Wipro UWIN

Developer Guide

Wipro Technologies
No.8, 7th Main, I Block,
Koramangala,
Bangalore – 560 034,
India
uwin@wipro.com
http://www.wipro.com/uwin

CONTENTS

Chapter 1
Introduction

What is UWIN?

UWIN is a Unix to Windows migration toolkit, providing source level compatibility between Windows and Unix operating systems. It provides a Unix runtime environment for Microsoft Windows NT/9x. The UWIN SDK (Software Development Kit) provides a Unix development environment for Windows. It provides more than 300 header files and more than 60 commonly used Unix shared libraries.

System Requirements

Software requirements

- UWIN Base toolkit + UWIN SDK
- Microsoft Visual C/C++ 4.0 or higher or GNU C/C++ compiler
- Microsoft Windows NT 4.0 or higher (Workstation or Server) or Microsoft Windows 95/98

Hardware requirements

- Intel x86, Pentium, Pentium Pro and compatible systems
- 30-100MB of available hard-disk space

Technical Support

Wipro Technology Solutions provides technical support from 8:30am to 6:00pm IST. E-mail is the best method for contacting technical support.

E-mail:	Mailto:uwin@wipro.com
Phone:	+91-080-5530035, (India)
Fax:	+91-80-5530086 (India)
	+1-810-963-0715 (USA)
Web site:	http://www.wipro.com/uwin/
Address:	Wipro Ltd, No 8, 7th main, 1st Block, Koramangala, Bangalore – 560 034, INDIA.

Chapter 2
UWIN Architecture

UWIN consists of two dynamically linked libraries namely, the posix.dll and **ast52.dll/ast54.dll**. The ast52.dll is the earlier version of ast54.dll that was distributed with 1.6x versions of UWIN. In addition to the libraries, a service named **UWIN Master Service** (UMS) runs in the system or administrator account. There is also another service, **UWIN Client Service** (UCS) which is installed for each user account in the system.

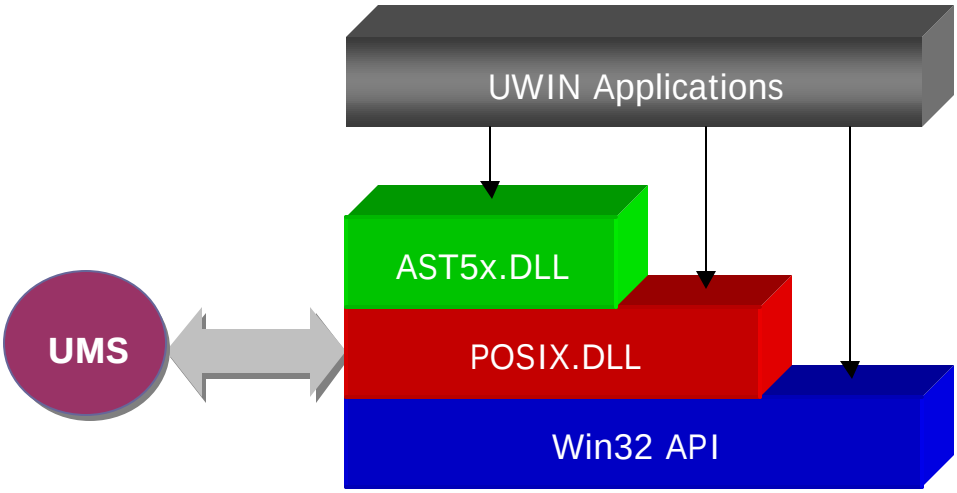

Figure 1: UWIN Architecture

The posix.dll implements all the Unix system calls and all the functions to handle console and serial line support, sockets, Unix permissions and other commonly used mechanisms such as memory mapping, IPC, and others. It implements malloc(), realloc(), and free() interface using the vmalloc library. The vmalloc library provides an interface to walk over all memory segments allocated by a process that is required for implementing fork() functionality described later.

The ast library provides better functionality than the standard C library by having its own version of some of the C functions. It provides a portable application-programming interface that is used by UWIN. The interface to this library is called libast.a, for compatibility with its name on UNIX systems. This library also provides its own version of the stdio library based on calls to Sfio

(safe and fast io). The Sfio library makes calls to posix.dll rather than making direct calls to the WIN32 API so that pathnames are correctly mapped. The use of Sfio also provides a simple solution to the carriage return- newline problem as described under *File Management - Line delimiters* below.

UWIN maintains an open file table in a memory-mapped region, which is shared by all the currently active UWIN processes. This region is writable by all UWIN processes so that the appropriate information can be shared between them. Even though all processes have read and write access to the shared segment, secure access to kernel objects in Windows NT is not compromised by this model because a process must have access rights to an object to use it; knowing its address or value doesn't give additional access rights. The shared memory is used to store process information, IPC information, file tables, and others.

