WINDOWS SHELL
FOR MICROSOFT WINDOWS 3.0

by

Greg McCain

December 5, 1991

Advisor: Charles Dana
Computer Science Department
School of Engineering
California Polytechnic State University
1991

TABLE OF CONTENTS

ABSTRACT

This document discusses the purpose, features, and design of a command line shell for Microsoft Windows 3.0, the Windows Shell. The Windows Shell allows users to launch both DOS and Windows programs from a command line environment, as well as perform disk maintenance operations such as copy, deleting, and moving files.

The Windows Shell is implemented in the Microsoft Windows 3.0 environment. The shell has been tested on systems running Windows in VGA and EGA video modes. The Windows Shell uses Windows 3.0 non-preemptive multitasking techniques to allow multiple instances of the Windows Shell to run concurrently, as well as other programs in the Windows environment.

The Windows Shell introduces several new features to the typical command shell environment, including interactive editing of Aliases and Environment variables, an "Action Bar" to quickly execute a command, the ability to set "permanent" options in shell commands, and customizable fonts and screen colors.

The purpose of the Windows Shell is to fill a gap in the Windows 3.0 environment. The original environment provides a highly graphical user interface for executing programs and file management. While this is great for naive users, the graphical interface can become cumbersome to experienced users. The Windows Shell is intended for experienced users, providing quick manipulation of files and directories, as well as executing Windows and DOS programs. The Windows Shell also includes some extra graphical niceties that are discussed in the following section.

HOW THIS DOCUMENT IS ORGANIZED

This document is divided into three main sections. The first presents the Windows Shell as seen by the user. It discusses the feature of Windows Shell, how to access them, and what they can do for a user. The second section presents the Windows Shell as seen by the programmer. This includes a discussion of the graphical objects seen in the Window Shell, as well as a discussion of how the command interpretation works. The last section discusses the development process of the Windows Shell. It describes the problems encountered in both design and implementation of the Windows Shell, and any other noteworthy considerations that were involved in the development process.

The document also provides the specifications of the DLL External Commands interface in Appendix A. The interface is a major consideration when developing external commands for use with the Windows Shell.

LIST OF FIGURES

I. FEATURES OF WINDOWS SHELL

I.A. THE COMMAND LINE

figure 1

The Windows Shell provides a command line interface to the windows environment. The command line is similar in appearance to a DOS shell, with the following exceptions:

- The Windows Shell prompt allows for text marking, cutting, and pasting.
- The Windows Shell offers the "Action Bar" seen on the left side of the shell (see figure 1.)
- The Windows Shell allows the user to configure the color and font used by each instance of the shell.
- The Windows Shell recognizes the '&' character to run a windows program minimized.
- The Windows Shell expands partial file names entered on the command line into the entire file name by pressing the tab key. If more than one file matches the partial name, the user will be given a choice of files.

I.B. ALIASES

Windows Shell allows the user to define aliases. Aliases allow long or complex commands to be abbreviated for quick access.

figure 2

Windows shell provides the Alias Editor (as seen above in figure 2) for quick viewing and editing of aliases.

I.C. ENVIRONMENT VARIABLES

The Windows 3.0 environment provides environment variables similar to that of DOS and UNIX. Unfortunately, the environment variables in the Windows Shell are not inherited by the programs it executes. Each new program gets a copy of the environment that was recorded when windows was started. However, the environment variables do effect the shell itself, which is useful for changing the path or prompt.

figure 3

Windows Shell provides the Environment Editor (as seen above in figure 3) for quick viewing and editing of environment variables.

I.D. ACTION BAR

The action bar is a column of eight buttons along the left side of the shell (see figure 1) It allows users to execute a predefined command at the press of a button. The following window is used to configure the action bar:

figure 4

The Button Configuration Window allows the users to define the text seen on the buttons and the command that is carries out. The user can access text marked on the Windows Shell using the '=' character. The '=' character is replaced by a string containing the marked text when the command is executed. This makes it easy for a user to mark a block of text and perform actions upon it, such as deleting or editing files. For example, to have a button which will delete any marked files, use the string:

del =

in a "button command" box, as seen in figure 4.

