computer.h 3-22, 5-10, 5-12, 6-8, T-5, A1-4 thru A1-6
Controlling
color 5-3 thru 5-13
output location 3-15
screen cursor 3-16
copy_wc( ) 8-7, 8-8, A2-20
Creating a menu memory file 7-5
csr_hide( ) 3-17, 3-18, 3-26, A2-21
csr_show( ) 3-17, 3-18, 3-26, A2-21
csr_type( ) A2-22
Cursor -- See Virtual cursor or Screen cursor
Cursor placement 3-9
CYAN 5-10
c_att( ) 5-4, 5-5, 5-6, 5-10, 5-11, A2-23

D

datt_tbl[][] 5-4, 5-5, 5-6
Declaring windows 3-5
Default window settings 3-6, T-2
Default windows 3-8, 3-9
Defining
initial value of window 3-5
menu display window 7-6
Definitions 3-7, A4-2 thru A4-4
defs_wn( ) 3-4, 3-5, 3-6, 3-7, 3-8, 3-13, 3-14, 3-26, 3-27, 5-11, 7-6, A2-24, A2-25
def_att.h 5-5, 5-6, T-4
def_fr( ) 6-3, 6-4, 6-5, 6-6, 6-7, 6-10, 7-5, 7-9, A2-26, A2-27
def_wn( ) 3-5, 3-7, 5-11, 7-6, A2-28, A2-29
Demonstration 8-3 thru 8-16
vertical-format pop-up menu T-11 thru T-13
moving images 8-10
printing side-by-side labels 8-12
viewing multiple files 8-5
demo_wfd.c 9-5
demo_wn.c 1-5, 8-5, 9-5, A3-2
dem_cmov.c 1-5, 8-10, A3-2
dem_grph.c 1-5, 8-12, A3-2
dem_menu.c 1-5, 7-7, 9-5, A3-2
Developing applications 8-15
Dimensions
FULL 3-15
INSIDE 3-15
dim_wn( ) 3-14, 3-15, 3-18, 8-6, A2-30
Direct assignment
virtual cursor 3-16
window members 3-8
Direct display windows 3-3
Diskette contents 2-3
Display adapters, alternative 8-13
Display speed 1-4
Displaying
files 1-3
help file 7-3, T-10
windows 3-14
di_file( ) 6-3, 6-5, 6-6, 6-10, 7-4, 7-5, 8-10, A2-31, A2-32, A3-2
di_st( ) A2-33, A2-34
DOS Screen
clear 3-25
restore 3-25
save 3-25
DOWN 3-21
Duplicating window structures 8-15
dup_wn( ) 8-15, A2-35, A2-36
_d_seg 8-13

E

Edit buffer 8-7
code example 8-7
Editor, line 4-8
EGA See Enhanced Graphics Adapter
ENDCOL 8-6, 8-11
ENDROW 8-5, 8-7, 8-11
ENDWIND 8-6, 8-11
Enhanced Graphics Adapter 1-5, 5-9
Error
codes A6-2 thru A6-4
fatal A6-2
non-fatal A6-3
Error exit function 8-15
Error handling 6-4, A6-2 thru A6-4
di_file( ) 6-5
sti_file( ) 6-6
Error message window 3-26
errout( ) 8-15, A2-37
ERR_OPEN 6-5
Example
changing memory file name 7-4
creating memory files 6-3
edit buffer 8-7
error message window 3-26, 3-27
graphics 5-13

help file 7-3
initializing logical attribute array 5-7
LSTATUS logical attributes 5-5
menu 7-7
moving character/attribute contents of a window 8-8
moving characters/attributes 8-8 thru 8-9
scrolling memory file 7-8
status line 3-27
windows 3-26
exit( ) 8-15
exit_key 6-8
Extended keycode 3-22