Unix applications that have been compiled on Windows using UWIN (called UWIN applications) use the functionality provided by ast5x.dll and posix.dll. The application can also directly use the functionality provided by the Win32 subsystem as depicted in figure 1 above. The posix.dll utilizes the functionality of UMS to provide Unix impersonation support (setuid).

File Management

Handles Vs File Descriptors

Each file is identified by a file descriptor in Unix and by a HANDLE in Windows. Handles are analogous to file descriptors except that they are unordered, so that a per process table is needed to maintain the ordering. The posix.dll library performs the mapping between handles and file descriptors. Usually, each file descriptor has one handle associated with it. In some cases, two handles may be associated with a single file descriptor. An example of this is a console that is open for reading and writing which uses separate handles for reading and writing.

Pathname mapping

The posix.dll library handles the mapping between UNIX pathnames and WIN32 pathnames. Many UNIX programs assume that the pathnames which do not begin with a / are relative pathnames. In addition, only / is recognized as a delimiter. Changing to another drive does not change the root directory. The posix.dll library maps all file names it encounters. If the file name begins with a / and the first component is a single letter, then this letter is taken as the

drive letter. Thus, the UNIX filename /d/bin/date gets translated to d:\bin\date. The file name mapping routine also recognizes special file names such as /dev/tty and /dev/null. A '/'not followed by a drive letter is mapped to the directory where UWIN has been installed so that programs that embed absolute pathnames for files in /bin, /tmp, /dev, and /etc work without modification. The file name mapping also solves the problem caused due to the special treatment by the Win32 interface of the names like aux, com1, com2, nul, and filenames consisting of these names followed by any suffix.

The pathname mapping function also takes care of exact case matching on file systems that require it. WIN32 API lacks support for pathname case distinction. It is not uncommon to have files named Makefile and makefile in the same directory in UNIX. UWIN handles case distinction by calling the WIN32 CreateFile() function both with and without the FILE_FLAG_POSIX_SEMANTICS flag.

UWIN also supports Universal Naming Convention (UNC). UWIN uses names of the form //hostname/filename to access files on a given host. UWIN also adds the mount point /sys as a way of accessing the system directory. This makes it easier to write shell scripts that are portable across Windows machines.

Pathname mapping introduces one problem. Passing file names as arguments to native Windows utilities is now more difficult since it understands DOS style names and not UNIX names. A library routine (/bin/unixpath) was added to return a DOS name, given a UNIX name.

Winpath

You can get the equivalent Windows path for any given Unix path using the /bin/winpath command.

"winpath" can be used to convert unix path to WIN32 pathnames. This is equivalent to **-H** option of typeset. winpath is inverse of unixpath command.

Options :

-q, --quote Quote the Win32 pathname if necessary so that it can be used by the shell.

Examples :

$ winpath /e/uwin
e:\uwin

```
$ winpath /sys
d:\winnt\system32

$ winpath /tmp/uwin_log
D:\Program Files\UWIN2.9/tmp\uwin_log
$
$ winpath /d/notes.txt
D:\notes.txt
$
```

Line Delimiters

Windows uses the DOS convention of a two-character sequence <cr><nl> (carriage return-new line) to signify the end of each line in a text file. UNIX uses a single character <nl> to signify the end of line. The result is that file processing is more complex than it is with UNIX. There are separate modes for opening a file as text and binary with the Microsoft C library. Binary mode treats the file as a sequence of bytes. Text mode strips off each <cr> in front of each new-line as the file is read, and inserts a <cr> in front of each <nl> as the file is written. When a file is explicitly opened for reading as a text file, a Sfio library discipline for read()and lseek()can be inserted on the stream to change all <cr><nl> sequences to <nl>. The lseek() discipline uses logical offsets so that the removal of <cr> character is transparent. UWIN does not provide a discipline to change <nl> to <cr><nl> since Windows NT/9x utilities work without the <cr>s. The <cr>s could be inserted by a filter such as sed, if required.

File I/O and Control

On Unix, all types of files (e.g., disk files, device files) use a uniform name space in the file system. A file descriptor represents each file and all these file descriptors are treated in the same manner. This is not the case on Windows. UWIN hides all these issues. When one process invokes another, all the handles associated with file descriptors will be passed to the process invoked.

In UWIN, if a file is unlinked while in use by one or more processes, its directory entry is removed. However the disk space used by the file is released only when the last process closes the file.