I.E. EXTERNAL COMMANDS

Most shells provide shell commands which are built into the shell itself. The Windows Shell provides only the most basic shell commands, including: change directory (cd), make directory (md), and remove directory (rd). The rest of the usual shell commands are implemented as "external commands".

Each external command is a Windows 3.0 Dynamic Link Library (DLL). The Windows Shell uses a Windows 3.0 DLL loading function to load and run external commands. Each external command must provide a set of functions to execute the command, show an about box, and show an options box.

The windows shell provides the "external commands window" to view and configure external commands (see figure 5).

figure 5

The options box allows the user to set "permanent" options on a command that will be invoked each time the command is run. The about box allows the user to view an about box for the command which could contain useful information on what the command does and how to use it.
The reason the "external commands" are implemented as DLL's is that it allows new commands to be written without modification to the shell. Also, the format of the DLL is such that a normal DOS shell program can easily be ported to run under the shell.

I.6. SHELL COMMANDS

The user accessible commands contained in Winshell, or Shell Commands, are as follows:

cd <dirname>
md <dirname>
rd <dirname>
<DRIVE> :
ps : Lists currently running processes.
min <task_name>: Minimizes program with caption matching task_name.
max <task_name>: Maximizes program with caption matching task_name.
kill <task_name>: Closes program with caption matching task_name.
exit
: Exits current Windows Shell.
exitwin
: Exits Windows.

II. DESIGN OF WINDOWS SHELL

The design of Windows Shell has been broken down into the many separate modules, both graphical and logical. To simplify the explanation of the design, this section is divided into two main parts. The first part will discuss the graphical object design of windows shell. That is, how the individual windows and buttons are designed, how they function, and they the communicate with one another. The second part of this section will discuss the modules used in command interpretation. This section will involve a detailed trace of the flow of control for the interpretation of a command.

II.A. GRAPHICAL OBJECTS OF THE WINDOWS SHELL

The windows shell has been divided into three main graphical objects, as seen in figure 6. Each of the objects is implemented in a separate C source file, and communicate with each other via Windows messages. Although the "Main" window as seen in figure 6 is at the top of the hierarchy, controlling the other two windows, this explanation will be more clear of it starts from the bottom.

figure 6

II.A.1 The WSTDIO Window

By far the most important window in the shell is the WStdio window. It is responsible for supplying the primitives such as reading and writing characters to a standard I/O type device. To do this in a windowing environment, is requires creating a "virtual" standard I/O display. This display is represented by a data structure, who's main element is an array of characters which represent the characters displayed on the screen. When characters are written to the WStdio window, they a first copied into this character array, and then are actually displayed on the screen.

II.A.1.a Pertinent Data

To better understand the implementation of the WStdio window, let's look at the elements of the Display data structure as defined by the WStdio window. The first feature is the array of characters representing the display (This is referred to as the "LineBuf".) Second, are a set of elements used to track the state of the display:

```
int  iTopLine;
// Index of the current top line in the LineBuf
int  iBottomLine;
// Index of the current bottom line in the

// LineBuf
int  TotalLineCount;
// the current number of line in the LineBuf
int  CurCharOffset;
// offset into LineBuff of current position
```

These first four elements are what control current state of LineBuff itself. LineBuff is an array of characters, which is logically divided into fixed length lines. The actual number of lines available in LineBuff is a constant at compile time, but is typically much larger than the number of lines that will currently fit in the window on screen. (This surpluss of lines in memory is used as a scrollback buffer, as will be discussed later.) The indices to the top and bottom lines are used because LineBuff is a circular buffer. These indices always point to the portion of the LineBuff that is currently displayed on the screen. The *TotalLineCount* variable is the absolute number of lines in LineBuff. *CurCharOffset* is absolute position in LineBuff at which the next character to be displayed will be written.

