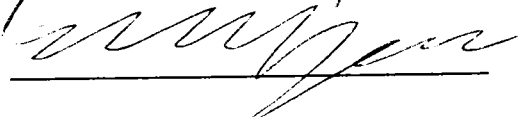


To the Graduate Council:

I am submitting herewith a thesis written by Kirk Hedden entitled "Transparent Cross-Platform Source Code Compatibility through the Use of Translation Libraries." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.


Michael Vose, Major Professor

We have read this thesis and
recommend its acceptance:

Accepted for the Council:


Associate Vice Chancellor and
Dean of The Graduate School

Transparent Cross-Platform Source Code Compatibility

Through the Use of Translation Libraries

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Kirk Hedden

August 1997

Acknowledgements

This research was supported under TVA contract TV-94549 with funds provided by the Office of National Drug Control Policy's Counterdrug Technology Assessment Center, Executive Office of the President.

The software was developed to support the mission of the West Florida Counterdrug Investigative Network (WFCIN).

Abstract

This study looks at the issue of cross-platform source code compatibility and how it can be achieved using a “translation library”, a set of source code files which implement calls for one computer platform on another through wrapper functions. This thesis was developed out of the need to port the Link Discovery Tool, written for the West Florida Counterdrug Investigative Network, from Unix X Windows to Microsoft Windows. Many native X Windows and Unix calls were simulated with wrapper functions which provided similar functionality through native Windows calls. The results were extremely promising, allowing the original Link Discovery Tool source code to compile and link through this translation library successfully on Windows NT and Windows 95 systems while still providing very good performance. Through this method only one set of source code files need be maintained and the time and expense of a traditional port is avoided.

Preface

Throughout this paper there are many examples of C source code accompanied by ample discussion. Several labeling conventions were used in order to make the code more readable but are not explained in the discussion of each code example. The conventions are as follows: global variables will begin with a lowercase 'g' and will have all subsequent words capitalized (e.g. gVariable), predefined constants will begin with a lowercase 'k' and will have all subsequent words capitalized (e.g. kConstant), and local variables will have the first word begin with a lowercase letter and will have all subsequent words capitalized (e.g. localVariable).

Additionally, the source code examples are formatted in a mono-spaced font rather than the proportionally-spaced font used for the text in order to set it apart. Any exceptions to these conventions will be noted.

Table of Contents

I. Cross-Platform Compatibility	1
The need for cross-platform code compatibility.	1
How can cross-platform code compatibility be obtained?	2
II. Unix X Windows to Microsoft Windows compatibility.	5
Simulating Unix functions under Microsoft Windows	6
printf(), gets(), getc(), and ungetc()	7
opendir(), readdir(), and closedir()	12
select() and system()	15
Simulating X Windows functions under Microsoft Windows	19
XOpenDisplay(), RootWindow(), DefaultScreen(), and XCreateSimpleWindow()	20
XCreateGC(), XSetForeground(), XSetLineAttributes(), XSetBackground(), and XSetFunction()	22
XDrawPoint(), XDrawLine(), and XDrawString()	27
XFlush(), XClearWindow(), and XAllocNamedColor()	28
Handling Events: XCheckMaskEvent()	30
III. Conclusion	37
Index	41
References	51
Appendix	53
Vita	75

Chapter I

Cross-Platform Compatibility

The need for cross-platform code compatibility.

Source code compatibility between different command-line systems is typically trivial, often needing only a simple recompile if the code makes calls supported over both systems. Source code which does not deviate from a specified standard, such as ANSI C, usually falls into this category. Such programs are generally rather simple in nature, providing simple alphanumeric output and accepting typed input commands from the user.

However, the standard for programs has changed drastically over the past decade with the proliferation of widely accepted graphical user interfaces (GUIs) such as Microsoft Windows. The success of GUIs can be attributed in large part to the success of the personal computer; as more and more people outside of the computing field began using computers in their work places and homes, a more intuitive method of interacting with the computer was welcomed. Indeed, now a GUI is considered standard fare for any computer platform, even Unix power-users use X Windows on their machines. However, while providing a GUI made the process of using a computer much easier for some, it dramatically changed the way programmers had to write programs.

Traditional command-line driven programs were written with a linear flow of control, making it relatively easy for the programmer to know what state a program would be in at any given point in the source code. This approach also allowed the program to be in "control", only allowing the user to input commands when they were needed.

The advent of GUIs requires a new approach to writing a program; it requires a

program with an event-driven flow of control. This approach puts the user in control of the program, allowing him to make choices and issue commands at will. Additionally, the interface is usually composed of graphic objects, such as buttons, menus, lists, and so on, which require a direct manipulation device such as a mouse. This point-and-click interface also requires extra effort on that part of the programmer; now, the programmer has much more user input to process in addition to key presses. The programmer must know when a mouse button is clicked, where the mouse cursor is located on the screen when a button was clicked, and what graphical interface object, if any, that cursor is positioned over when the click occurs. This makes the programmer's job of writing a program much more complicated.

In order to make the programmer's task easier, the vendors of these GUIs created application programmer interfaces (APIs) which provide high-level access to the functionality of the GUI. Unfortunately, there was little thought of cross-platform compatibility when these APIs were being created. Therefore, a program written to work with one system's GUI would have to be completely rewritten if it was to be compiled for a different GUI. When a program already exists for one system and the need for it to operate on a different system arises, this porting problem can be exacerbated when the time, money, or knowledge resources are not available to rewrite large portions of the program.

How can cross-platform code compatibility be obtained?

When looking at how to obtain source code compatibility across multiple platforms, there are two distinct choices. A common API can be used providing that one is available across all the target platforms, or a translation layer can be used to map code already written for one platform to another.

The first option, a common API, is attractive but has several disadvantages. There would be a learning curve necessary for anyone wishing to learn this universal API, all OS vendors would have to agree on the implementation of this universal API and actively support it or else this API would need to be implemented in a "middleware" product that would be available on all platforms, this generic API would need to support the feature sets of every platform while at the same time making all of these features available on every other platform, and, finally, when a new operating system is produced the user of this generic API would have to depend upon those that implement this universal API to make it available on this new platform.

The second option, having the specific platform code work on other platforms, shares some of the same disadvantages of the first option but is also more attractive in some ways. The greatest advantage to this method is that the programmer would not have to learn an entirely new and different API or language in order for the program to operate on different platforms. Code already written for one platform can be linked through a "translation library" that allows it to work on another platform. The greatest significance of this advantage is that legacy code already keyed to a specific platform can be made to work under an entirely different platform without any additional coding. Another advantage is that the operating system vendors need not be involved for these "translation libraries" to be written. The disadvantages are that these libraries need to be written specifically for each pair of platforms. Maintenance for each translation library is also a significant issue to consider since vendor's APIs frequently change.

This paper looks at building a translation library that will take source code written for an X Windows system and allow it to compile and execute successfully on a Microsoft Windows system. The major concerns when creating a transparent solution such as this are compatibility and efficiency. How well does the translation library allow X Windows code to execute on a different system, and how efficiently does the code execute?

This project was conceived from the Link Discovery Tool developed for the West Florida Counterdrug Investigative Network. The program was originally written for an X Windows system but the need to have it ported became evident due to the fact that the law enforcement offices were more able to afford Windows personal computers than the more expensive Sun workstations needed to host an X Windows environment.

Chapter II

Unix X Windows to Microsoft Windows compatibility.

When creating a translation library for taking code from one system and making it compile and operate on a different system, we must first look at the ways these two systems are implemented and try to create a mapping between functionality and implementation. Fortunately, many GUI systems take a common approach to most design issues; X Windows and Microsoft Windows are two such systems that utilize this similar design structure: both are windowing systems, with one window being the "focus" or "active" window at any given time, the graphics API routines can be used to draw primitive objects (i.e. lines, circles, etc.) in a window, and both use an event-loop method for handling user and system commands.

These similarities make it fairly easy to simulate functionality of the X Windows system on a Microsoft Windows system. The most daunting problems arise in making the Windows system handle functionality related to the systems' greatest differences. Although both systems employ the use of a command-line as well as a GUI, the command-line interface is much more integrated with the GUI on a X Windows system than on a Microsoft Windows system. Unlike the Microsoft Windows OS, where an icon is double-clicked to execute a program, X Windows executes programs from a command-line window. Therefore, a X Windows program will be associated with a particular command-line window during execution whereas a Microsoft Windows program will not be. While this may not appear to be a significant distinction, it has several ramifications, most notably regarding programs that rely on the ANSI C concepts of standard input (stdin) and standard output (stdout).

When an X Windows program makes calls to ANSI C functions that use `stdin` and `stdout`, these functions look to the associated command-line window for the source of the input or the target for the output. This is simple enough, but only works because of X Windows reliance on the command-line interface. The Microsoft Windows GUI has no concept of `stdin` and `stdout`, and, although ANSI C functions which rely upon them will compile without argument, they are treated as being undefined. Therefore, where a call to `printf()` in an X Windows executable will provide feedback to the user, the same call in a Microsoft Windows program will produce no output because the program has no defined way of knowing where it should send output to.

The one exception to this statement is if the program is compiled and executed as a console application, an application that has no reliance on the graphical windows and runs entirely in its own text window. In this case, the application treats its console as the target for `stdin` and `stdout`. But for a program to operate correctly as a console application, it must rely almost completely on ANSI C, bypassing the biggest advantage of Microsoft Windows: its robust GUI.

So, before the process of supporting the X Windows API can be attempted, the more basic task of supporting Unix-specific calls under Microsoft Windows must be addressed. The Unix calls discussed and implemented in this paper are:

```
gets()  
getc()  
select()  
ungetc()  
opendir()  
readdir()  
closedir()  
system()
```

Simulating Unix functions under Microsoft Windows

Fortunately, conventional Windows programs are able to support a text-based

console window which can easily output or receive text to and from the user. By using this feature, it is easier to simulate the functionality of an X Windows program, even though there are several issues which must be dealt with. Namely, even though a Windows program can support a text-based console window, the program has no defined stdin or stdout. Calls to `printf()` and other such functions will not read or write to that console window. The Windows programmer is expected to use the platform-specific calls `WriteConsole()` and `ReadConsoleInput()` to interact with the console window.