UWIN provides partial support for files larger than two gigabytes. The underlying NTFS file system supports 64 bit file offsets. However, the size of off_t is stored as a 32bit integer because some programs would otherwise break. The type off64_t is defined and

the functions ftruncate64(), lseek64(), and truncate64() have been implemented. The current stat structure actually fills in a 64-bit file size, but only the low 32 bits are accessible. The current version of Sfio supports 64 bit file offsets, but this option has not been enabled as of now.

Special Files

UWIN supports both hard links and symbolic links. Hard links are restricted regular files that are on the same drive. As in Unix, the links are symmetric in the sense that each one refers to the same data, and there is a reference count that gives the number of links to the given file.

Unlike hard links, symbolic links do not have the restrictions. They are not symmetric and there is no reference count. The symbolic link is essentially a file that contains the name of the file that it refers to. Thus, deleting the linked file will link it to a non-existent file.

The implementation of symbolic links and hard links depends on the type of file system. In NTFS, hard links are supported by the underlying file system and UWIN uses this mechanism for its implementation. As a result, hard links to files are recognized by the native NT commands as well as by UWIN commands. Symbolic links, on the other hand, are created as a separate NTFS data stream. They are visible only to the UWIN commands and native programs will find these as zero length files.

On a FAT file-system, the files in the .links directory keep track of all the hard links to files in that drive. The native system recognizes all linked files but one of these files is visible as a zero length file. The primary problem created by this relates to executables that are linked under separate names. As a result, UWIN makes a separate copy of each link for files that end with the .exe suffix. Symbolic links on FAT file systems are implemented as regular files with a special header and the system file attribute set for these files.

Process Management

The process model of UWIN and Windows differ from each other. The WIN32 subsystem does not have an equivalent function for fork() or for the exec*()family of functions. There is a single primitive in Windows, named CreateProcess() which creates a new process but does not perform the operation of overlaying the current process with a new program as exec*() requires.

Implementation of fork()

The fork() system call is implemented by creating a new process with the same startup information as the current process. Before executing main(),the data and stack of the parent process is copied into child process. Handles owned by the parent process are duplicated into the child process. The address space of the executable and DLLs built using UWIN are also replicated. Address space replication adds some overhead that can be minimized by avoiding fork() in the program. The alternatives to fork() are vfork().

The WIN32 process heap, the data sections and private heaps of non-UWIN DLLs will not be replicated because UWIN does not control the inheritance of objects that were not allocated through UWIN calls.

To minimize the effects of differences in process models of UWIN and Windows it is advised to use UWIN versions of DLLs as much as possible.

Implementation of exec()

UWIN implementation of exec()causes the child process to be re-parented to the grandparent and the process that calls exec*() to exit. getpid()in the child returns the process id of the process that invoked the exec*() function. To prevent pid of the process that invoked the exec*() function from being used again by WIN32, a handle to the process is kept open by the grandparent process.

Because the Win32 API CreateProcess() function does not have an overlay flag, two processes need to be created in order to do both fork() and exec*(). posix.dll provides a spawn*() family of functions that combines the functionality of fork()/exec*(). All functions such as system() and popen() that create processes are programmed using this interface. On most UNIX systems, the spawn*() family is written using fork() or vfork() and exec*(). The uwin_spawn() function has a provision to specify process startup information and options to CreateProcess() that cannot be specified with fork() and exec().

Process Identification

On Windows NT, the UWIN process id will be the same as the Windows NT process id unless the process has been created by exec. On Windows 9x, the pid is a huge value and hence become negative when converted to pid_t type. UWIN has a way of mapping the actual id with the one that is returned by getpid() function.

Process Invocation

When UWIN invokes a process, it does not know whether the process is a UWIN process or a native process. It modifies the PATH variable so that it uses the ; separated DOS format. Open files are opened in the same manner as the Microsoft C library does, so programs that are compiled with this library should correctly inherit open files from UWIN programs. The initializations function also checks whether a security token has been placed in its address space by the UMS server, and if so, it impersonates this token.

The posix.dll has an initialization routine that sets up the file descriptors, assigns the controlling terminal, and starts the terminal emulation threads as required. The library also supplies a WinMain() function which is called when the program begins. This function initializes the stdin, stdout, and stderr file descriptors and then calls a posix.dll function that leads to invocation of main() funciton. The posix.dll function starts up the signal thread and sets the exception filter for signal processing.

Much of the process invocation is handled by the posix.dll so that the programs do not require recompilation when changes are added there.

Signal Management

Windows does not support interprocess exchange of software signals. There is no command in Windows analogous to the Unix kill() command. In addition, most Win32 kernel operations are not interruptible. UWIN provides Unix signal semantics on Windows, supporting both synchronous and asynchronous signal delivery, and interruption of Unix system calls.