The next set of elements is used when actually outputting lines to the screen. Their uses and intentions are explained in the comments.

int nLinesOnScreen;

// in it's current size
int nCurLineOnScreen;
// the current line the cursor is on in window

// starting from 1 (NOT 0!)
int yChar,
// height in pixels of a line
int xCursorPos;
// distance (in pixels) from the left hand

// side of the screen that the cursor is at

When the user activates the scrollback feature by using the vertical scrollbar or the PAGE UP/PAGE DOWN keys, these variables are initialized and used to track the positioning of the scrollback:

BOOL
// True is window is in a scrolling back state
int
// index of the top line in the LineBuff

// during a scrollback

Another noteworthy element in the Display data structure is used to expedite the actual outputting of lines on the screen.

int nUnpurgedLines,
// # of line waiting to be written to

// the screen

The WStdio window does not display a line of text immediately when it is received. If possible, the window will wait until a predefined constant number of lines come in before it actually displays the lines. This can dramatically increase the rate at which lines are displayed on the screen. The **nUnpurgedLines** variable counts the number of lines waiting to be displayed.

Finally, there are elements used to track how the user has highlighted any text in the WStdio window:

WORD wSelectState;
RECT rectInversion;

The *wSelectState* variable can be in three states: one indicating there is no highlighted rectangle, when all bits are turned off. The second, SS_SELECTING, indicates the user is currently marking a rectangle. The third, SS_RECTSELECTED, indicates that a rectangle is currently highlighted. When in third state, the *rectInversion* variable will hold the coordinates of the highlighted rectangle.

II.A.1.b Painting The Display

When a message is sent to the WStdio window to write a line to the display, it's first job is to copy that

line to the line buffer. It then *invalidates* the region of the window that will be effected by the new text. Having done this, the function is essentially done. The job of actually writing the text to the screen comes later, in response to a Windows WM_PAINT message. To understand why it is implemented this way, one must understand the concept of the MS Windows WM_PAINT message. Windows dictates that all screen I/O should be done in response to a WM_PAINT message. The message informs a window that it needs to repaint a portion of it's client area, and supplies the window with the coordinates of a rectangle it needs to repaint. Thus upon receiving this message, the WStdio window calculates what lines need to be painted, and paints them. Back when a client module requested the WStdio window to display a line, the WStdio window only had to copy the line into it's internal Linebuff, and invalidate the portion of the window that will be effected by the new line. By invalidating a portion of the window, Windows will generate a WM_PAINT message, and the window will be repainted, reflecting the new line to be displayed.

II.A.1.c Obtaining Standard I/O Input

Perhaps the most interesting feature in the implementation of the WStdio window is how it obtains input from the user. To provide a function like getstr(), which does not return until the user presses the ENTER key, the Windows message loop had to be placed inside the getstr() command. This allows other processes to run while the WStdio window is waiting for input.

When a client module sends a DM_GETS message to get a string from the WStdio window, the WStdio window calls the DisplayGetStr() function. This function first positions the caret at the appropriate position, and then falls into a message loop. Inside this message loop, the function monitors the incoming messages looking for the ENTER key to be pressed, in which case it will fall out of the loop and return the text that was entered. The function also monitors the incoming messages for keys like the arrow keys, in which case it will invoke the command history, and for WM_CLOSE message. If a WM_CLOSE message comes in, the DisplayGetStr function exits the message loop and returns a value indicating that the function failed.

With this method of implementing the message loop in the input function, the client modules need not use a message loop. The main module, WINSHELL.C, in fact does not use a message loop. Instead, it falls into a loop that might be expected out of a UNIX type command, in which it displays a command prompt, gets an input string, interprets it, and executes the appropriate action. It does not use a message loop at all, like most other Windows WinMain functions have to do. But this is an aside and will be cover more in the section on the WINSHELL.C module.