F

FAPPEND 8-8
Files
viewing in multiple windows 8-4
viewing key assignments T-9
copy window contents to 8-8
Files, multiple,
arrays of structures 8-5
demonstration 8-5
handling 8-3
one window 8-3
overlapping windows 8-4
viewing 8-3
File Line Structure A5-8
File messages, turning top/bottom off 6-8
File mode 8-8
File record
declaring 6-4
initializing 6-4
Filespec 6-4
FILETOOBIG 6-5, 6-6
file_lnp( ) 6-9, 7-8, A2-38, A5-8
FLINE 6-4
Foreground color 5-3, 5-10
Formatted output 1-4
Formatting text for printing windows 8-12
FREC 6-4, 6-5, 6-11, 7-4, 8-3, 8-4
Freeing memory file 6-11
free_file( ) 6-11, 8-3, A2-39
free_mem( ) 6-11, 8-10, A2-40
FULL 3-15, 3-18, 8-6
Full screen, writing to 3-26
Functions, building new ones 8-16
FWRITE 8-8

G

Get video buffer call 9-3
get_mem( ) 8-6, 8-7, A2-41
Globals, using and modifying 8-13
Graphics 5-12
Graphing functions 8-12
GREEN 5-5, 5-10

H

Handling multiple files 8-3 thru 8-5
Heap 8-6
Hello world 3-3, 3-15, 3-26
color attributes 5-12
in color using physical attr. T-8
hello_wn.c 5-12, A3-2
Help file 6-3, 7-3 thru 7-9
code example 7-3
display 7-3
displaying T-10
multiple 7-4
pop-up 7-3
preparing 7-3
reading 7-3
scrolling T-10
help.txt 6-3, 7-3
Highlighting 1-4
attributes 8-11
specified number of characters 8-11
HIGH_INT 3-11, 3-13, 5-9, 5-10

I

IBM BIOS 3-22, 5-9, 9-3
_ibmega 5-7, 8-13
IN 8-7
Include files 3-4, A1-1 thru A1-15
referencing 2-5
required 2-5
Initialization 3-5
init_wfc( ) 3-4, 3-5, 5-6, 3-27, 8-13, A2-42
INSIDE 3-15, 3-18, 8-6
Internal subroutines v

K

Keyboard
checking buffer 3-23
loop functions 3-23
read single keystroke 3-22
reading 3-22
Keycode conventions 3-22
ki( ) 3-22, 3-26, A2-43, A2-44
ki_chk( ) 3-23, A2-45
ki_cum( ) A2-46, A2-47
k_vcom( ) 6-8, 6-9, T-9, A2-48, A2-49, A3-2

L

LATTQ 5-5
latt[] 5-6, 5-7
_lattsw 3-12, 5-9
LAVR 5-5
LDOS 3-24, 3-25, 3-26
LEFT_TXT 3-19
LERROR 3-11, 3-13, 3-26, 5-4
LGREEN 5-4
LHIGHLITE 3-11, 5-4
Library diskette 2-3
Library files 2-3
LIGHT 5-10
Line editor 4-8
Linking
    libraries 2-5
    programs 2-5
Listings T-1 thru T-15
LMESSAGE 3-11, 5-4, 5-6
LNORMAL 3-11, 3-12, 3-18, 5-4, 5-5
Logical attribute 3-11, 3-24,
        5-3 thru 5-8
    adding to 5-5
    adding new columns 5-6
    changing number of 5-5
    changing physical attributes of
        5-4 thru 5-5
    constructing and using window
        specific tables 5-7
    defining a name 5-5
    definitions T-4
    incompatible with physical
        attributes 3-12, 5-8
    problem with UNDERLINE 5-9
    system 5-4
    window-specific 5-4, T-7
Logical attribute array 5-6
    initializing 5-6, T-6
    setting window to 5-8
Logical attribute table 5-4
    adding a new definition 5-5
Logical color attributes 5-4
Logical video attributes 1-4
loop.c 3-23, A3-2
Low-level
    character functions 8-14
    string functions 8-14
lower_st() 8-14, A2-75, A2-76
LRED 3-13, 5-4
LSTATUS 5-5, 5-6
_l_ptr 8-13