`printf()`, `gets()`, `getc()`, and `ungetc()`

The problem that must be faced is how to get the ANSI C functions which rely on stdin and stdout to work properly with this Windows console. While it would be possible to simply replace all the calls to `printf()`, `gets()`, and so on in the original source code with the console functions, this would be highly undesirable since it would be best if the X Windows source code could be compiled and run on a Windows system with little or no modification. The most transparent solution is to redefine these ANSI C calls so that they will work correctly with the console window under Windows. In order for this method to work properly, functions which simulate the functionality of the ANSI C calls under Windows must be written and then `#defined` to invisibly replace the ANSI C calls with the redefinitions; the need for these `#defines` is discussed in more detail at the end of this section. For the sake of consistency, all of the redefined functions will be named by their ANSI C names prefixed with 'win'. For example, a function which implements the functionality of `printf()` is named `winprintf()`, and the following line of code is placed in a header file to be included in the original Unix source code:

```
#define printf winprintf
```

The redefinition of `printf()` is:

```
void winprintf( char *string, ... )
```

```

{
    va_list ap;
    int    x, num=0;
    void *a[10];
    char ch = '%';
    char *pos, *oldPos, *ptr, buf[20], newStr[200]="";

    va_start( ap, string );

    pos = strchr( string, ch );
    oldPos = string;

    while( pos )
    {
        strncat( newStr, oldPos, pos-oldPos );

        switch( *(pos+1) )
        {
            case 'i':
            case 'd':
                x = va_arg(ap, int );
                _itoa( x, buf, 10 );
                strcat( newStr, buf );
                break;
            case 's':
                ptr = va_arg(ap, char*);
                strcat(newStr, ptr);
                break;
        }

        oldPos = pos+2;
        pos = strchr( pos+1, ch );
    }

    strcat( newStr, oldPos );

    WriteConsole( gConsoleOut, newStr, strlen(newStr), &num, vd );

    va_end( ap );
}

```

A variable argument function must be used to be compatible with the use of printf(). The function steps through the string, looking for the '%s', '%d', and '%i' escape sequences, replacing them with the proper variable values provided as parameters to the function. For full compatibility, the function would need to support all escape sequences (%e, etc.); only the most common usages are supported at the moment. Once all substitution has taken place, the final string is written to the console window with the WriteConsole() function.

Notice that the first parameter, `gConsoleOut`, to the `WriteConsole()` function is not declared as a local variable; it is a handle to the console window which is declared as a global variable and is instantiated when the console window is created at program initialization.

Support for reading user input from the console window is also needed, so we redefine `gets()` as:

```
char *wingets( char *out )
{
    long n, length=0; short j, k, l;
    INPUT_RECORD recBuf[10];
    char newline = '\n', space = ' ', buffer[100];
    BOOL done = 0;

    if( gBufLength > 0 )
    {
        for( k=0; k<gBufLength; k++ )
        {
            out[k] = gConsoleBuf[k];
            gConsoleBuf[k] = '\0';
        }
        gBufLength = 0;
    }
    else
    while( !done )
    {
        ReadConsoleInput( gConsoleIn, recBuf, 10, &n );
        for( j=0; j<n; j++ )
        {
            if( recBuf[j].EventType == KEY_EVENT &&
                recBuf[j].Event.KeyEvent.bKeyDown )
            {
                for( l=1; l<=recBuf[j].Event.KeyEvent.wRepeatCount; l++ )
                {
                    if( buffer[length] = recBuf[j].Event.KeyEvent.uChar.AsciiChar )
                    if( buffer[length] == kHardReturn )
                    {
                        WriteConsole( gConsoleOut, &newline, 1, &num, vd );
                        for( k=0; k<length; k++ )
                            out[k] = buffer[k];
                        out[length] = '\0';
                        done = 1;
                    }
                    else if( buffer[length] == kBackspace )
                    {
                        if( length > 0 )
                        {
                            WriteConsole( gConsoleOut, &buffer[length], 1, &num, vd );
                            WriteConsole( gConsoleOut, &space, 1, &num, vd );
                        }
                    }
                }
            }
        }
    }
}
```

```

        WriteConsole( gConsoleOut, &buffer[length], 1, &num, vd );
        length--;
    }
}
else
{
    WriteConsole( gConsoleOut, &buffer[length], 1, &num, vd );
    length++;
}
}
}
}
}

return( out );
}

```

This redefinition of `gets()` retains the ANSI C function's functionality by blocking the system until a string is entered into the console window. This function starts by checking the `gConsoleBuf` global array to see if any input is waiting in the console window's buffer. If so, then this is the string that is returned and no blocking occurs. If no input is readily available, the function enters a loop where it checks input to the console, storing it in a local array until a carriage return is detected, at which point the input is copied into the array passed in as a parameter and returned. If a backspace character is entered into the console, several steps must be taken to ensure that the previous character in the console is erased; first, the backspace character must be echoed to the console to move the insertion point back one character, then a space character is entered to blank whatever character is there, then the backspace is entered once more to place the insertion point at the correct location.

The global variables `gConsoleBuf` and `gBufLength` referenced here are also used in many functions throughout the compatibility code. They are used to keep track of the state of the console window; `gConsoleBuf` holds the characters entered into the console but not yet read by the program, and `gBufLength` is an integer denoting the number of characters currently held by `gConsoleBuf`. These global variables are necessary due to the fact that the program using these compatibility functions may use them to implement non-blocking

reads of the console through the use of the Unix function `select()`, discussed later. When a user types into the console window, the characters must be registered by this source code and echoed back to the console with calls to `WriteConsole()` in order for the characters to be displayed.

The function `getc()` is also supported for handling user input:

```
char wingetc( struct _iobuf *z )
{
    char x;
    short i;

    x = gConsoleBuf[0];

    for( i=0; i<gBufLength; i++ )
        gConsoleBuf[i] = gConsoleBuf[i+1];

    gConsoleBuf[i-1] = '\0';

    gBufLength--;

    return( x );
}
```

This implementation of `getc()` looks at the console window's buffer, returns the first character in it, and removes that character from the buffer. Note that if there is no input, a null value (0x00) is returned.

Accordingly, `ungetc()` is also supported:

```
void winungetc( char x, struct _iobuf *z )
{
    short i;
    char t;

    gConsoleBuf[gBufLength+1] = '\0';

    for( i=gBufLength-1; i>=0; i-- )
    {
        gConsoleBuf[i] = gConsoleBuf[i+1];
    }

    gBufLength++;
}
```

```

    gConsoleBuf[0] = x;
}

```

This function takes the character passed as an argument and places it back onto the front of the console window's buffer.

Now that we have redefined those ANSI C functions with versions that will work properly under Windows, the next problem is getting them linked into the project. The problem arises because all of the functions we are trying to replace (printf(), gets(), etc.) are all defined in the file stdlib.h and are called in the original X Windows source code. Since avoiding any direct modification to the original source code is desirable, the solution is to place some redefinitions in a header file which will be included in that original source:

```

#undef getc

#define printf    winprintf
#define gets      wingets
#define getc      wingetc
#define ungetc    winungetc

```

Note that it is necessary to undefine getc with #undef since it is defined as a macro rather than a true function. The result of the definitions is to transparently change all of the ANSI C calls in the original source into the compatible custom functions defined above.

opendir(), readdir(), and closedir()

There are several other basic UNIX and ANSI C functions that need to be implemented in the compatibility code in order for them to work as intended. Any such basic functions that may be used in general X Windows programs need to be supported.

The functions opendir(), readdir(), and closedir() are used to examine the contents of a directory and are defined in the dirent.h header file on UNIX systems. They are not available on Microsoft Windows systems. In addition to defining these functions under Windows, the ability to mimic the use of directory streams (type DIR) must also be

implemented. A directory stream is returned by the `opendir()` call and is repeatedly passed into the `readdir()` function in order to step through the associated directory's contents. This functionality can be duplicated on a Windows system by using the `HANDLE` type and the related system calls.

However, first a struct must be defined as type `DIR`:

```
typedef struct {
    char name[255];
    HANDLE findHandle;
} DIR;
```

The 'name' data member of this struct is used to hold the path name of the directory currently associated with this `DIR`, and the 'findHandle' data member holds a handle to the item currently indexed in that directory.

The replacement for the `opendir()` function is written like so:

```
DIR *opendir( char *c )
{
    DIR *x;
    short i;
    WIN32_FIND_DATA findData;

    x = malloc( sizeof( DIR ) );

    strcpy( x->name, c );
    for( i=0; x->name[i] != '\0'; i++ )
        if( x->name[i] == '/' )
            x->name[i] = '\\';

    strcat( x->name, "\\*" );

    x->findHandle = FindFirstFile( x->name, &findData );
    if( x->findHandle != INVALID_HANDLE_VALUE )
        return( x );
    else
    {
        free( x );
        return( 0 );
    }
}
```

The first thing this function does is allocate memory for a DIR type. Then it takes the string passed in as a parameter and changes any back slashes to forward slashes, in order to alter UNIX-style path names to DOS-style path names. Then a '*' is appended to the string since the function will be looking at all the contents of the directory. The call to FindFirstFile returns a HANDLE, which is stored in the DIR struct, indicating the first file in the given directory. If the returned HANDLE is valid, we return the DIR structure; otherwise, the memory to the struct is freed, and a NULL value is returned.

When a valid DIR struct has been returned, it can be passed into a call to readdir() to step through the contents of that directory. The redefined readdir() functions is:

```
struct dirent *readdir( DIR *x )
{
    struct dirent *d;
    WIN32_FIND_DATA findData;
    BOOL suc=0;
    short i;

    d = malloc( sizeof( struct dirent ) );

    if( x && !(x->findHandle) )
    {
        x->findHandle = FindFirstFile( x->name, &findData );

        if( x->findHandle != INVALID_HANDLE_VALUE ) suc = 1;
    }
    else
    {
        if( x->findHandle != INVALID_HANDLE_VALUE )
            if( FindNextFile( x->findHandle, &findData ) )
                suc = 1;
    }

    if( suc )
        strcpy( d->d_name, findData.cFileName );
    else
    {
        free( d );
        d = NULL;
    }

    return( d );
}
```

This function first allocates memory for a dirent structure, which will hold the name of the next item in the directory and will be returned. The DIR structure passed as a parameter is checked for validity and then either FindFirstFile() or FindNextFile() is called depending upon whether the 'findHandle' of that struct has been initialized. The relevant data is stored in the WIN32_FIND_DATA data variable 'findData' and the variable 'suc' is set if an item was found. Assuming an item was found, the name of the item is copied into the 'd_name' field of the dirent struct. Otherwise, a NULL value is returned.