In order to obtain input from the user, the WStdio window uses a Windows edit control. When the user sees a prompt at which he or she can type, that prompt is actually inside a separate edit control, and not in the WStdio window itself. The edit control is always positioned at the end of the last character entered, much as a caret would be. This way, however, leaves much of the work of obtaining key-presses and displaying characters to the edit control. It also helps provide the standard controls a user might expect from an input prompt. Such things as marking text and replacing text will remain consistent with other edit controls, and in future versions of Windows.

II.A.1.d Marking Text With The Mouse

The job of marking text on the display is quite simple. The WStdio window responds to a WM_LBUTTONDOWN message (indicating the left mouse button is being pressed,) by obtaining a mouse capture. This forces Windows to send all subsequent mouse message to the window obtaining the capture. While in this state, the WStdio window then responds to all WM_MOUSEMOVE messages by inverting a rectangle between the position where the mouse was originally pressed and the current position. When the left mouse button is released, the capture is also released, and the rectangle is left highlighted.

The portion of highlighted text can now be accessed by both the WStdio window itself, and by client windows via a DM_GETMARKEDTEXT message. The function GetMarkedText(), in the

wstdio.c module, is responsible for determining what characters are actually marked and copying them to a buffer. This is no simple task when proportionally spaced fonts are in use. The function must navigate LineBuff and determine the actual length in pixels of each character in the buffer. It then compares this to coordinates that are marked in the screen, and can determine what characters are actually marked.

II.A.1.e Command History

The WStdio is also responsible for providing a command history. Whenever the user enters a command at the command prompt, the WStdio window records the command entered with the CommandHistory() function. This function manages a simple queue of a constant size. When a user enters a command, it is added at the end of the queue, and the first item in the queue is discarded if there are more than the constant limit of items in the queue. When the user presses the up and down arrow keys, the WStdio window responds by displaying items from this queue on the command line.

II.A.1.f Wstdio Message Directory

The following is a list of messages that the WStdio window provides for client modules:

- Writes a string to the display.

- Gets a string from the display.

- Clears the Wstdio window.

- Sets the font the Wstdio window will use.

DM_GETNUMCOLUMNS - Returns the approximate number of columns on the display. A column is space enough for about 12 of the widest characters in the current font, and a trailing tab. This feature essentially indicates the number of file names that can be displayed on one line.

DM_GETMARKEDTEXT - Returns a global handle to memory block containing marked text. This memory must be freed by the user

DM_SETMORE - Turns the more feature on and off. When this feature is turned on, the WStdio will automatically display a ----more---- at the bottom of the screen after the last number of lines displayed has filled up the screen. The more feature is automatically turned off when after DM_GETS message is sent.

II.A.2 The ACTNBAR Window

The actnbar window provides the column of user configurable push buttons along the left hand side of the Windows Shell. The ACTNBAR window is actually a rectangular window surrounding the set of push buttons. Associated with each push button is a caption and a command. The caption is the text that is displayed in the button on the screen. The command is the command string which will be invoked when the button is pressed.

When the ACTNBAR window is created, it creates it's push button children and initializes them to defaults saved in the WINSHELL.INI configuration file. It's job thereafter is to report to it's parent window whenever one of it's buttons has been pushed, passing the parent window the command string to be executed.

This is accomplished using the standard Windows WM_COMMAND message. This message is sent to the ACTNBAR window whenever one of it's children is pressed. The ACTNBAR window then sends this same message to the parent, and indicates what function to perform by setting it's own caption text the command text assocaiated with the button. Thus when the parent receives the WM_COMMAND from the ACTNBAR window, it reads caption text of the ACTNBAR window, and executes the command contained therein.

Note that this logical command path is different from that of the other command interpretation path. A command executed in response to the ACTNBAR being pressed is interpreted and executed in response to the WM_COMMAND message, and is not obtained via the command interpretation loop in the WinMain function. This is discussed more in section II.B.2.

Also contained inside the ACTNBAR.C module is action button configuration dialog box. This allows users to configure both the caption and command of each action button.