Signals are handled by having each process run a thread that waits on an event. To send a signal to a process, the bit corresponding to the given signal number is set in the process block of the receiving process, and then its signal thread event is set. The signal thread then wakes up and looks for signals. It is important for the signal handler to be executed in the primary thread of the process, since the handler may contain a longjmp() out of the handler function. Prior to calling main(), an exception filter is added to the primary thread that checks for signals. The signal thread does this by suspending the primary thread raising an exception, which will activate the exception filter of the primary thread, and then resumes the primary thread. UWIN uses the structured exception

handling mechanism to implement signals and also handles signals generated within a process.

The Unix signals are mapped to Win32 exceptions. For e.g., a signal whose default action is exit will cause the process to exit with the Win32 exception number corresponding to the signal as its exit code. This makes the process compatible with native Win32 programs. UWIN converts Win32 exception codes, so that the UWIN processes calling wait() or waitpid() receive status in the form they expect.

UWIN also handles job control signals for interrupting, killing, suspending, and restarting a job.

Memory Management

Unix has a single process heap and a single process stack. The process stack is dynamic growing until it reaches a system-imposed limit. But in Windows, the stack grows to a fixed per process limit. The heap growth is not necessarily contiguous as it is in Unix. If the heap grows, the new address range may not be adjacent to the current heap block.

UWIN uses its own libraries for memory management. This ensures that it has greater control over the memory layout of a process so that it can be replicated exactly as and when it is required (like the fork() call). However, UWIN does not have control over the memory managed by the native Windows APIs. Hence, they will not get replicated for applications that use them.

Security

Ids and Permissions

Windows NT uses security identifiers to identify users and groups. A security identifier consists of an array of numbers that identify the administrative authority and sub-authorities associated with a given user. A UNIX user or group id is a single number that uniquely identifies a user or group only within a single system. Information about users is kept in the Registry database, which is accessible through the Win32 API and the LAN manager API. UWIN maintains a table of security identifier prefixes, and constructs the user id and group id by a combination of the index in this table and the last component of the security identifier. The number of security identifier prefixes that are likely to be encountered on a given

machine is much smaller than the number of accounts so that this table is easier to maintain.

Security for Files/Objects

Windows NT uses an access control list (ACL) for each file or object to control the access for each user. UNIX uses a set of permission bits associated with the three classes of users - the owner of the object, the group to which the object belongs, and others. While it is possible to construct an access control list that more or less corresponds to a given UNIX permission, it is not always possible to represent a given access control list with UNIX permissions.

Permissions for files are available only on NTFS file systems. It has separate permissions for writing a file, deleting a file, and for changing the permission of a file. These permissions are given to the owner of the file. The write bit on UNIX systems determines these three permissions. Thus, it is possible to encounter files that have partial write capability. The UNIX umask() command sets the default ACL so that native applications run by UWIN will create files with UNIX type permissions.

UNIX processes have real and effective user and group ids which control access to resources. Windows NT assigns each process a security token that defines its set of privileges. UNIX systems use setuid/setgid to delegate privileges to processes. Windows NT uses a technique called impersonation to carry out commands on behalf of a given user. There is no user who has unlimited privileges as the root user in UNIX. Instead the special privileges of root have been broken apart into separate privileges that can be given to one or more users.

Interprocess Communication

UWIN supports the standard System V IPCs – semaphores, message queues, and shared memory. The Unix pipes are also supported. All the open IPCs are deleted when the first UWIN process comes up (typically on a system reboot).

IPCs have been implemented using the Windows events, mutex and memory mapped files. The information regarding the IPCs opened by UWIN processes are stored in a memory mapped file under /usr/tmp/.ipc directory. Each file under this directory represents a single open IPC. The pipes have been implemented using the anonymous Win32 pipes.

Networking

Communication between processes running on different systems is possible through the implementation of FIFOs and sockets.

FIFO

FIFOs are implemented by using WIN32 named pipes. The first process that reads from and the first one that writes into the FIFO are the only processes that can create and connect to the named pipe. All other instances duplicate the handle of either the reader or the writer end. This way, all writers to a FIFO use the same handle as required by FIFO semantics.

Sockets

Both Internet domain protocol (i.e. AF_INET family) and UNIX domain sockets are implemented in UWIN. The multicast socket protocol is also supported. Winsock does not provide the AF_UNIX domain sockets.

The socket interface uses the UCB header files and naming conventions. Most of the calls except a few like socket(), select(), connect(), accept() are provided as a plain wrapper over the WINSOCK interface, the Microsoft API for BSD sockets. Unix domain sockets are provided using the window's named pipe facility.