The closedir() redefinition simply calls FindClose() on the findHandle field of the modified DIR struct and frees the memory held by the struct.

```
void closedir( DIR *x )
{
    FindClose( x->findHandle );
    free( x );
    x = 0;
}
```

Unlike the redefined i/o functions above, it is not necessary to #define these custom routines in a header file. Since these functions are not normally defined under Windows, we merely need to supply the source file and an appropriate header file with the function prototypes in order for them to link properly with the original X Windows source code.

select() and system()

Other Unix functions that need to be supported are the select() and system() calls. The select() function is used to test if a file descriptor is ready for reading or writing. The incompatibility arises when this call is used to test stdin or stdout. Since neither stdin nor stdout are defined under Windows, this function is undefined when testing these descriptors. The solution is to redefine the function to operate properly under these conditions:

```

int  winselect( int x, struct fd_set *fdset, fd_set *y, fd_set *z,
const          struct timeval *timeout )
{
    long n;
    short j, k, l;
    INPUT_RECORD recBuf[10];
    char newline = '\n', space = ' ';

    GetNumberOfConsoleInputEvents( gConsoleIn, &n );

    if( n )
    {
        ReadConsoleInput( gConsoleIn, recBuf, 10, &n );
        for( j=0; j<n; j++ )
        {
            if(recBuf[j].EventType == KEY_EVENT &&
                recBuf[j].Event.KeyEvent.bKeyDown)
            {
                for( l=1; l<=recBuf[j].Event.KeyEvent.wRepeatCount; l++ )
                {
                    if(gConsoleBuf[gBufLength] ==
                        recBuf[j].Event.KeyEvent.uChar.AsciiChar)
                    if( gConsoleBuf[gBufLength] == kHardReturn )
                    {
                        WriteConsole( gConsoleOut, &newline, 1, &num, vd );
                        gBufLength++;
                        return(1);
                    }
                    else if( gConsoleBuf[gBufLength] == kBackspace )
                    {
                        if( gBufLength > 0 )
                        {
                            WriteConsole( gConsoleOut, &gConsoleBuf[gBufLength], 1,
                                &num, vd);
                            WriteConsole( gConsoleOut, &space, 1, &num, vd );
                            WriteConsole( gConsoleOut, &gConsoleBuf[gBufLength], 1,
                                &num, vd);
                            gBufLength--;
                            gConsoleBuf[gBufLength] = '\0';
                        }
                    }
                    else
                    {
                        WriteConsole( gConsoleOut, &gConsoleBuf[gBufLength], 1,
                            &num, vd);
                        gBufLength++;
                    }
                }
            }
        }
    }
}

```

```

    return(0);
}

```

Currently, this function redefinition only checks the console for available input, descriptors other than stdin are not supported.

Through the use of ReadConsoleInput(), this function copies the console input into the global gConsoleBuf array and takes the appropriate actions based upon the input received. Only if a hard return is received does this function return 'true', indicating that there is input waiting in the buffer. Any other input besides a hard return is saved in the buffer for later retrieval. If a backspace character is entered into the console, several steps must be taken to ensure that the previous character in the console is erased; first, the backspace character must be echoed to the console to move the insertion point back one character, then a space character is entered to blank whatever character is there, then the backspace is entered once more to place the insertion point at the correct location.

The system() function is supported under Windows through DOS. However, an incompatibility with its use under Unix and Windows arises when path names are passed into the call. Since Unix uses forward slashes ('/') as a directory delimiter and DOS uses a back slash ('\'), Unix path names passed into the function will not be processed properly by DOS. The solution to this problem is to redefine the system() call as:

```

void winsystem( char *string )
{
    short i;
    char *s;

    s = malloc( strlen(string) + 1 );

    for( i=0; string[i] != '\0'; i++ )
        if( string[i] == '/' )
            s[i] = '\\';
        else
            s[i] = string[i];

    s[i] = 0x00;

    system( s );
}

```

}

This function parses the string passed in as an argument, changing all the forward slashes to back slashes in order to conform any Unix path names to DOS path names. The corrected string is then passed to the default `system()` function to be processed normally by DOS. A potential problem could arise is, for some reason, a forward slash passed into the system call was meant to be interpreted literally as a forward slash rather than a directory separator since this function indiscriminately alters them to back slashes. The implementation of this function brings up another question in regards to compatibility between disparate systems: How to support the functionality of Unix system utilities under Windows? The simple solution to this problem is to create an executable of the same name as the Unix utility which performs an identical function. For example, if the Unix code made a call to `system()` as "`system( "rm file.txt")`", the intention would be to delete the file named 'file.txt'. The similar system utility under Microsoft Windows is 'erase', but how to execute 'erase' by sending 'rm' to the DOS command line? The solution is to use a batch file. Creating a batch file named 'rm' containing the commands:

```
@echo off
erase %1
```

will emulate a call to 'rm' on a Unix system. Turning the echo off ensures that the commands will not be displayed to the user. Next, 'erase' is called with %1 as a parameter. %1 is a variable holding the first parameter which was entered with the call to this batch file.

As with other redefinitions of functions already defined under Windows, both the `select()` and `system()` functions must be #defined to the compatible functions `winselect()` and `winsystem()`:

```
#define system    winsystem
#define select    winselect
```

While the Unix functions implemented above are vital to securing compatibility between the systems, the larger task of supporting the X Windows function calls on Microsoft Windows still remains. With these two GUI systems, X Windows and Microsoft Windows, we have two independent systems which operate on similar principles: the ability to support graphical windows and to accept and act upon various user events directed at these windows.

Simulating X Windows functions under Microsoft Windows

The X Windows calls discussed and implemented under Microsoft Windows in this paper are:

```
XSetForeground()  
XSetLineAttributes()  
XDrawLine()  
XFlush()  
XDrawPoint()  
XDrawString()  
XClearWindow()  
XCheckMaskEvent()  
XSetFunction()  
XQueryPointer()  
XPutBackEvent()  
DefaultScreen()  
BlackPixel()  
DefaultColormap()  
XAllocNamedColor()  
XSetBackground()  
XOpenDisplay()  
RootWindow()  
XCreateSimpleWindow()  
XCreateGC()  
XSelectInput()  
XMapWindow()  
XWindowEvent()  
XSetDashes()
```

At the core of the differences between X Windows and Microsoft Windows are the differing data types each uses to store information for the windows and events. X

Windows uses the data types Window, GC, and Display to track windows, so these types must be defined and supported under the Microsoft Windows GUI for source code written for X Windows to work transparently under it. Microsoft Windows has its own data types for much the same purposes: HWND and HDC. Since both the Microsoft Windows HWND and X Windows Window types are used to hold a 'handle' or pointer to an individual window, we can simply define the Window type in the Microsoft environment:

```
typedef HWND Window;
```

Similarly, the GC and HDC types are used to point to graphic contexts and may be viewed as parallels:

```
typedef HDC GC;
```

The Display type used on X Windows systems has no real counterpart in Microsoft Windows, but there is still the need to define the type in order for the X Windows code to compile without complaint. Defining it as an unsigned long seems as good a solution as any:

```
typedef unsigned long Display;
```

It should be noted that there is the assumption made that these structures will not be dereferenced in the Unix code. If they were dereferenced in the code, the expected data members would not be available and compile errors would result.

With these types defined, the work of bringing the X Windows specific calls over to Microsoft Windows can begin.

XOpenDisplay(), RootWindow(), DefaultScreen(), and XCreateSimpleWindow()

The first step in writing an X Windows program is to call the function XOpenDisplay(), which returns a pointer to a Display type. A NULL value is returned if

the display is inaccessible. On Microsoft Windows, this is a non-issue since remote displays will not be used, so it is basically an empty wrapper call which returns a non-zero value:

```
Display *XOpenDisplay( char *s )
{
    Display *x;

    x = malloc( sizeof( Display ) );

    return( x );
}
```

Since the Display type has been declared as an unsigned long, there is not much in the way of memory allocation taking place here. However, a Display type is allocated in the case that it is changed to a different type or struct at a later time. This Display type is passed to many X Windows function calls, and it will consequently also be passed into those calls simulated on a Microsoft Windows system; however, it will never be used for any practical purpose in those simulated calls.

The functions RootWindow() and DefaultScreen() are also specific to X Windows systems and have no comparable functionality on a Microsoft Windows system.

Consequently, these are defined as empty wrapper calls that return the proper types:

```
Display *DefaultScreen( Display *dis )
{
    return( dis );
}

Window RootWindow( Display *dis, Display *dis2 )
{
    Window win;

    return( win );
}
```

Often, a call to RootWindow() is passed as the second parameter to a call to XCreateSimpleWindow():

```
Window XCreateSimpleWindow( Display *dis, Window win, short x, short y,
                           short width, short height,
```

```

short borderWidth, long border, long back )
{
    Window hwnd;
    GC  hdc;

    hwnd = CreateWindow ( "",          // window class name
                        "",          // window caption
                        WS_OVERLAPPEDWINDOW, // window style
                        x,           // initial x position
                        y,           // initial y position
                        width,        // initial x size
                        height,       // initial y size
                        NULL,         // parent window handle
                        NULL,         // window menu handle
                        gInstance,    // program instance handle
                        NULL) ;      // creation parameters

    ShowWindow (hwnd, SW_SHOWNA);
    UpdateWindow(hwnd);
    hdc = GetDC( hwnd );
    SetBkColor( hdc, back );
    ReleaseDC( hwnd, hdc );

    return( hwnd );
}

```

This function call creates a window on an X Windows system with the given properties and returns a pointer to it. To simulate this call under Microsoft Windows, the `CreateWindow()` function is called, passing in the appropriate arguments. Next, `ShowWindow()` and `UpdateWindow()` are called to ensure that the window is drawn to the screen. Additionally, the window's background color is set to the given value. A handle to the window (i.e. the retyped `Window` type) is returned so the original X Windows program can keep track of it as normal.

The `gInstance` global variable is of type `HANDLE` and is defined in an initialization function discussed later. In brief, it defines many of the attributes of a window.

`XCreateGC()`, `XSetForeground()`, `XSetLineAttributes()`, `XSetBackground()`, and `XSetFunction()`

X Windows requires a graphics context to be passed into many of its function calls

and provides the `XCreateGC()` function to enable the programmer to get the graphics context for a given window. Similarly, Microsoft Windows requires a drawing context to be passed to many of its API calls and provides the `GetDC()` call to return a context for a window. Therefore, a simple wrapper is written for the `XCreateGC()` function:

```
GC XCreateGC( Display *dis, Window h, unsigned long x, XGCValues *v )
{
    return( GetDC( h ) );
}
```

Whenever the graphics context is passed into an X Windows call, it will conveniently provide the drawing context needed to simulate the call's functionality with the Microsoft Windows API.

Once a window is registered to the system and drawn on the screen, there are a number of X Windows calls that can alter its appearance or functionality. Foremost among these functions are `XSetForeground()`, `XSetLineAttributes()` and `XSetBackground()`, which will change the foreground color, line style, and background colors of a window, respectively. While there are corresponding calls under Microsoft Windows, there are quite a few differences in how they are implemented, leading to several problems that must be solved.

First of all, the `XSetBackground()` function is defined easily enough as:

```
void XSetBackground( Display *dis, GC hdc, unsigned long color )
{
    SetBkColor( hdc, color );
}
```

This function is easy to simulate because both X Windows and Microsoft Windows define colors as 32-bit integer values, and both systems pass these types of values to the function that sets the background color of a window.