II.A.3 The WINSHELL Window

This window is the main window of the application, and controls the other windows as seen in figure 6. The graphical job of this window is quite simple. It's job is to manage that size and position of the other two windows, namely the WSTDIO window and the ACTNBAR window. These windows are both children of the WINSHELL window, and reside inside the client are of the WINSHELL window.

Apart from this, the WINSHELL also provides the menu bar as seen at the top of the window. It must respond to menu messages and execute the appropriate functions. These functions include changing the font in the WSTDIO window, changing the colors of the Windows Shell, and popping up the various configuration dialog boxes. These duties mostly involve sending a single message to the appropriate window to perform the task. In this way, the WINSHELL window serves more as a message router for it's children than anything else.

However, if this seems to simple, that because it is. This is only a discussion of the graphically oriented tasks the WINSHELL window must perform. It's main task, that of command interpretation, is discussed in the next section.

II.B LOGICAL MODULES FOR COMMAND INTERPRETATION

This section discusses the modules involved in command interpretation. Now that the relationship of the 3 main windows has been defined, the matter of understanding command interpretation will be much easier. First let's look at the design of the command interpreter, as seen in figure 7.

figure 7

The flow of control in figure 7 moves left to right. The diamond shaped modules indicate modules which are returning user input. The circular modules perform some logical operation on the input data, and the square boxes will be the end result of the command.

There exist two paths in which a command can be executed by the shell. The first is by the user entering a command at the command prompt. The second, is the user pressing an action button. The former is accomplished by looping for user input, interpreting it, and executing it; the latter is done

only in response to a user pressing an action button.

II.B.1 The Command Prompt Path

Command interpretation begins in the WINSHELL.C module. This module contains the WinMain function, which is the entry point of a Windows application. After performing it's initializations and creating it's child windows, the WINSHELL.C module falls into the command interpretation loop as follows:

```
do

{
  DisplayPrompt (hwndDisplay);

  // exit if display says to
  if (dgets (hwndDisplay, szCmdLine, MAX_COMMAND_LENGTH)== -1)
    break;

  ExpandAliasString (szCmdLine, MAX_COMMAND_LENGTH);

  iInterp = InterpretCommand (hwndWinShell, hwndDisplay, szCmdLine);
} while (bContinue && iInterp != -1);
```

The first function in the loop displays the command prompt on the WStdio window. The next job is to obtain a line of text from the user, which is accomplished by the dgets() macro. Note that if dgets() returns -1, it means the user has closed the window, and the loop must be exited. After the user input has been obtained, the input is passed to the ExpandAliasString() function in the ALIAS.C module, which will expand any aliases found in the string. Finally, the string is passed to the InterpretCommand function in the INTERP.C module. It is in this module that the string is parsed and executed.

Note that commands retrieved from the command line are not given a chance to expand the marked text symbol by using the WSTDIO.C module. The marked text symbol is provided so that the symbol in the command string is replaced by the text marked in the WStdio window. Although this would be a desirable alternative, a minor design flaw stopped me from implementing it.

II.B.2 The Action Button Command Path

The ACTNBAR.C module can instigate a command by sending a message to the WINSHELL window. The WINSHELL responds by executing the command using the flow of control as seen in figure 7. The following code is executed in response to such a message:

```
GetWindowText (LOWORD (lParam), szCmdLine, MAX_COMMAND_LENGTH);

ExpandAliasString (szCmdLine, MAX_COMMAND_LENGTH);
ExpandMarkedText  (szCmdLine, MAX_COMMAND_LENGTH);
InterpretCommand  (hwndWinShell, NULL, szCmdLine);
```

As you can see, the code is very similar to that used in the command interpretation loop in the previous section. The only difference is that commands executed in response to the ACTNBAR module get to use the ExapandMarkedText() function, which expand the marked text symbol into the text currently marked on the WStdio window. Also, the command executed in response to the

ACTNBAR module are passed a WStdio window handle of NULL. This means that these function will not be able to output any data to the WStdio window.