M

Macros 3-8
    column size 3-16, 8-15
    row size 3-16, 8-15
MAGENTA 5-10
malloc() 6-7, 8-7
Managing
    memory files 6-10
    windows 1-3
Margins 7-6
Maxcols 6-4, 6-5, 6-6, 6-7
Maxlines 6-4, 6-5, 6-6
MEMLACK 6-4, 6-5, 6-6
Memory file 3-21, 7-3, 7-8, 8-3
    accessing lines 6-9, A5-8
    alternative to virtual screens 7-7
    array 6-4
    as off-screen buffer 7-8
    assigning to window 7-6
    blank lines 6-6, 6-7
    bottom of file message 6-8
    changing name 7-4
    clearing 6-11
    creating and viewing 6-3 thru 6-11
    declare FREC 6-4
    display 3-3
    error handling 6-4, 6-7
    error handling in di_file() 6-5
    error handling in sti_file() 6-6
    freeing 6-11
    initialize FREC 6-4
    management 6-10
    memory requirements 8-3
    modifying 6-9
    modifying lines 6-9
    move origin in window 6-7
    operations on 6-9
    placing lines into 6-5
    reading from disk 6-5
    replacing lines 6-9
    scroll 6-3
    scrolling contents 6-10, 7-8
    scrolling in window 6-8
    set window for viewing 6-7
    structure 6-10, 7-4,
        A5-6 thru A5-7
    top of file message 6-8
    viewing through windows 6-7
    writing directly to 6-5
Memory
    allocating 8-6
    requirements 8-3
    usage 1-4
Memory management 8-10
    error handling A6-2
    sti_file() 6-7
Menu 6-3, 7-3 thru 7-9
    calling 7-6
    code example 7-7
    creating memory file 7-5

defining display window 7-6
demonstration program 7-7
display on screen 7-6
format 7-5
placing in a memory file 7-5
pop-up 7-4, T-11 thru T-13
pop-up, creating 7-4
preparing to call 7-5
tutorial T-11
menu( ) 7-4
menu2( ) 6-6, 7-4, 7-5, 7-6, 7-7, 8-12, A2-50, A2-51, A3-2
mfile.c_q 8-5
mfile.farray 6-4, 6-11
mfile.fn 7-4
mfile.ln_q 8-5
mfile.maxcol 6-7
mfile.wfc 8-5
mfile.wfr 8-5
Microsoft Windows -- See MS Windows
Miscellaneous utilities 8-14
mode_col( ) A2-52
Modifying
memory files 6-9
window location 3-10
window size 3-10
mod_wn( ) 3-10, T-2, A2-53
MONO 5-12
Monochrome attributes 3-11, 5-3, 5-9
problem with UNDERLINE 5-9
Moving
character contents of windows 8-5 thru 8-6
character/attribute information of window 8-8
information from window 8-5
information to window 8-5
virtual cursor 3-15
windows 8-9
MS Windows 8-13, 9-3 thru 9-5
control of screen updates 9-4
direct control of screen updates 9-4
incompatibility with some compatibles 9-4
incompatible machines 2-6
running demonstrations 9-5
WFC operation 9-3
WFC video management 9-3
Multiple files
handling of 8-3
managing 8-5
using same window 8-3
Multiple help files 7-4
Multiple windows on same file 8-4
mv_cs( ) 3-15, 3-16, 3-20, A2-54
mv_csr( ) 3-16, 3-17, 3-18, 3-25, A2-55
mv_rws( ) 3-20, 3-21, A2-56
mv_scr( ) 3-26
mv_wi( ) 8-10, 8-9, A2-57

N

Naming windows 3-9
NORMAL 3-11, 3-12, 3-13, 3-24, 3-25, 3-26, 5-4, 5-9, 5-11

O

Off-screen buffer 1-3, 7-3 thru 7-9
code example 7-8, T-14
Organization of material 2-3
OUT 8-7
Overlapping windows 8-4
Overwrite windows 3-3