Unfortunately, setting the foreground color is not so easy. While the X Windows call `XSetForeground()` is implemented the same way as `XSetBackground()`, Microsoft

Windows is not so accommodating. In order to set the properties (e.g. color, style, etc.) of primitives drawn to a window under Microsoft Windows, a data type of type HPEN must be allocated and passed into the function SelectObject(). One possible solution would be to simply allocate a new HPEN data type each time XSetForeground() or XSetLineAttributes() was called, disposing of any previously used HPEN type. While this solution would work, poor performance would result if the program rapidly called these functions while expecting fast response. A better solution is to allocate all necessary HPEN types ahead of time and then just swap them in and out via the SelectObject() call, thus avoiding the repeated overhead of allocating and deallocating memory. Unfortunately, the code has no way of knowing which colors and line patterns, and combinations thereof, will be used, but it is able to store these HPEN objects instead of disposing of them. To keep track of these HPEN types, a new data type, Pen, will be defined to hold the HPEN pointer as well as some other data members:

```
struct Pen {
    unsigned long color;
    unsigned int style;
    HPEN penHandle;
    struct Pen *next;
};

typedef struct Pen Pen;
```

This will enable the use of a linked list to keep track of all allocated Pen types, and provide easy access to the crucial attributes of the HPEN object in the color and style data members.

With this new data member defined, consider this definition of the XSetForeground() function:

```
void XSetForeground(Display *dis, GC hdc, unsigned long color )
{
    Pen *cur;
    BOOL done = FALSE;

    gCurColor = color;
```

```

cur = gPenList;

while( cur != 0 && !done )
{
    if( cur->color == gCurColor && cur->style == gCurStyle )
    {
        SelectObject( hdc, cur->penHandle );
        done = TRUE;
    }
    else
        cur = cur->next;
}

if( !cur ) /* create new pen object */
{
    cur = (Pen*)malloc( sizeof( Pen ) );
    cur->penHandle = CreatePen( gCurStyle, 0, gCurColor );
    cur->style = gCurStyle;
    cur->color = gCurColor;
    cur->next = gPenList;
    gPenList = cur;
    SelectObject( hdc, gPenList->penHandle );
}
}

```

where gCurColor and gCurStyle are global variables which hold the current color and style settings, respectively. The variable gPenList is also a global which is a pointer to a linked list of all allocated Pen objects. After changing the current color to the one specified, this function searches the gPenList linked list for a Pen object with a color and style which matches the current color and style set within the program. If one is found, it is installed into the system with a call to SelectObject(). If not, then the appropriate Pen object is created, placed in the linked list, and then installed into the system.

A similar algorithm is used for the XSetLineAttributes() function, the only difference being that the current style is set at the beginning rather than the color:

```

void XSetLineAttributes( Display *dis, GC hdc, unsigned int width,
                        int style, int cap, int join )
{
    Pen *cur;
    BOOL done = FALSE;

    gCurStyle = style;

    cur = gPenList;

```

```

while( cur != 0 && !done )
{
    if( cur->color == gCurColor && cur->style == gCurStyle )
    {
        SelectObject( hdc, cur->penHandle );
        done = TRUE;
    }
    else
        cur = cur->next;
}

if( !cur ) /* create new pen object */
{
    cur = (Pen*)malloc( sizeof( Pen ) );
    cur->penHandle = CreatePen( gCurStyle, 0, gCurColor );
    cur->style = gCurStyle;
    cur->color = gCurColor;
    cur->next = gPenList;
    gPenList = cur;
    SelectObject( hdc, gPenList->penHandle );
}
}

```

This approach avoids the overhead of continuously allocating and deallocating memory for each call to either of these functions, but could pose a performance hit in cases where many hundreds or thousands of color and style combinations are used. A better solution for these types of programs would be a balanced tree of some sort, but the linked list approach works well for most cases.

`XSetFunction()` is another call which dictates how graphics are drawn to the screen under X Windows. This function sets the bitwise logical operation in a graphical context. Before implementing the actual function, though, several constants must be defined:

```

#define GXxor      R2_XORPEN
#define GXcopy     R2_COPYPEN

```

By matching up the X Windows constants with their Microsoft Windows counterparts the operation of `XSetFunction()` is made completely transparent:

```

void XSetFunction( Display *dis, GC hdc, int function )

```

```

{
    SetROP2( hdc, function );
}

```

The integer constant passed in as a parameter is passed through to SetROP2(), which has the same functionality under Microsoft Windows as XSetFunction() has under X Windows.

XDrawPoint(), XDrawLine(), and XDrawString()

With these basic functions implemented, attention can now be given to the functions that actually draw primitives in the graphical window: XDrawPoint(), XDrawLine(), and XDrawString().

```

void XDrawPoint( Display *dis, Window hwnd, GC hdc, int x, int y )
{
    SetPixel( hdc, x, y, gCurColor );
}

```

The pixel at coordinate (x,y) in the window hwnd is highlighted in the current fore color by a call to XDrawPoint(). The corresponding Microsoft Windows call is SetPixel(), to which we pass the relevant arguments along with gCurColor, which holds the current color as specified by the most recent call to XSetForeground().

XDrawString() is much the same:

```

void XDrawString( Display *dis, Window hwnd, GC hdc, int x, int y,
                  char *string, int length )
{
    TextOut( hdc, x, y, string, length );
}

```

TextOut() is the Microsoft Windows equivalent to XDrawString(), so it's merely a matter of passing in the correct parameters. The character string 'string' of length 'length' is

drawn to the window at coordinates (x,y).

XDrawLine() requires two Windows calls to emulate its function due to differences in implementation:

```
void XDrawLine( Display *dis, Window hwnd, GC hdc, int x1, int y1,
               int x2, int y2 )
{
    POINT pt;

    MoveToEx( hdc, x1, y1, &pt );
    LineTo( hdc, x2, y2 );
}
```

XDrawLine() draws a line from point (x1,y1) to point (x2,y2). There is no direct correlation in the Microsoft Windows API, so two calls must be used, MoveToEx() and LineTo(). The former call sets the 'pen' down at point (x1,y1) and the latter draws a line in the current color and style from that point to point (x2,y2) in the window.

XFlush(), XClearWindow(), and XAllocNamedColor()

With X Windows, a call to XFlush() empties the request buffer, displaying all queued requests. This functionality is unnecessary under Microsoft Windows since all drawing requests are immediately dispatched, but the function must still be supported for full compatibility; an empty wrapper function is declared:

```
void XFlush( Display *dis )
{
}

}
```

The X Windows function XClearWindow() blanks the window passed in as a parameter. Again, there is no direct correlation with a Microsoft Windows function call, but the same effect can be generated by:

```

void XClearWindow( Display *dis, Window hwnd )
{
    RECT rect;
    GC  hdc;

    hdc = GetDC( hwnd );
    GetClientRect (hwnd, &rect) ;
    Rectangle( hdc, rect.left, rect.top, rect.right, rect.bottom );
    ReleaseDC( hwnd, hdc );
}

```

First, the HDC of the window must be determined by the GetDC() call, since this function does not take a GC as a parameter. Then the interior rectangle of the window is returned by GetClientRect() and stored in a variable of type RECT. Next the window is blanked by a call to Rectangle() which draws a filled rectangle at the given coordinates (in this case, the left, top, right, and bottom members of the RECT data type) to the window. Lastly, the HDC must be released with ReleaseDC().

X Windows requires the programmer to allocate colors in the display's colormap, which can hold a discrete number of colors at any given time. Microsoft Windows is similar in that a palette of colors is held by the system, and the entries may be changed by the programmer; however, it does not require the 'registration' of colors that X Windows does. These registration functions must still be supported in this library of compatibility functions. First, the X Windows data type XColor must be defined and supported:

```

typedef struct {
    unsigned long pixel;
    unsigned short red, green, blue;
    char flags;
    char pad;
} XColor;

```

Most notable among the registration functions is XAllocNamedColor(), which allows the programmer to pass in the name of a color in order to retrieve its RGB value.

The simulated function is listed here with a subset of the available colors:

```
void XAllocNamedColor(Display *dis, Colormap cmap, char *s,
                    XColor *ccell, XColor *color)
{
    if( strcmp( s, "wheat" ) == 0 )
        color->pixel = RGB( 245, 222, 179 );
    else if( strcmp( s, "forestgreen" ) == 0 )
        color->pixel = RGB( 35, 142, 35 );
    else if( strcmp( s, "goldenrod" ) == 0 )
        color->pixel = RGB( 219, 219, 112 );
    else if( strcmp( s, "plum" ) == 0 )
        color->pixel = RGB( 234, 173, 234 );
    else if( strcmp( s, "blue" ) == 0 )
        color->pixel = RGB( 0, 0, 255 );
    else if( strcmp( s, "grey" ) == 0 )
        color->pixel = RGB( 192, 192, 192 );
    else if( strcmp( s, "yellow" ) == 0 )
        color->pixel = RGB( 255, 255, 0 );
    else if( strcmp( s, "red" ) == 0 )
        color->pixel = RGB( 255, 0, 0 );
    else if( strcmp( s, "yellowgreen" ) == 0 )
        color->pixel = RGB( 153, 204, 50 );
    else if( strcmp( s, "darkturquoise" ) == 0 )
        color->pixel = RGB( 0, 206, 209 );
    else if( strcmp( s, "violetred" ) == 0 )
        color->pixel = RGB( 208, 32, 144 );
    else if( strcmp( s, "orangered" ) == 0 )
        color->pixel = RGB( 255, 69, 0 );
    else if( strcmp( s, "green" ) == 0 )
        color->pixel = RGB( 0, 255, 0 );
    else if( strcmp( s, "black" ) == 0 )
        color->pixel = RGB( 0, 0, 0 );
}
```

When a match is found for the string passed in as a parameter, the XColor struct is filled in with the appropriate values. While the string comparisons may be slow, note that this function need only be called once at the beginning of the program for each color and therefore should not have any discernable impact on application performance. The function RGB() returns a 32-bit integer value composed of the three 8-bit parameters representing the red, green, and blue components of the desired color.

Handling Events: XCheckMaskEvent()

The most divergent implementation between X Windows and Microsoft Windows

is how both systems handle user events. Although they both use an event-loop model, each has its own event structures and ways for limiting which events are reported by the system. Since the X Windows API is being implemented on top of Microsoft Windows, the solution requires that we create the X Windows event model through the use of the Microsoft Windows APIs. Naturally, as a first step, the X Windows event structs must be defined in the Microsoft environment:

```
typedef struct {
    int type;
    DWORD time;
    int x, y;
    unsigned int state;
    unsigned int button;
} XButtonEvent;

typedef struct {
    int type;
    int x, y;
    unsigned int state;
} XMotionEvent;

typedef struct {
    int type;
    unsigned int state;
    unsigned int keycode;
} XKeyEvent;

typedef union _XEvent {
    int type;
    XButtonEvent xbutton;
    XMotionEvent xmotion;
    XKeyEvent xkey;
} XEvent;
```

The XEvent structure is a union of the three event type structures, XKeyEvent, XMotionEvent, and XButtonEvent. The programmer can then pass an instance of an XEvent into a call to XCheckMaskEvent(), which will then fill in the data members of the given XEvent dependant upon the type of event waiting in the queue. The 'type' data member of the XEvent can be checked to determine what that event type was. The following valid constants for the 'type' member must be defined:

```
#define Expose      WM_PAINT
```

```
#define KeyPress    WM_KEYDOWN
#define ButtonPress WM_LBUTTONDOWN
#define ButtonRelease WM_LBUTTONUP
#define MotionNotify WM_MOUSEMOVE
```

Other constants which are needed are:

```
#define Button1    1
#define Button2    2
#define Button3    3

#define ShiftMask  0x01
#define ControlMask 0x02
#define Mod1Mask   0x04
```

The first three constants fill in the 'button' member of an XButtonEvent and the last three are used in the 'state' member of any type of event to indicate which modifier keys were down at the time of the event.