II.B.3 The ALIAS.C Module

The ALIAS.C module provides aliases expansion for command strings. It is implemented as a Windows DLL, which provides the following advantages. For all instances of the Windows Shell running, they all share a common ALIAS module. This means that if an alias changed in one shell, it is changed for all shells. This methods also expedites the loading process of Windows Shell, because the default aliases only have to be read from disk once, for the first module.

The implementation of the ALIAS module is quite simple. It maintains a dynamically resizable array of elements. Each element contains an alias name and an alias value. The module provides a dialog box for adding and deleting aliases, and the ExpandAliasString() function, which takes a string and expands any aliases within it.

II.B.4 The INTERP.C Module

The third and perhaps most important module used in command interpretation is the INTERP.C module. This module is responsible for parsing a command line, determining what type of command it is, and executing the command accordingly. There are three types of commands that the module must discriminate between.

The first type of command the INTERP module looks for are shell commands. These are commands who's code is kept inside the Window Shell. This is accomplished by a large switch statement that checks if the requested command is amongst those known to be external commands. If INTERP determines that command is a shell command, it merely has to call the function associated with the shell command. All shell commands are contained in the COMMANDS.C module. These commands are so trivial they will not be discussed.

The second type of command the INTERP module looks for are external commands. These commands are implemented as Windows DLLs. The INTERP function searches the default directory for files who's names match the specified command name and ending with the .WS extension. If a corresponding external command file is found, the DLL is loaded and control is passed to it. The specifications for an external command are outline in appendix A.

The third type of command the INTERP module looks for are executable files. Windows provides the WinExec() function which performs this function. However, the WinExec function only searches for executable files with and extension of .EXE. DOS on the other hand, allows a different type of executable file to end with the .COM extension. Also, Windows provides a DOS shell configuration file which can also be executed, who's extension is .PIF. In order all these types of files to be executed, the INTERP module first gives the WinExec function a crack at executing it. If this fails, INTERP then searches the PATH for a file who's name matches the command name and ends in either a .COM or a .PIF. If either of these are found, an explicit file name and path is created and passed to the WinExec function, which will then execute the appropriate files.

If none of the above types of commands are found to match the specified command, an "unknown command" message is displayed.

III. DEVELOPMENT PROCESS

This section discusses some of the problems and considerations that I ran into while designing and implementing the Windows Shell. On a general note, it seems that many of the major design issues I originally set out to implement worked quite well. From the onset, I intended to create a Windows application that did not have a message loop in WinMain function. I wanted the main module worry

more about the matters of command interpretation than windowing. The message loop was to be hidden in the screen I/O functions, which is how it is now implemented. Also, the idea of having external commands implemented as DLLs worked out great. I really had no idea if either of these ideas were feasible when I started.

User Feedback

So far I have received feedback from on;y one person. I received a call from a software tester in Pennsylvania, who had downloaded the Windows Shell from a local BBS. He said that he really liked the product, and commented that I should consider going shareware with the it. Furthermore he had the following recommendations for the Windows Shell:

-User loadable alias files, supporting multiple loads of different files

- Allow more variables to be user configurable, such as the scrollback buffer size.

Problems Encountered

The remainder of this section will be organized in a problem-solution format. I will first present a problem or consideration, and then discuss how it was solved or overlooked.

P - As mentioned in section II.B.1, commands entered at the command line aren't given a chance to expand the marked text symbol into text that is actually marked on the WStdio window. I had originally intended to provide this feature, but the implementation in the WStdio window prevented it. The problem lies in that after a command is entered by the user, the ENTER key immediately generates a newline character. When a newline is output to the WStdio window, the marked text is automatically un-marked. Thus by the time the command is being interpreted, any marked text is no longer valid. The Action Bar does not have this problem because it does not generate a newline character when pressed.

S - The solution for this problem was to disallow marked text expansion on the command line. While this is more or less avoiding the problem, I have not found an acceptable way to rectify it.