P

Page 3-25
PATTQ 5-6, 5-7
Pause, create 3-23
Physical attribute 3-11, 3-24, 5-3 thru 5-6, 5-8 thru 5-13
adding new column 5-6
color 5-4, 5-9
definitions T-5
enabling 5-8
incompatible with logical attribute 3-12, 5-8
monochrome 5-4, 5-9
problem with underline 5-9
problems 3-11
quantity parameter 5-6
usage of 5-8
using 2-6, 3-12
Physical cursor -- See screen cursor
PIF 9-3
pl_cs( ) 3-26, 3-27
pl_csr( ) 3-17, 3-18, A2-58
pl_mfwn( ) 6-8, 8-5, A2-59
pl_wn( ) 8-9, 8-12, A2-60
Pointers, user-reserved 8-15
Pop-up
help file 7-3
menu 1-3, 7-4, T-11 thru T-13
window 1-3, 3-3, 3-8, 3-14, 8-9
Printer 8-12
printf( ) 3-19
Printing windows 1-4, 8-12
demonstration 8-12
Program Information Files 9-5
Program Interface File 9-3
Program pause, create 3-23
prt_labl.c 1-5, 8-12, A3-2

prt_wn( ) 8-7, 8-12, A2-61
putchar( ) 3-19

R

rd_csr( ) 3-17, 3-18, 3-25, A2-62, A2-63
rd_line( ) 4-3, 4-8, 8-7
rd_mode( ) 5-12, A2-64
read.aaa 2-3, 2-6
READERR 6-5
Reading
    attributes 3-21
    help files 7-3
    keyboard 1-4, 3-22, 3-23
    files from disk 6-5
Reading windows 3-21
    attribute 3-21
    character 3-21
    character/attribute 3-21
    multiple characters 3-21
Real-time process 7-7
RED 5-5, 5-10
Removing
    color window 5-11
    window names 3-9
    windows 3-14, 3-24
repl_wi( ) 8-10, 8-11, A2-65
REVERSE 3-11, 5-9, 5-10, 8-11
RIGHT_TXT 3-19
ROW 8-5, 8-11
Row quantity macro 3-16, 8-15
row_qty( ) 3-16, 8-15

S

sav_wi( ) 5-12, 8-10, 8-11, A2-66
scanf( ) 3-19
Screen
    background color 3-24
    border color 5-11
    clear 3-24
    remove window 3-24
Screen cursor 3-7
    automatic placement 3-16
    direct placement 3-17
    hiding 3-17
    moving 3-25
    reading location 3-17
    restoring 3-17, 3-25
    saving 3-25
scrl_file( ) 6-10, 7-8, A2-67, A2-68, A2-69
Scrolling 3-18
    contents of memory file 6-10, 7-8
    direct display window 3-20
    file in a window 6-8
    help file T-10
    horizontal 6-3
    vertical 6-3
Set window member functions 3-8
Setting a window to view a file 6-7
Setting window on screen 3-6
set_wn( ) 3-4, 3-6, 3-9, 3-13, 3-14, 3-15, 3-18, 3-25, 3-26, 3-27, 5-11, A2-70, A2-71
size_wn( ) 8-6, 8-11, A2-72
skip_wh( ) 8-14, A2-75, A2-76
Snow 5-9
Source code supplied, list of A3-2
sprintf( ) 3-19
Standard compiler library 2-5
Standard keycode 3-22
Status line
    establish 3-27
    managemnent 1-3
stblank( ) 8-14, A2-75, A2-76
sti_buf( ) 7-8, 7-9, T-14, A3-2
sti_file( ) 6-5, 6-6, 6-7, 6-8, 6-9, 6-10, 7-5, 7-8, 8-5, A2-73, A2-74
Storing window images on disk 8-11
strcpyp( ) 8-14, A2-75, A2-76
String functions 8-14
String output 3-6
    at specified location 3-19
    basic 3-18
    centered 3-19
    formatted 3-19
    full 3-18
    left justified 3-19
    low level 8-14
    right justified 3-19
String utilities 8-14
stringf.c 8-14, A2-75, A2-76
strip_wh( ) 8-7, 8-14, A2-76, A2-78
sw_att( ) 3-13, 3-26, 3-27, 5-10, A2-77, A2-78, T-2
sw_bdratt( ) 3-13, 3-14, 5-11, T-2, A2-77, A2-78
sw_border( ) A2-77, A2-78
sw_cleor( ) 3-10, 3-27, T-2, A2-77, A2-78
sw_csadv( ) 3-10, 3-19, 3-20, 3-27, 8-7, T-2, A2-77, A2-78
sw_latt( ) 3-12, 5-8, 5-9, T-2, A2-77, A2-78
sw_margin( ) 3-9, 7-6, T-2, A2-77, A2-78
sw_mfile( ) 6-7, 7-6, 8-3, 8-4, T-2, A2-77, A2-78
sw_name( ) 3-9, 3-26, T-2, A2-77, A2-78
sw_plcsr( ) 3-10, 3-17, 3-27, T-2, A2-77, A2-78