The most complicated part of handling X Windows event types under Microsoft Windows is how the programmer tells the system which events to report. Under X Windows a 32-bit integer is manipulated with bitwise operators and passed into XCheckMaskEvent(), each bit representing an event type. Microsoft Windows is slightly more complicated since it handles many dozens of event types; it requires that the programmer pass in two 16-bit values to PeekMessage(), representing a range of events to report. The solution to this problem is to examine the bits of the event mask passed into XCheckMaskEvent() and build a corresponding event range applicable to the Microsoft Windows environment:

```
BOOL XCheckMaskEvent(Display *dis, long int mask, XEvent *evt)
{
    BOOL result;
    MSG msg;
    unsigned short lowMask=0xFFFF, hiMask=0x0000;
    long int maskCopy = mask;
    unsigned short event=0;

    while( maskCopy )
    {
        if( maskCopy & ExposureMask ) {
```

```

        event = WM_PAINT; maskCopy -= ExposureMask; }
    else if( maskCopy & KeyPressMask ) {
        event = WM_KEYDOWN; maskCopy -= KeyPressMask; }
    else if( maskCopy & KeyReleaseMask ) {
        event = WM_KEYUP; maskCopy -= KeyReleaseMask; }
    else if( maskCopy & ButtonPressMask ) {
        event = WM_MOUSEFIRST; maskCopy -= ButtonPressMask; }
    else if( maskCopy & ButtonReleaseMask ) {
        event = WM_MOUSELAST; maskCopy -= ButtonReleaseMask; }
    else if( maskCopy & Button1MotionMask ) {
        event = WM_MOUSEMOVE; maskCopy -= Button1MotionMask; }
    else if( maskCopy & Button2MotionMask ) {
        event = WM_MOUSEMOVE; maskCopy -= Button2MotionMask; }
    else if( maskCopy & Button3MotionMask ) {
        event = WM_MOUSEMOVE; maskCopy -= Button3MotionMask; }

    if( event ) {
        if( event < lowMask ) lowMask = event;
        if( event > hiMask ) hiMask = event;
        event = 0;
    }
    else
        maskCopy = 0;
}
.
.
.

```

Next, this range of events indicated by the variables 'lowMask' and 'hiMask' are passed into PeekMessage(), which reports if an event within the desired range is in the queue:

```

.
.
.
result = PeekMessage( &msg, NULL, lowMask, hiMask, PM_REMOVE );
.
.
.

```

If PeekMessage() returns 'TRUE', then the 'msg' parameter is checked to see what the event type was and is used to fill in the XEvent record accordingly:

```

.
.
.
if( result )
{
    switch( msg.message )
    {

```

```

case WM_LBUTTONDOWN:
    ((XButtonEvent *) evt)->type = ButtonPress;
    ((XButtonEvent *) evt)->button = Button1;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_MBUTTONDOWN:
    ((XButtonEvent *) evt)->type = ButtonPress;
    ((XButtonEvent *) evt)->button = Button2;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_RBUTTONDOWN:
    ((XButtonEvent *) evt)->type = ButtonPress;
    ((XButtonEvent *) evt)->button = Button3;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_LBUTTONUP:
    ((XButtonEvent *) evt)->type = ButtonRelease;
    ((XButtonEvent *) evt)->button = Button1;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;

```

```

case WM_MBUTTONDOWN:
    ((XButtonEvent *) evt)->type = ButtonRelease;
    ((XButtonEvent *) evt)->button = Button2;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_RBUTTONDOWN:
    ((XButtonEvent *) evt)->type = ButtonRelease;
    ((XButtonEvent *) evt)->button = Button3;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_MOUSEMOVE:
    ((XMotionEvent *) evt)->type = MotionNotify;
    ((XMotionEvent *) evt)->x = LOWORD(msg.lParam);
    ((XMotionEvent *) evt)->y = HIWORD(msg.lParam);
    break;
case WM_KEYDOWN:
    ((XKeyEvent *) evt)->type = KeyPress;
    ((XKeyEvent *) evt)->keycode = (TCHAR)msg.wParam;
    ((XKeyEvent *) evt)->state = 0;
    break;
default:
    evt->type = 0;
    break;
}

TranslateMessage( &msg );
DispatchMessage( &msg );
}

return( result );
}

```

The calls to `TranslateMessage()` and `DispatchMessage()` are necessary since the system may need to act upon certain events. Lastly, the value of 'result' is returned so that the original

program will know whether or not a valid event was found. Despite the discrepancies between the two systems methods for reporting events, this translation between the two works remarkably well.

Chapter III

Conclusion

As can be seen from the above implementation, creating a translation library which allows code created for one system to work on another is not a trivial process. The question of the value of such a library may then arise; since so much time must be put into writing a translation library, would it not be much simpler to port the source code from one system to another? A simple answer to such a question is that the translation library must only be coded once and then may be reused repeatedly to take any piece of code written for the source system and allow it to work on the destination system. An additional advantage is that once the translation library is in hand, a programmer need not learn the API for an entirely new system but is instead able to continue writing code within a system with which he is already familiar.

So with all of this said, what are the disadvantages to using translation libraries for cross-platform source code compatibility? For applications which are very computation intensive or which have a heavy reliance on fast, responsive video, it may be worth the effort to do a more conventional port of the program to different systems to gain whatever slight performance advantage might be lost with the remapping which takes place with a translation library. However, the X Windows to Microsoft Windows translation library which is discussed here was used to take the link discovery tool from its Unix development platform over to Microsoft Windows NT and Windows 95 with very little performance degradation, taking into account the differences in hardware, despite the program's need to display a connected graph consisting of thousands of points which could be translated,

rotated, or scaled by the user in real time. Undoubtedly, the program would operate somewhat faster with code tuned to the Microsoft environment. It was decided early in the project, however, that the extra time and effort spent to write and maintain two versions of the Link Discovery Tool code was not worth the marginal gain in performance.

However, it should be noted that the specific X Windows to Microsoft Windows translation library outlined in the second chapter of this thesis has several shortcomings. It will be ineffectual in translating some programs due to features which are implemented under Unix but have no direct or indirect corollary under Microsoft Windows. Specific examples of some of these unsupported features are Unix pipes and the `fork()` function call. While Microsoft Windows does allow a program to launch other programs, there is no contingency for enabling a process to spawn a copy of itself dynamically as there is under Unix. Because of this, any Unix programs which rely upon the `fork()` call, such as network applications, will not be able to be translated with the given translation library detailed in this thesis. Another contingency not accounted for by this translation library is the use of Unix runtime environment variables. These variables will be undefined under Microsoft Windows when the program is compiled with the translation library. Defining such environment variables as global variables under Windows could provide a simple solution for some, but this would undoubtedly not be an adequate solution for all situations. Although the given translation library was very successful at its intended purpose, transparently taking the X Windows source code of a program heavily dependant upon system and video performance to the Microsoft Windows platform, it can be seen that it is not a solution for all cases and reveals some challenging obstacles in the use of translation libraries.

Another disadvantage with translation libraries in general is that any assembly code within the program will need to be rewritten for the target platform since the translation library can do nothing to aid in taking it across platforms. But most programs tuned with

assembly are most likely very dependant upon system performance and would be very likely to fall into the category of programs which would benefit the most from being hand ported to different systems.

While in many cases a translation library will work transparently (i.e. without modification) on the source code, some concessions may be necessary. For example, since Unix and Windows systems have different hardware keycodes for the keyboard, any reference to the system-specific keycodes in the link discovery tool source code mentioned earlier had to be changed to constants held in a header file; the corresponding Windows keycodes could then be placed in a different header file. After this fix, the only change necessary in the source code was either to place an `#ifdef` statement in the code to allow the compilers to include the correct header file at compile time.

In the many cases a translation library will perform an adequate port of the program from one system to another with little or no effort once the library has been implemented.

The other way in which translation libraries must be examined is within the scope of other cross-platform solutions which are available. Today, the most widely accepted and anticipated cross-platform solution is the Java programming language. Although both Java and translation libraries could be termed "middle-ware", they have vastly different approaches. While translation libraries create a mapping between the various platform-dependent system calls, Java is an entirely new programming language with its own calls and, in fact, its own byte-code. Since these Java byte-codes must be interpreted by the system at run-time, Java programs have a much lower performance than programs which are compiled in native machine code. Therefore, it is fairly obvious that translation libraries will perform noticeably better than raw Java byte-code despite its layer of mapping and redirection. However, Java is overcoming this shortfall with the development of Just-In-Time compilers which will compile a Java program into native code and therefore achieve much improved performance.

Additionally, while Java requires the programmer to learn an entirely new language, translation libraries allow the programmer to work in an environment which is already familiar. The main disadvantage for translation libraries is that it is a one-way mapping from one system to another and will only take code from the source system to the destination system. Java is a well-defined language which compiles down to specified byte-codes which can be implemented on any system, after which that system is able to run any Java program created on any other Java capable system.

Overall, translation libraries can be valuable tools which can easily take programs from one system to another once the initial effort of the implementation is done. The widest use for these libraries would be with programs that have already been written for one system, but which need to be ported to another quickly and with little loss in performance. Translation libraries are currently the only solution which can take such unmodified system-specific source code from one platform to another.