DNS routines have been implemented to return the result of the query in a static memory, unlike the windows which returns these results as pointer to memory location. The static memory is overwritten with new information, when a domain DNS query routine is called next time.

No socket based terminal is available with UWIN.

Terminal Interface

The UWIN termios interface is implemented by creating two threads; one for processing keyboard input events, and the other for processing output events and escape sequences. These threads are connected to the read and write file descriptors of the process by pipes. The same architecture is used for serial I/O lines. Initially, these threads run in the process that created the console and make that process as the controlling terminal. These threads service all processes that share the controlling terminal. When a process is created, these threads are suspended and the console handles are passed down to the child. This enables a native application to run with its standard input and output as console handles. If the

application has been linked with the posix.dll, then these threads are resumed before main() is called so that UNIX style terminal processing takes place. The result is that Unix processes will echo characters as they are typed and respond to special keys specified by stty, whereas native WIN32 applications will only echo characters when they are read and will use Control-C as the interrupt character.

Error Mapping/logging

Errors returned by WIN32 functions are mapped onto UNIX errors by UWIN. The Unix APIs supported by UWIN return the same errors for these kinds of failures as on a Unix system. This ensures that Unix applications get proper information about the failures.

The Unix APIs are internally implemented by the posix.dll using the Win32 APIs and all the failures that occur in it are logged into /tmp/uwin_log file.

The failures that occur in UWIN Master Service and UWIN Client Service are logged in /tmp/ums.out and /tmp/ucs.out respectively. Other than failures, the logs also contain some information on the flow of control in the services.

Some information on the flow of control during installing UWIN and failures in it, if any, are logged into /tmp/install_log file.

Chapter 3

Programming Languages Supported on UWIN

The two programming languages supported by UWIN are C and C++. UWIN relies on compilers from Microsoft Visual C++ or from Borland C/C++. It provides cc for C & CC for C++ compilation, and ld wrapper that behave like their Unix counterparts. To look up messages and warnings, the *online Win32 SDK Tools Reference* can be used since the UWIN compilers are front-ends for the native Microsoft/Borland compiler.

Compiling C programs

UWIN provides the cc compiler that takes the same options as the compiler on any Unix platform. The UWIN compiler is a wrapper over the underlying native C compiler. To pass options directly to the native compiler/linker/preprocessor, the following options can be used.

- Use –Y option to pass additional compiler or linker flags with cc. Use "–Y c,<option>" for the compiler,
  "–Y l,<option>" for the linker and
  "–Y p,<option>" for the preprocessor
- Use the verbose flag -V to determine how cc invokes the native compiler and linker

Compiling C++ programs

Use the CC command to compile C++ applications, and not the cc command. The options for compiling C++ programs are the same as those for C programs. For linking applications that contain mixed language sources(C and C++), CC command must be used to compile and link the application.

Apart from the cc compiler that comes along with UWIN, the GNU gcc compiler can also be used. The source code for gcc compiler is the same as that used in Unix. However, this is optional and has to be installed separately if the developer needs it. It can be obtained from the website http://www.wipro.com/uwin

Chapter 4
Debugging Using Visual C++ IDE

Applications compiled on UWIN using the cc compiler can be debugged using the Microsoft Visual C++ IDE. However, when the GNU gcc compiler is used, the debugger that comes along with it must be used. The IDE cannot be used for debugging when compiled under gcc.

The application to be debugged must be compiled in debug mode so that break points can be put in the sources before running it. The source will not be loaded when it is not compiled under debug mode. Any application can be compiled in debug mode by using the g flag of cc.

Eg: cc -g -o test.exe test.c
will compile test.c in debug mode.

Debugging

The various ways of debugging an application are discussed in the following sections.

Debugging Standalone applications

A standalone application can be directly run with the debugger as below:

$ msdev app.exe <arguments>
where <arguments> is the command line parameters passed to the application when run normally.

The application will be loaded in the IDE and appropriate break points can be put in the source code(by opening the source file from "*File->open*") and the program is then executed. The arguments can also be provided after the debugger gets loaded by selecting "*Project->Settings->Debug->Program arguments"* from the main menu.

Debugging non-standalone applications

If the application is started from some other application, then the debugger can be attached to the application when it comes up as given below.

```
$ msdev -p 0x0<pid>
```

where <pid> is the process identifier of the application in hexadecimal.

There is another way to attach a debugger to a running process in Windows NT. From the task manager, a right mouse click on the application to be debugged brings up a pop-up menu. Selecting the *'Debug'* option from the menu attaches the debugger to the application. If a message pops up saying "The operation could not be completed. The system cannot find the file specified ", then the following registry key needs to be edited using any registry editor (like regedit/regedt32).

HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\AeDebug\Debugger

and a double quote needs to be added around the path where msdev.exe is present.

Eg: The original string might be (depends on where msdev is installed)

D:\Program Files\Microsoft VisualStudio\Common\MSDev98\Bin\msdev.exe -p %ld -e %ld

This has to be changed to

"D:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\msdev.exe" -p %ld -e %ld

Note that only double quotes have been added. The rest remains same. After making this change, close the task manager and reopen before trying to debug.

These methods are suitable only when the application stays long enough so that it can be attached to a debugger or it hangs during execution. The other way is to modify the application and put the Win32 API DebugBreak() at the point where the application needs to break and then recompile and run it. When the application executes the break instruction, it will pop-up a dialog box and selecting 'Cancel' will load it in the debugger. Note that when the Win32 API DebugBreak() is used, the windows header file "windows.h" needs to be included along with other header files.

Chapter 5
Building shared libraries

Dynamically Linked Libraries (DLL) in UWIN can be built using the ld or nld command. The nld command builds shared libraries that do not implicitly depend on the UWIN runtime. To build a DLL the following command can be used.

```
$ld –G –o name *.o –lxxx
```

where xxx is every library that the DLL depends on. The ld command creates the files name.lib and name.dll. The file name.dll is the dynamic library and name.lib is the interface file. This is needed to link to this library. There is no requirement that the static library be named as name.lib, but the cc and ld commands will look for name.lib and libname.a in that order when searching each library directory during compilation. Since the name for the DLL is embedded into the static library it should be named as name.dll.

The ld command allows .o's to be placed in the static library name.lib instead of name.dll by using the -Bstatic on the link line.

The .o's should be compiled with the _DLL flag set. Setting this flag allows only a few of the data symbols to be referenced by indirection from a DLL, and the _DLL flag changes the references to indirection.

There are two methods to export symbols from a DLL. The first method is to create a .def file. This file lists out symbols that are made visible by the library. The .def file can be included on the ld link line to build the export list.

The other is to use __declspec(export) lines in the code. This strategy does not work with some gcc versions. To make it less intrusive in the code, UWIN defines the variable __EXPORT__ to be __declspec(export) when the _DLL compiler switch is enabled. To build the name.dll, make the following changes to the header files that defines the interface:

- The name.dll is to be compiled with _BLD_name and _DLL defined.
- Insert the following lines in the interface headers before the first exported function definition.

```
#if defined(_BLD_name) && defined(__EXPORT__)
```

```
#define extern __EXPORT__
#endif
```

- The following line is to be inserted at the end of the function definitions.

  ```
  #undef extern
  ```

- Data symbols are more complex and should be avoided wherever possible. For data symbols the following lines are inserted before the declarations in addition to the one in the second step described above.

  ```
  #if !defined(_BLD_name) && defined (__IMPORT__)
  #define extern __IMPORT__
  #endif
  ```

  The __IMPORT__ symbol is defined by UWIN.

In this way, it is possible to keep a single platform-independent source code base. The <curses.h> header provides an example of a file that has been modified to produce a dynamically linked version. The AT&T nmake program that comes with the UWIN has a library rule that can be used to specify library construction in a platform independent manner. The rule can be used to build static or dynamic rules or both.

Chapter 6
Daemons and Services

Daemons on UNIX are referred to as services on Windows. These are processes, which run in the background. Windows NT provides a Service Control Manager, which controls the starting and stopping of services. Windows 9x has a simpler architecture, which provides a list of programs to be invoked automatically when the system boots up.

Differences

Windows treat services in a manner different from Unix. Some of the differences are listed below.

Windows NT

When a service is installed, it is registered in the Service Control Manager and it is removed from it when the service is deleted. Users can modify the service database with appropriate permissions using the applet or the Server Manager application, which allows manipulation of services on remote systems.

The Service Control Manager maintains a database of all the registered services in the system. It contains information such as:

- The executable that is providing the service; a single executable can provide multiple services on Windows NT.
- The startup option of the service(boot time or on demand or never)
- The severity level for service start failures
- The list of other services the service depends on, so that service startup can be ordered correctly
- User account information identifying the user context in which the service runs

Windows 95/98

Windows 9x does not have a service control manager. Instead a list of services to be run at boot up is maintained in the registry. Each executable is run at boot time but there is no way of controlling it. It will not appear in the task list that is displayed when you press Ctrl+Alt+Del.