P - The Microsoft C functions for manipulating ENVIRONMENT variables do not work in a Windows program.

S - The solution was to get a pointer to the environment area, and do all ENVIRONMENT manipulation functions by hand. This was a messy job, but the only way I found to get it to work.

P - In Windows, the ENVIRONMENT of the parent is not inherited by the programs is spawns. This has made the ENVIRONMENT editor almost useless.

S - There is nothing that can be done for this, except to not use Windows.

P - The speed of I/O to the display was too slow. For example, as lines were output from an LS command, the rate at which lines were displayed was very slow. This is of course do to the graphical nature of displaying and scrolling text.

S - The solution was to buffer the lines as they came in, and to output them in bursts. This made the code considerably more difficult to understand, but the end result was highly desirable. I found that by just buffering every other line, screen I/O was greatly increased.

P - Use of "strtok" C library function is dangerous. Because I am lexically analyzing a command line to determine what
to execute, what flags to set, etc, I intended to use strtok. However, the C
library version is not compatible with Windows, because it allocates memory
in an incompatible way. So I decided I would write my own.
However, I realized that the way strtok is used will not work in a multi tasking

environment. Strtok "remembers" the last parameter you gave it, which allows you to call it successively to get the next tokens. But with different programs using strtok simultaneously, it will get garbled.
S - I considered two possible solutions for this problem. The first was to redesign my strtok function to always require the string to be parsed as a parameter. The calling routine would have to supply two buffers, one holding the source string, the second holding a buffer in which strtok could do it's work. This however, would be quite cumbersome for the client using strtok. The second solution is to carefully organize the use of strtok so that no two modules would ever conflict. I chose to go with the latter, since it seems to work and required the least amount of change.

APPENDIX A. THE EXTERNAL COMMANDS DLL INTERFACE

OVERVIEW

associated files in the GENERIC directory provide a template for creating a new External Command.

your DLL to be completely reentrant, no variable can be stored in the data segment. That is, don't declare variables outside of a function, and no static variables inside a function. The reason for this is that each invocation of a DLL function uses that same data segment. If a DLL function was called reentrantly, static variables would be overwritten. Thus, if a DLL requires more data than will fit on the stack, use dynamic allocation.

REQUIREMENTS

Each DLL must provide the following 3 functions for use by Windows Shell:

int FAR PASCAL ModuleProc (HWND hwndDisplay, int argc, LPSTR argv[]);
@ ORDINAL 3
hwndDisplay - Window handle of STDIO Display to use for I/O.
argc - Number of command line arguments.
argv - Array of pointers to command line arguments. The first pointer always points to the name of the DLL.
This function is called to let the DLL do the function which it is providing. For example, if this were a DLL providing a file deletion function, the DLL would perform the deletion at this time.

int FAR PASCAL ShowOptions (HWND hwndParent);
@ ORDINAL 4
hwndDisplay - Window handle of STDIO Display to use for I/O or as parent.
This function is called to tell the DLL to show it's options box. The DLL should display a window which allows the user to set options in the DLL.

int FAR PASCAL ShowAbout (HWND hwndParent);
@ ORDINAL 5
hwndDisplay - Window handle of STDIO Display to use for I/O or as parent.
This function is called to tell the DLL to show it's about box. The DLL should display an about window at this time.

NOTE - It is essential that the DLL export these functions at the specified ordinal value in it's .DEF file. Otherwise, The Windows Shell will not properly access the DLL.

USING THE DISPLAY

The header file 'wstdio.h' has been provided for outputing lines and other function to the display. The most common of these is dputs(), which you can use to output a line to the display. See the header file for the description of the rest of the functions.

UTILITY FUNCTIONS

The 'wslib.dll' provides several useful functions for parsing command lines, and yielding to other applications. It is extremely important that you use the YieldToOthers() function your code sits in a tight loop for an extended length of time. See the header file 'wslib.h' for a list of useful functions.