Index

BlackPixel	19
Button1	32, 34
Button1MotionMask	33
Button2	32, 34-35
Button2MotionMask	33
Button3	32, 34-35
Button3MotionMask	33
ButtonPress	32, 34
ButtonPressMask	33
ButtonRelease	32, 34-35
ButtonReleaseMask	33
Closedir	6, 12, 15

Console	10
ControlMask	32, 34-35
CreateWindow	22
DefaultColormap	19
DefaultScreen	19-21
DIR	12-14
DispatchMessage	35
Display	20-21
Expose	31
ExposureMask	33
FindClose	15
FindFirstFile	13-15
FindNextFile	14-15

GC	20
Getc	6-7, 11-12
GetClientRect	29
GetDC	22-23, 29
Gets	6-7, 9-10, 12
Java	39-40
KeyPress	32, 35
KeyPressMask	33
KeyReleaseMask	33
LineTo	28
Mod1Mask	32, 34-35
MotionNotify	32, 35
MoveToEx	28

Opendir	6, 12-13
PeekMessage	32-33
Pen	24-25
Printf	6-8, 12
ReadConsoleInput	7, 9, 17
Readdir	6, 12-14
Rectangle	29
ReleaseDC	22, 29
RGB	30
RootWindow	19-21
Select	6, 11, 15, 18
SelectObject	24-26
SetBkColor	22-23

SetPixel	27
SetROP2	27
ShiftMask	32, 34-35
ShowWindow	22
Stdin	15, 17
Stdlib.h	12
Stdout	15
System	6, 15, 17-18
TextOut	27
TranslateMessage	35
Ungetc	6-7, 11
UpdateWindow	22
Window	20, 22

Wingetc	11-12
Wingets	9, 12
Winprintf	7, 12
Winselect	16, 18
Winsystem	17-18
Winungetc	11-12
WM_KEYDOWN	32-33, 35
WM_KEYUP	33
WM_LBUTTONDOWN	32, 34
WM_LBUTTONUP	32, 34
WM_MBUTTONDOWN	34
WM_MBUTTONUP	35
WM_MOUSEFIRST	33

WM_MOUSELAST	33
WM_MOUSEMOVE	32-33, 35
WM_PAINT	31, 33
WM_RBUTTONDOWN	34
WM_RBUTTONUP	35
WriteConsole	7-11
XAllocNamedColor	19, 28-30
XButtonEvent	31-32
XCheckMaskEvent	19, 30-32
XClearWindow	19, 28-29
XColor	29-30
XCreateGC	19, 22-23
XCreateSimpleWindow	19-21

XDrawLine	19, 27-28
XDrawPoint	19, 27
XDrawString	19, 27
XEvent	31, 33
XFlush	19, 28
XKeyEvent	31
XMapWindow	19
XMotionEvent	31
XOpenDisplay	19-21
XPutBackEvent	19
XQueryPointer	19
XSelectInput	19
XSetBackground	19, 22-23

XSetDashes	19
XSetForeground	19, 22-24, 27
XSetFunction	19, 26-27
XSetLineAttributes	19, 22-25
XWindowEvent	19

References

Petzold, Charles. *Programming Windows 95*. Redmond: Microsoft Press, 1996.

Naughton, Patrick and Herbert Schildt. *Java: The Complete Reference*. Berkeley: Osborne McGraw-Hill, 1997.

Nye, Adrian, ed. *XLib Reference Manual*. Vol. 2. The Definitive Guide to the X Window System. O'Reilly and Associates, Inc 1992.

Appendix

```

/* This is the main.c file where the Windows program begins execution. *
 * After initializing variables and creating a window definition, the   *
 * program calls main(), the starting point of the X Window program    *
 * being translated, passing in any parameters. */

#define _MAIN_
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
#include "XWinFunc.h"
#include "globals.h"
#include "resource.h"

unsigned long curColor;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int APIENTRY WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                     LPSTR lpszCmdParam, int nCmdShow)
{
    static char szAppName[] = "Application" ;
    HWND        hwnd ;
    MSG         msg ;
    WNDCLASS     wndclass ;
    LPDWORD num;
    LPVOID      vd;
    char *arg[2];
    char buf[100];
    short  i, j, argc = 1;

    gInstance = hInstance;

    arg[0] = "app.exe";
    arg[1] = strtok( lpszCmdParam, " " );
    if( arg[1] )
        argc = 2;

    if( !hPrevInstance )
    {
        wndclass.style           = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
        wndclass.lpfnWndProc     = WndProc ;
        wndclass.cbClsExtra      = 0 ;
        wndclass.cbWndExtra      = 0 ;
        wndclass.hInstance       = hInstance ;
        wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
        wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
        wndclass.hbrBackground   = GetStockObject (BLACK_BRUSH) ;
        wndclass.lpszMenuName     = MAKEINTRESOURCE( IDR_MENU1 ) ;
        wndclass.lpszClassName   = szAppName ;

        RegisterClass (&wndclass) ;
    }

    /* init consoleBuf */

```

```

for( i=0; i<=49; i++ )
    consoleBuf[i] = '\0';

AllocConsole();
consoleOut = GetStdHandle( STD_OUTPUT_HANDLE );
consoleIn = GetStdHandle( STD_INPUT_HANDLE );

main( argc, arg );

FreeConsole();
SendMessage( hwnd, WM_CLOSE, 0, 0L );

}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam,
                          LPARAM lParam)
{
    HDC          hdc ;
    PAINTSTRUCT ps ;
    RECT         rect ;
    HPEN         curPen;
    HBRUSH       curBrush;
    HMENU        hMenu;

    switch (message)
    {
        case WM_CREATE:
            curPen = CreatePen(PS_SOLID, 0, RGB( 0, 0, 0 ) );
            hdc = GetDC( hwnd );
            SelectObject( hdc, curPen );
            SelectObject( hdc, GetStockObject( BLACK_BRUSH ) );
            SetBkMode( hdc, TRANSPARENT );
            SetTextColor( hdc, RGB(255, 255, 255 ) );
            ReleaseDC( hwnd, hdc );
            break;
        case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;
            GetClientRect (hwnd, &rect) ;
            EndPaint (hwnd, &ps) ;
            break;
        case WM_COMMAND:
            hMenu = GetMenu( hwnd );

            switch( LOWORD(wParam) )
            {
                case ID_FILE_EXIT:
                    menuBuf[0] = 'q';
                    menuBuf[1] = '\0';
                    menuChosen = 1;
                    WriteConsole( consoleOut, "q\n", 2, &num, vd );
                    break;
                case ID_COMMAND_VERBOSE:
                    menuBuf[0] = 'v';
                    menuBuf[1] = '\0';
                    menuChosen = 1;
                    WriteConsole( consoleOut, "v\n", 2, &num, vd );
            }
    }
}

```

```

        break;
    }
    break;
case WM_DESTROY :
    PostQuitMessage( 0 );
    break ;
}

return DefWindowProc( hwnd, message, wParam, lParam );
}

```

```

/* This file is globals.h, a file that holds all global variables      *
 * internal to the translation engine. Since it is included in several  *
 * files, steps must be taken to ensure that the global variables are  *
 * decalared as external in all files but main.c and that some are     *
 * initialized. If the file is main.c, there is no prefix, otherwise   *
 * 'extern' is specified. Similiarly, if the file is main.c, those     *
 * that need be initialized to zero are done so. */

#ifdef _MAIN_
#define PREFIX
#define ASSIGN = 0
#else
#define PREFIX extern
#define ASSIGN
#endif

PREFIX unsigned long gCurColor ASSIGN;
PREFIX unsigned int gCurStyle ASSIGN;
PREFIX int curWidth;
PREFIX HANDLE consoleOut ASSIGN;
PREFIX HANDLE consoleIn ASSIGN;
PREFIX int printLength;
PREFIX char printBuffer[100];
PREFIX BOOL menuChosen ASSIGN;
PREFIX char menuBuf[20];
PREFIX long num; //added types
PREFIX LPVOID vd;
PREFIX char consoleBuf[100];
PREFIX short bufLength ASSIGN;
PREFIX HANDLE gInstance;
PREFIX HANDLE gFindHandle ASSIGN;
PREFIX POINT gPoint;
PREFIX Pen *gPenList ASSIGN;

```

```

/* This file is wincompat.h, the only file that need be included in      *
 * the X Window source code. It redefines the necessary functions and      *
 * includes any application specific constants. */

#include "windows.h"
#include "XWinFunc.h"
#include "globals.h"

#undef  getc

#define system    winsystem
#define printf    winprintf
#define gets      BlockingReadConsole
#define select    winselect
#define getc      wingetc
#define ungetc    winungetc

/* examples of application specific constants */

#define kEndLine      '\r'
#define kZoomIn      38
#define kZoomOut     40

```

```

/* This file is XWinFunc.h and includes all necessary definitions of
 * X Window types, structures, and function prototypes that need to
 * be defined and simulated in order for native X Window code to
 * compile and run under the Microsoft Windows environment. */

#define LineSolid      PS_SOLID
#define LineOnOffDash PS_DOT
#define LineDoubleDash PS_DASHDOTDOT

#define JoinRound      PS_JOIN_ROUND
#define JoinBevel      PS_JOIN_BEVEL
#define JoinMiter      PS_JOIN_MITER

#define CapButt        PS_ENDCAP_FLAT
#define CapRound       PS_ENDCAP_ROUND
#define CapProjecting  PS_ENDCAP_SQUARE

#define Expose         WM_PAINT
#define KeyPress       WM_KEYDOWN
#define ButtonPress    WM_LBUTTONDOWN
#define Button1        1
#define Button2        2
#define Button3        3
#define ButtonRelease  WM_LBUTTONUP
#define MotionNotify   WM_MOUSEMOVE
#define ShiftMask      0x01
#define ControlMask    0x02
#define Mod1Mask       0x04
#define ExposureMask   0x00000001
#define ButtonReleaseMask 0x00000002
#define StructureNotifyMask 0x00000004
#define KeyPressMask    0x00000008
#define KeyReleaseMask  0x00000010
#define ButtonPressMask  0x00000020
#define Button1MotionMask 0x00000040
#define Button2MotionMask 0x00000080
#define Button3MotionMask 0x00000100

#define GXxor          R2_XORPEN
#define GXcopy         R2_COPYPEN

typedef struct {
    unsigned long pixel;
    unsigned short red, green, blue;
    char flags;
    char pad;
} XColor;

typedef struct {
    int type;
    DWORD time;
    int x, y;
    unsigned int state;
    unsigned int button;
} XButtonEvent;

```

```

typedef struct {
    int type;
    int x, y;
    unsigned int state;
} XMotionEvent;

typedef struct {
    int type;
    unsigned int state;
    unsigned int keycode;
} XKeyEvent;

typedef union _XEvent {
    int type;
    XButtonEvent xbutton;
    XMotionEvent xmotion;
    XKeyEvent xkey;
} XEvent;

typedef struct {
    int x;
} Colormap;

typedef struct {
    int x;
} XGCValues;

struct dirent {
    char d_name[255];
};

typedef struct {
    int tv_sec, tv_usec;
} timeval;

typedef struct {
    char name[255];
    HANDLE findHandle;
} DIR;

struct Pen {
    unsigned long color;
    unsigned int style;
    HPEN penHandle;
    struct Pen *next;
};

typedef struct Pen Pen;

typedef HWND Window;
typedef long Display;
typedef HDC GC;

void XSetForeground(Display *dis, GC hdc, unsigned long color );
void XSetLineAttributes( Display *dis, GC hdc, unsigned int width,
                        int style, int cap, int join );