sw_popup() 3-8, 3-25, 3-26, T-2, A2-77, A2-78
sw_scroll() 3-10, T-2, A2-77, A2-78
sw_wwrap() 3-10, T-2, A2-77, A2-78
System diskette 2-3
System globals 8-13
s_keyloop() 3-23
s_latt() 5-7, A2-80
s_tbfmsg() 6-8

T

Tables T-1 thru T-15
test.adr 8-12
Text
    centered 3-19
    left-justified 3-19
    right-justified 3-19
TopView 1-4, 8-13, 9-3
    compatability 1-4, 9-3 thru 9-5
    control of screen updates 9-4
    direct control of screen updates 9-4
    incompatibility with some compatibles 9-4
    incompatible machines 2-6
    running demonstrations 9-5
    WFC operation 9-3
    WFC video management 9-3
tutor.c 4-3, 8-7, A3-2
Tutorial 4-3
tut_help.c 7-3, T-10, A3-2
tv_upd 9-4, 9-5

U

uatt_tbl[][] 5-7
ulatt[] 5-7
UNDERLINE 3-11, 5-9, 5-10
    problems 5-9
UNIX 3-23
Unresolved externals 2-5
unsav_wi() 5-12, 8-10, 8-11, A2-81
unset_wn() 3-14, 3-24, 3-25, 3-26, 3-27, 5-11, 8-9, 8-10, A2-82
UP 3-21
Update video call 9-3, 9-5
upper_st() 8-14, A2-76, A2-78
User-reserved pointers 8-15
Utilities
    miscellaneous 8-14
    string 8-14
u_init() 5-6, 5-7, 8-13, A2-83

V

vextern.h 8-13, 8-14
Video buffer 9-3
Video management
    MS Windows 9-3
    TopView 9-3
Video mode
    change 5-12
    current 5-12
    determine 5-12
    graphics 5-12
Video string 3-21, 8-8, 8-11
vid_bdr() 5-11, A2-84
vid_int() A2-85
vid_mode() 3-27, 5-12, A2-86, A2-87
Viewing a file 6-7
Virtual cursor 3-7, 3-10, 3-15, 3-16, 3-21, 8-7
    automatic advance 3-10
    direct assignment 3-16
    moving 3-15
    origin 3-15
Virtual screen 7-7, 7-8
vmenu.c T-11, A3-2
vo_att() 3-21, A2-88
vo_ch() 3-21, A2-89
vs_file() 6-8, 6-9, 6-10, 7-3, 7-8, 8-3, 8-4, A2-90, A2-91
vs_file.() A3-2
v_att() 3-20, 8-11, 8-12, A2-92
v_axes() 8-12, A2-93
v_bar() 8-12, A2-94, A2-95
v_border() 3-13, 3-14, A2-96, A2-97
v_ch() 3-19, 8-7, A2-98
v_co() 3-20, A2-99
v_contig 8-13
v_coq 3-26, 3-27, 8-13
v_file() 6-7, 6-8, 6-10, 7-3, 8-4, 8-9, 8-10, 9-5, A2-100, A2-101, A3-2
v_fst() 3-18, A2-102
v_mode 5-7, 8-13
v_mov() 3-21, 8-8, A2-103, A2-104
v_mov(), code example 8-8
v_mova() 3-21, 8-5, 8-6, 8-7, A2-105, A2-106
v_mova(), applications of 8-7
v_natt() 8-7, 8-11, A2-107
v_pbytes 8-13
v_plst() 3-19, 3-26, A2-108, A2-109
v_printf() 3-19, 3-27, A2-110, A2-111
v_qch() 8-14, A2-112
v_rw() 3-20, A2-113
v_rwq 3-17, 3-26, 3-27, 8-13
v_seg 8-13
v_st() 3-10, 3-16, 3-18, 3-19, 3-26, 3-27, 3-4, 3-6, 6-9, 8-7, 8-10, 8-14, A2-114, thru A2-116
v_st_nop() A2-117