UWIN Specific Services

The following services are specific to UWIN:

- UWIN Master Service (UMS)

- UWIN Client Service (UCS)

UWIN Master Service (UMS)

The UWIN Master Service starts up on system boot and keeps running in the background. The following command stops the service

```
net stop uwin_ms
```

from the command prompt. It can be restarted with the command

```
net start uwin_ms
```

It provides the following services to UWIN and its applications.

Creation of /etc/passwd and /etc/group files

The UMS is responsible for creation of /etc/passwd and /etc/group files. Every time the NT system boots up, UMS queries the user database and recreates these files with the NT User database information. The user information to be stored in these files is obtained from the NT User Database. If the NT system is part of a WorkGroup, only the Local user database is queried and the /etc/passwd and /etc/group files contain the local accounts only. If the NT system is part of an NT Domain, then the Domain database is also queried along with the Local database and the /etc/passwd and the /etc/group file contains both Local and Domain accounts. For a Domain Controller, only one set of Domain accounts is present in the /etc/passwd and /etc/group files. This Domain accounts enumeration feature can be controlled from the UWIN Control Panel applet. Any changes made to the /etc/passwd or /etc/group files will get over-written at the next system reboot. Hence, for creating user accounts, or modifying user information, the NT User Manager (musrmgr.exe) should be used. Any changes made to the user database through the NT User Manager get reflected in the /etc/passwd and /etc/group files only after the next reboot. /etc/mkpasswd can be used to regenerate the /etc/passwd and /etc/group files with the new changes incorporated.

Starting of daemons

On startup, UMS runs the script /etc/rc. This script is used to execute tasks that are scheduled to run at system startup. Since /etc/rc starts on bootup, any application that needs to be start on boot up can be invoked by this script.

This script invokes the inetd daemon. Apart from this, the other tasks accomplished by the rc script are :

- clean up utmp and utmpx files
- truncate the /tmp/uwin_log file
- mount any filesystem specified in /etc/fstab

- run sshd (if available)
- run crontab

Setuid functionality

A user belonging to the Administrators group can make a setuid() call. UMS provides the functionality through which setuid()calls can be made by other applications.

UWIN Client Service (UCS)

A UWIN Client Service (UCS) has to be installed for every user who wants to make a setuid() call.

The various ways of installing/uninstalling a UCS are :

- UCS can also be installed by running the following command:

  $ /etc/ucs install <username> <passwd>

  UCS can be uninstalled by running the following command:

  $ /etc/ucs delete <username>

  where <username> is the account for whom a UCS service needs to be installed/deleted, and <passwd> is the password of that user account.
- Using the UWIN applet from the Control Panel.
- Telneting to a user account in UWIN automatically installs a UCS service for that user.

If the password of a user account for which a UCS service is already installed needs to be changed, delete and reinstall the UCS service with the new password.

Daemons

UWIN provides telnet, ftp, rlogin and rsh daemons. These daemons are invoked by inetd, the Internet super-server. UMS runs /etc/rc in startup and this script starts the inetd daemon, which in turn invokes other daemons.

Connection-oriented services are invoked by the inetd each time a connection is made, by creating a process. Datagram oriented services are invoked when a datagram arrives; a process is created and passed a pending message on file descriptor 0.

Inetd uses a configuration file /etc/inetd.conf which is read at startup and possibly later in response to a HUP signal. The configuration file is ``free format'' with fields given in the order shown below. Continuation lines for an entry must begin with a space or tab. All fields must be present in each entry.

Service name	Must be in /etc/services or must name a tcpmux service

Socket type	Stream/dgram/raw/rdm/seqpacket
Protocol	Must be in /etc/protocols
Wait/nowait	Single-threaded/multi-threaded
Username	User in whose context daemon runs
Server program	Full path name
Server program arguments	Maximum of MAXARGS (20)

TCP services without official port numbers are handled with the RFC1078-based tcpmux internal service. tcpmux listens on port 1 for requests. When a connection is made from a foreign host, the service requested is passed to tcpmux, which looks up in the servtab list and returns the proper entry for the service. tcpmux returns a negative reply if the service doesn't exist, otherwise the invoked server is expected to return the positive reply if the service type in inetd.conf file has the prefix "tcpmux/". If the service type has the prefix "tcpmux/+", tcpmux will return the positive reply for the process; this is for compatibility with older server code, and also allows you to invoke programs that use stdin/stdout without putting any special server code in them. Services that use tcpmux are "nowait" because they do not have a well-known port and hence cannot listen for new requests.

Any user-defined service can also be invoked by the inetd daemon. An appropriate entry should be made in the inetd.conf file. Inetd has to be running all the time for the services to come up.