```

```

void XDrawLine( Display *dis, Window hwnd, GC hdc, int x1, int y1,
               int x2, int y2 );
void XFlush( Display *dis );
void XDrawPoint( Display *dis, Window hwnd, GC hdc, int x, int y );
void XDrawString( Display *dis, Window hwnd, GC hdc, int x, int y,
                 char *string, int length );
void XClearWindow( Display *dis, Window hwnd );
BOOL XCheckMaskEvent( Display *dis, long int mask, XEvent *evt );
void XSetFunction( Display *dis, GC hdc, int function );
void XQueryPointer( Display *dis, Window hwnd, Window *root,
                  Window *child, int *rx, int *ry, int *wx, int *wy,
                  unsigned int *kb );
void XPutBackEvent( Display *dis, XEvent *evt );
Display *DefaultScreen( Display *dis );
unsigned long BlackPixel( Display *dis, Display *dis2 );
Colormap DefaultColormap( Display *dis, Display *dis2 );
void XAllocNamedColor( Display *dis, Colormap cmap, char *s,
                     XColor *ccell, XColor *color );
void XSetBackground( Display *dis, GC hdc, unsigned long color );
char *wingets( char *buf );
Display *XOpenDisplay( char *s );
Window RootWindow( Display *dis, Display *dis2 );
Window XCreateSimpleWindow( Display *dis, Window win, short x, short y,
                          short width, short height,
                          short borderWidth, unsigned long border,
                          unsigned long back );
GC XCreateGC( Display *dis, Window hwnd, unsigned long x,
             XGCValues *v );
void XSelectInput( Display *dis, Window win, int x );
void XMapWindow( Display *dis, Window win );
void XWindowEvent( Display *dis, Window win, short mask, XEvent *evt );
void XSetDashes( Display *dis, GC gc, int x, unsigned char *dash, int y );
void winprintf( char *string, ... );
int winselect( int x, struct fd_set *fdset, fd_set *y, fd_set *z,
              const struct timeval *timeout );
char wingetc( struct _iobuf *z );
void winungetc( char x, struct _iobuf *z );
DIR *opendir( char *c );
struct dirent *readdir( DIR *x );
void closedir( DIR *x );
double cbrt( double x );
void winsystem( char *string );

```

```

/* This file is XWinFunc.c and contains all the actual function      *
 * definitions of the simulated X Window functions under Microsoft    *
 * Windows. */                                                        *

#define __XWINFUNC__

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <sys/types.h>
#include "voserand.h"
#include <windows.h>
#include <stdarg.h>
#include "XWinFunc.h"
#include "globals.h"

void XSetForeground(Display *dis, GC hdc, unsigned long color )
{
    Pen *cur;
    BOOL done = FALSE;

    gCurColor = color;

    cur = gPenList;

    while( cur != 0 && !done )
    {
        if( cur->color == gCurColor && cur->style == gCurStyle )
        {
            SelectObject( hdc, cur->penHandle );
            done = TRUE;
        }
        else
            cur = cur->next;
    }

    if( !cur ) //create new pen object
    {
        cur = (Pen*)malloc( sizeof( Pen ) );
        cur->penHandle = CreatePen( gCurStyle, 0, gCurColor );
        cur->style = gCurStyle;
        cur->color = gCurColor;
        cur->next = gPenList;
        gPenList = cur;
        SelectObject( hdc, gPenList->penHandle );
    }
}

void XSetLineAttributes( Display *dis, GC hdc, unsigned int width,
                        int style, int cap, int join )
{
    Pen *cur;
    BOOL done = FALSE;

    gCurStyle = style;

```

```

cur = gPenList;

while( cur != 0 && !done )
{
    if( cur->color == gCurColor && cur->style == gCurStyle )
    {
        SelectObject( hdc, cur->penHandle );
        done = TRUE;
    }
    else
        cur = cur->next;
}

if( !cur )    //create new pen object
{
    cur = (Pen*)malloc( sizeof( Pen ) );
    cur->penHandle = CreatePen( gCurStyle, 0, gCurColor );
    cur->style = gCurStyle;
    cur->color = gCurColor;
    cur->next = gPenList;
    gPenList = cur;
    SelectObject( hdc, gPenList->penHandle );
}

void XDrawLine( Display *dis, Window hwnd, GC hdc, int x1, int y1,
               int x2, int y2 )
{
    POINT pt;

    MoveToEx( hdc, x1, y1, &pt );
    LineTo( hdc, x2, y2 );
}

void XFlush( Display *dis )
{
}

void XDrawPoint( Display *dis, Window hwnd, GC hdc, int x, int y )
{
    SetPixel( hdc, x, y, gCurColor );
}

void XDrawString( Display *dis, Window hwnd, GC hdc, int x, int y,
                 char *string, int length )
{
    TextOut( hdc, x, y, string, length );
}

void XClearWindow( Display *dis, Window hwnd )
{
    RECT rect;
    GC  hdc;

    hdc = GetDC( hwnd );

```

```

    GetClientRect (hwnd, &rect) ;
    Rectangle( hdc, rect.left, rect.top, rect.right, rect.bottom );
    ReleaseDC( hwnd, hdc );
}

BOOL XCheckMaskEvent(Display *dis, long int mask, XEvent *evt)
{
    BOOL result;
    MSG msg;
    unsigned short lowMask=0xFFFF, hiMask=0x0000;
    long int maskCopy = mask;
    unsigned short event=0;

    while( maskCopy )
    {
        if( maskCopy & ExposureMask ) {
            event = WM_PAINT; maskCopy -= ExposureMask; }
        else if( maskCopy & KeyPressMask ) {
            event = WM_KEYDOWN; maskCopy -= KeyPressMask; }
        else if( maskCopy & KeyReleaseMask ) {
            event = WM_KEYUP; maskCopy -= KeyReleaseMask; }
        else if( maskCopy & ButtonPressMask ) {
            event = WM_MOUSEFIRST; maskCopy -= ButtonPressMask; }
        else if( maskCopy & ButtonReleaseMask ) {
            event = WM_MOUSELAST; maskCopy -= ButtonReleaseMask; }
        else if( maskCopy & Button1MotionMask ) {
            event = WM_MOUSEMOVE; maskCopy -= Button1MotionMask; }
        else if( maskCopy & Button2MotionMask ) {
            event = WM_MOUSEMOVE; maskCopy -= Button2MotionMask; }
        else if( maskCopy & Button3MotionMask ) {
            event = WM_MOUSEMOVE; maskCopy -= Button3MotionMask; }

        if( event ) {
            if( event < lowMask ) lowMask = event;
            if( event > hiMask ) hiMask = event;
            event = 0;
        }
        else
            maskCopy = 0;
    }

    result = PeekMessage( &msg, NULL, lowMask, hiMask, PM_REMOVE );

    if( result )
    {
        switch( msg.message )
        {
            case WM_LBUTTONDOWN:
                ((XButtonEvent *) evt)->type = ButtonPress;
                ((XButtonEvent *) evt)->button = Button1;
                ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
                ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
                ((XButtonEvent *) evt)->time = msg.time;
                ((XButtonEvent *) evt)->state = 0;

```

```

    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_MBUTTONDOWN:
    ((XButtonEvent *) evt)->type = ButtonPress;
    ((XButtonEvent *) evt)->button = Button2;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_RBUTTONDOWN:
    ((XButtonEvent *) evt)->type = ButtonPress;
    ((XButtonEvent *) evt)->button = Button3;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_LBUTTONUP:
    ((XButtonEvent *) evt)->type = ButtonRelease;
    ((XButtonEvent *) evt)->button = Button1;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;
    if( GetKeyState(VK_SHIFT) < 0 )
        ((XButtonEvent *) evt)->state |= ShiftMask;
    if( GetKeyState(VK_CONTROL) < 0 )
        ((XButtonEvent *) evt)->state |= ControlMask;
    if( GetKeyState(VK_MENU) < 0 )
        ((XButtonEvent *) evt)->state |= ModlMask;
    break;
case WM_MBUTTONUP:
    ((XButtonEvent *) evt)->type = ButtonRelease;
    ((XButtonEvent *) evt)->button = Button2;
    ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
    ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
    ((XButtonEvent *) evt)->time = msg.time;
    ((XButtonEvent *) evt)->state = 0;

```

```

        if( GetKeyState(VK_SHIFT) < 0 )
            ((XButtonEvent *) evt)->state |= ShiftMask;
        if( GetKeyState(VK_CONTROL) < 0 )
            ((XButtonEvent *) evt)->state |= ControlMask;
        if( GetKeyState(VK_MENU) < 0 )
            ((XButtonEvent *) evt)->state |= Mod1Mask;
        break;
    case WM_RBUTTONDOWN:
        ((XButtonEvent *) evt)->type = ButtonRelease;
        ((XButtonEvent *) evt)->button = Button3;
        ((XButtonEvent *) evt)->x = LOWORD(msg.lParam);
        ((XButtonEvent *) evt)->y = HIWORD(msg.lParam);
        ((XButtonEvent *) evt)->time = msg.time;
        ((XButtonEvent *) evt)->state = 0;
        if( GetKeyState(VK_SHIFT) < 0 )
            ((XButtonEvent *) evt)->state |= ShiftMask;
        if( GetKeyState(VK_CONTROL) < 0 )
            ((XButtonEvent *) evt)->state |= ControlMask;
        if( GetKeyState(VK_MENU) < 0 )
            ((XButtonEvent *) evt)->state |= Mod1Mask;
        break;
    case WM_MOUSEMOVE:
        ((XMotionEvent *) evt)->type = MotionNotify;
        ((XMotionEvent *) evt)->x = LOWORD(msg.lParam);
        ((XMotionEvent *) evt)->y = HIWORD(msg.lParam);
        //((XMotionEvent *) evt)->state = msg.wParam;
        break;
    case WM_KEYDOWN:
        ((XKeyEvent *) evt)->type = KeyPress;
        ((XKeyEvent *) evt)->keycode = (TCHAR)msg.wParam;
        ((XKeyEvent *) evt)->state = 0;
        break;
    default:
        evt->type = 0;
        break;
}

TranslateMessage( &msg );
DispatchMessage( &msg );
}

return( result );
}

void XSetFunction( Display *dis, GC hdc, int function )
{
    SetROP2( hdc, function );
}

void XQueryPointer(Display *dis, Window hwnd, Window *root,
                  Window *child, int *rx, int *ry, int *wx, int *wy,
                  unsigned int *kb)
{
    POINT point;
    RECT rect;