v_st_rw() 8-14, A2-118, thru A2-120
v_tv() 9-4, 9-5, A2-121
v_vrb 8-13

W

Warning to all users v
WFC operation
    MS Windows 9-3
    TopView 9-3
WFC
    applications 8-15
    capabilities 1-3
    initialization 3-5
    overview 1-3 thru 1-5
wfc_defs.h 8-6, 8-8, 8-11,
    A1-7 thru A1-10
wfc_stru.h 3-8
wfd.h 2-5, 3-4
wfd_glob.h 2-5, 3-4, 3-22
WHITE 5-4, 5-10
WIND 8-6, 8-11
WINDOW 3-7
Window
    as an edit buffer 8-7
    changing attributes 3-13
    changing defaults 3-7
    changing margins 3-9
    character and attribute contents
        8-8
    clear 3-24
    copying contents of to a file 8-8
    declare 3-5
    default 3-9
    default settings 3-6, T-2
    define initial values 3-5
    dimensions 8-6
    displaying 3-14
    duplicating structures 8-15
    error message 3-26
    fill with specified character 3-20
    formatting text for printing 8-12
    initialization required 3-8
    managing colors 5-9
    member change functions T-2
    members T-2
    memory file 8-9
    modifying location 3-10
    modifying size 3-10
    moving character contents of
        8-5 thru 8-6
    moving character/attribute
        information 8-8
    moving information from 8-5
    moving information to 8-5
    name 3-9, 3-14
    number of columns 8-15
    number of rows 8-15
    overlapping 8-4
    overwrite 3-24, 5-11, 8-9
    place 8-12
    pointing to memory file 7-6
    pop-up 3-8, 3-14, 3-24, 3-26, 8-4,
        8-9
    practical examples 3-26
    print 8-12
    reading 3-21
    removing from screen 3-14, 3-24,
        5-11
    removing name 3-9
    scroll memory file 6-8
    scrolling 3-20
    setting on screen 3-6
    structure 3-7, 3-8, 5-7, 8-15,
        A5-2 thru A5-6
    using logical attribute array 5-8
    viewing memory files 6-7
    viewing multiple files 8-3
    writing to 3-6, 3-18
    writing to full screen 3-26
Window image 8-10
    demonstration 8-10
    memory management 8-10
    moving 8-9, 8-10
    moving under user control 8-9
    replacing 8-10
    restoring 8-9
    saving 8-9, 8-10
    storing on disk 8-11
window.h 2-5, 3-4, 3-12, 3-26, 5-8,
    8-13, 9-4, A1-11 thru A1-13
WINDOWPTR 3-7
Windows for C, previous users 7-6
Windows for Data 2-5, 3-4, 3-11, 5-5,
    5-8, 9-3
wn.setsw 3-14, 8-6
wn.storp 8-10, 8-11
wn0 3-25, 3-26, 5-12
_wn_err 6-4, 6-5, 6-6, 8-7, A6-2
Word wrap 3-9, 3-18
Working dimensions 3-14, 3-15, 8-6
    changing 3-14
write() 8-11
Writing
    columns of characters 3-20
    lines to a memory file 6-5
    rows of characters 3-20
    string to an off-screen buffer
        T-14 thru T-15
    to windows 1-4, 3-6, 3-18

X

XENIX 3-23

_attcolq 5-7
_attrowq 5-7
_d_seg 8-13
_ibmega 5-7, 8-13
_lattsw 3-12, 5-9
_l_ptr 8-13
_wn_err 6-4, 6-5, 6-6, 8-7, A6-2