```

```

GetCursorPos( &point );

ScreenToClient( hwnd, &point );

*wx = point.x;
*wy = point.y;
}

void XPutBackEvent( Display *dis, XEvent *evt )
{

}

Display *DefaultScreen( Display *dis )
{
    return( dis );
}

unsigned long BlackPixel( Display *dis, Display *dis2 )
{
    return( RGB(0, 0, 0) );
}

Colormap DefaultColormap( Display *dis, Display *dis2 )
{

}

void XAllocNamedColor( Display *dis, Colormap cmap, char *s, XColor
*ccell, XColor *color)
{
    if( strcmp( s, "wheat" ) == 0 )
        color->pixel = RGB( 245, 222, 179 );
    else if( strcmp( s, "forestgreen" ) == 0 )
        color->pixel = RGB( 35, 142, 35 );
    else if( strcmp( s, "goldenrod" ) == 0 )
        color->pixel = RGB( 219, 219, 112 );
    else if( strcmp( s, "plum" ) == 0 )
        color->pixel = RGB( 234, 173, 234 );
    else if( strcmp( s, "blue" ) == 0 )
        color->pixel = RGB( 0, 0, 255 );
    else if( strcmp( s, "grey" ) == 0 )
        color->pixel = RGB( 192, 192, 192 );
    else if( strcmp( s, "yellow" ) == 0 )
        color->pixel = RGB( 255, 255, 0 );
    else if( strcmp( s, "red" ) == 0 )
        color->pixel = RGB( 255, 0, 0 );
    else if( strcmp( s, "yellowgreen" ) == 0 )
        color->pixel = RGB( 153, 204, 50 );
    else if( strcmp( s, "darkturquoise" ) == 0 )
        color->pixel = RGB( 0, 206, 209 );
    else if( strcmp( s, "violetred" ) == 0 )
        color->pixel = RGB( 208, 32, 144 );
    else if( strcmp( s, "orangered" ) == 0 )
        color->pixel = RGB( 255, 69, 0 );
    else if( strcmp( s, "green" ) == 0 )

```

```

        color->pixel = RGB( 0, 255, 0 );
    else if( strcmp( s, "black" ) == 0 )
        color->pixel = RGB( 0, 0, 0 );
    else
        MessageBox( NULL, "Cannot allocate unknown color.", "Application",
                    MB_OK );
}

void XSetBackground( Display *dis, GC hdc, unsigned long color )
{
    SetBkColor( hdc, color );
}

char *BlockingReadConsole( char *out )
{
    long n, length=0; short j, k, l;
    INPUT_RECORD recBuf[10];
    char newline = '\n', space = ' ', buffer[100];
    BOOL done = 0;

    if( bufLength > 0 )
    {
        for( k=0; k<bufLength; k++ )
        {
            out[k] = consoleBuf[k];
            consoleBuf[k] = '\0';
        }
        bufLength = 0;
    }
    else
        while( !done )
        {
            ReadConsoleInput( consoleIn, recBuf, 10, &n );
            for( j=0; j<n; j++ )
            {
                if( recBuf[j].EventType == KEY_EVENT &&
                    recBuf[j].Event.KeyEvent.bKeyDown )
                {
                    for( l=1; l<=recBuf[j].Event.KeyEvent.wRepeatCount; l++ )
                    {
                        if( buffer[length] = recBuf[j].Event.KeyEvent.uChar.AsciiChar )
                            if( buffer[length] == '\r' )
                            {
                                WriteConsole( consoleOut, &newline, 1, &num, vd );
                                for( k=0; k<length; k++ )
                                    out[k] = buffer[k];
                                out[length] = '\0';
                                done = 1;
                            }
                        else if( buffer[length] == 8 )
                        {
                            if( length > 0 )
                            {
                                WriteConsole( consoleOut, &buffer[length], 1, &num, vd );
                                WriteConsole( consoleOut, &space, 1, &num, vd );
                                WriteConsole( consoleOut, &buffer[length], 1, &num, vd );
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        length--;
    }
}
else
{
    WriteConsole( consoleOut, &buffer[length], 1, &num, vd );
    length++;
}
}
}
}

return( out );
}

Display *XOpenDisplay( char *s )
{
    Display *x;

    x = malloc( sizeof( Display ) );

    return( x );
}

Window XCreateSimpleWindow( Display *dis, Window win, short x, short y,
                           short width, short height,
                           short borderWidth, unsigned long border,
                           unsigned long back )
{
    Window hwnd;
    GC hdc;

    hwnd = CreateWindow ( "Window Name",          // window class name
                          "Application",          // window caption
                          WS_OVERLAPPEDWINDOW,    // window style
                          x,                      // initial x position
                          y,                      // initial y position
                          width,                  // initial x size
                          height,                 // initial y size
                          NULL,                   // parent window handle
                          NULL,                   // window menu handle
                          gInstance,              // program instance handle
                          NULL ) ;                // creation parameters

    ShowWindow (hwnd, SW_SHOWNORMAL) ;
    UpdateWindow(hwnd);

    hdc = GetDC( hwnd );
    SetBkColor( hdc, back );
    ReleaseDC( hwnd, hdc );

    return( hwnd );
}

```

```

Window RootWindow( Display *dis, Display *dis2 )
{
    Window win;

    return( win );
}

GC XCreateGC( Display *dis, Window hwnd, unsigned long x, XGCValues *v
)
{
    return( GetDC( hwnd ) );
}

void XSelectInput( Display *dis, Window win, int x)
{
}

void XMapWindow( Display *dis, Window win)
{
}

void XWindowEvent ( Display *dis, Window win, short mask, XEvent *evt)
{
}

void XSetDashes( Display *dis, GC hdc, int x, unsigned char *dash, int y)
{
}

void winprintf( char *string, ... )
{
    va_list ap;
    int x, num=0;
    void *a[10];
    char ch = '%';
    char *pos, *oldPos, *ptr, buf[20], newStr[200]="";

    va_start( ap, string );

    pos = strchr( string, ch );
    oldPos = string;

    while( pos )
    {
        strncat( newStr, oldPos, pos-oldPos );

        switch( *(pos+1) )
        {
            case 'i':
            case 'd':
                x = va_arg(ap, int );
                _itoa( x, buf, 10 );

```

```

        strcat( newStr, buf );
        break;
    case 's':
        ptr = va_arg(ap, char*);
        strcat(newStr, ptr);
        break;
    }

    oldPos = pos+2;
    pos = strchr( pos+1, ch );
}

strcat( newStr, oldPos );

WriteConsole( consoleOut, newStr, strlen(newStr), &num, vd );

va_end( ap );
}

int winselect( int x, struct fd_set *fdset, fd_set *y, fd_set *z,
               const struct timeval *timeout )
{
    long n; short j, k, l;
    INPUT_RECORD recBuf[10];
    char newline = '\n', space = ' ';

    GetNumberOfConsoleInputEvents( consoleIn, &n );

    if( menuChosen )
    {
        menuChosen = 0;
        consoleBuf[bufLength] = menuBuf[0];
        bufLength++;
        return(1);
    }

    if( n )
    {
        ReadConsoleInput( consoleIn, recBuf, 10, &n );
        for( j=0; j<n; j++ )
        {
            if( recBuf[j].EventType == KEY_EVENT &&
                recBuf[j].Event.KeyEvent.bKeyDown )
            {
                for( l=1; l<=recBuf[j].Event.KeyEvent.wRepeatCount; l++ )
                {
                    if( consoleBuf[bufLength] =
                        recBuf[j].Event.KeyEvent.uChar.AsciiChar )
                    if( consoleBuf[bufLength] == '\r' )
                    {
                        WriteConsole( consoleOut, &newline, 1, &num, vd );
                        bufLength++;
                        return(1);
                    }
                }
                else if( consoleBuf[bufLength] == 8 )
                {

```

```

        if( bufLength > 0 )
        {
            WriteConsole( consoleOut, &consoleBuf[bufLength], 1, &num,
                           vd );
            WriteConsole( consoleOut, &space, 1, &num, vd );
            WriteConsole( consoleOut, &consoleBuf[bufLength], 1, &num,
                           vd );

            bufLength--;
            consoleBuf[bufLength] = '\\0';
        }
    }
    else
    {
        WriteConsole( consoleOut, &consoleBuf[bufLength], 1, &num,
                       vd );

        bufLength++;
    }
}

}

}

return(0);
}

char wingetc( struct _iobuf *z )
{
    char x;
    short i;

    x = consoleBuf[0];

    for( i=0; i<bufLength; i++ )
    {
        consoleBuf[i] = consoleBuf[i+1];
        consoleBuf[i+1] = '\\0';
    }

    bufLength--;

    return( x );
}

void winungetc( char x, struct _iobuf *z )
{
    short i;
    char t;

    consoleBuf[bufLength+1] = '\\0';

    for( i=bufLength-1; i>=0; i-- )
    {
        consoleBuf[i] = consoleBuf[i+1];
    }
}

```

```

    bufLength++;

    consoleBuf[0] = x;
}

DIR *opendir( char *c )
{
    DIR *x;
    short i;
    WIN32_FIND_DATA findData;

    x = malloc( sizeof( DIR ) );

    strcpy( x->name, c );
    for( i=0; x->name[i] != '\0'; i++ )
        if( x->name[i] == '/' )
            x->name[i] = '\\';

    strcat( x->name, "\\*" );

    x->findHandle = FindFirstFile( x->name, &findData );
    if( x->findHandle != INVALID_HANDLE_VALUE )
        return( x );
    else
    {
        free( x );
        return( 0 );
    }
}

struct dirent *readdir( DIR *x )
{
    struct dirent *d;
    WIN32_FIND_DATA findData;
    BOOL suc=0;
    short i;

    d = malloc( sizeof( struct dirent ) );

    if( !(x->findHandle) )
    {
        x->findHandle = FindFirstFile( x->name, &findData );

        if( x->findHandle != INVALID_HANDLE_VALUE ) suc = 1;
    }
    else
    {
        if( x->findHandle != INVALID_HANDLE_VALUE )
            if( FindNextFile( x->findHandle, &findData ) )
                suc = 1;
    }

    if( suc )
        strcpy( d->d_name, findData.cFileName );
    else
    {

```

```

    free( d );
    d = NULL;
}

return( d );
}

void closedir( DIR *x )
{
    FindClose( x->findHandle );
    free( x );
    x = 0;
}

double cbrt( double x )
{
    return( pow(x,1./3) );
}

void winsystem( char *string )
{
    short i;

    for( i=0; string[i] != '\0'; i++ )
        if( string[i] == '/' )
            string[i] = '\\';

    system( string );
}

```

Vita

Kirk Hedden was born in Greenville, SC on July 22, 1973. He attended the public schools of District 7 in Spartanburg County, graduating from Spartanburg High School in May of 1991. He entered Wofford College of Spartanburg, SC during August of 1991 where he received a Bachelor of Arts in English and a Bachelor of Arts in Computer Science in May of 1995. In August of 1995 he entered the Master's program in Computer Science at the University of Tennessee, Knoxville. He was employed by the University throughout his studies there, in the Network Services department until September of 1996 when he transferred to the University's Laboratory for Information Technologies, officially receiving the Master's degree in August of 1997.

He is currently employed at Mid-South Data Systems in Greenville, SC as a Programmer/Analyst.