Telnetd

The /etc/in.telnetd daemon allows users to logon to the NT machine from the network. When a user logs into a NT/UWIN system using telnet, the default shell is as specified in /etc/passwd file. The UWIN telnetd requires users to login using their Windows NT domain and their login name. Login names are case-sensitive, and must be entered as they were entered into the User Manager.

If the NT system is part of NT Domain, a telnet user can either log into the Local System or the Domain. To log into the Local System, only the username (in the Local System) should be used. For logging in the Domain, the Username should be given as "Domain/Username" (where username is the name of the user account in the Domain).

Ftpd

This service allows users to transfer data over the network. The corresponding daemon is /etc/in.ftpd. For this daemon to function,

a UCS service has to be installed for the user in whose account, a ftp login is made.

Rlogind

This service allows users to login from a remote machine. The corresponding daemon is /etc/in.rlogind.

Rsh

This allows users to execute a command from a remote machine. The daemon is /etc/in.rshd.

Chapter 7
The Communication Port Interface

This chapter describes the UWIN serial communication port interface. As Windows does not support logging into the system through a serial port, UWIN supports com ports for data transfer only – a com port cannot be the controlling terminal of a process.

Com Port Devices

UWIN recognizes two forms of device names for com ports:
/dev/mod[0-7]
/dev/tty[0-9]

/dev/mod[0-7]
This is the modem control version of the Windows device COMn.
Where n is the com port number.

This device is used for making serial connections through modems. It has a major number 9 and minor numbers in the range 0-7. Eight /dev/mod devices are supported. However the actual number of ports that can be opened depend on number of physical ports available on the machine.

These devices cannot be shared - a device can be opened by only one process at a time. If a second process tries to access the device while it is being used by another process, the second process will block the open() call, until the first process frees the device. However, these devices can be shared by duplicating the file descriptor explicitly through dup() function call or through inheritance from parent to child as in case of fork().

Each device is associated with the termios structure, which controls the communication of the data over the serial lines. The APIs tcgetattr()and tcsetattr()are to be used for this purpose.

The communication parameters are used to control the speed, number of bits per character, parity check, flow control, enabling/disabling the receiver, and others. Setting the CREAD bit

of c_lflag in termios structure enables a device to send/receive data. The same can also be achieved with TIOCSDTR flag in ioctl command. Enabling the CLOCAL bit of c_flag allows the modem status to be monitored during transmission. The modem status will be ignored otherwise.

The different control lines of the serial line are

TIOCM_DTR	data terminal ready
TIOCM_RTS	request to send
TIOCM_CTS	clear to send
TIOCM_CAR	Carrier detect
TIOCM_CD	Same as TIOCM_CAR
TIOCM_RNG	Ring
TIOCM_RI	Same as TIOCM_RNG
TIOCM_DSR	Data set ready

The following ioctl() commands are used to check the status and change the values on the above control lines. All these commands require a pointer to an integer as an argument.

TIOCMSET
This will turn on or turn off the control lines whose corresponding bits are set in the integer passed as an argument.

TIOCMGET
This will fetch the status and set the argument to that value.

TIOCMBIS
The argument acts as a mask. The lines, whose bits are set in the mask, are turned on. No other lines are affected.

TIOCMBIC
The argument acts as a mask. The lines, whose bits are set in the mask, are turned off. No other lines are affected.

Apart from these commands, some others which do not take any argument, are:

- **TIOCSBRK** - set break bit.
- **TIOCCBRK** - clear break bit.
- **TIOCSDTR** - set data terminal ready.
- **TIOCCDTR** - clear data terminal ready.

The following steps are to be followed while using the device :

1. Open the device with the open() API. The device is opened in the non-blocking mode.
2. Adjust the communication parameters using the API.
3. Send or receive data using the read()/write() APIs. The select() API can be used with these devices for detecting the readiness of the device for sending/receiving data or the break condition. Any break condition occurring in transmission will be detected as an exception by select().
4. Close the device using the close() API.

/dev/tty[0-9]

This device is analogous to dumb terminals on Unix. This allows dumb ttys to be connected to a UWIN system. It has a major number 1 and minor number in the range 0-9. Depending on the com ports available, a maximum of 10 devices have been provided.

This device establishes a direct communication with remote terminal connected to the serial port. The remote terminal could be a Teletype or an application like Windows Hyperterm, Procom.

It assumes a hardwired serial connection and ignores the carrier detect serial line. To establish a connection, "tlogin.exe" should be running on the host, preferably in the background. tlogin.exe acts as the login process to access the windows system through the dumb terminal. This program opens the serial port(/dev/tty00 i.e COM1) and execs the ksh. When a connection is established with the serial port through a direct serial connection, a '$' prompt appears on the dumb terminal.

The remote side should have the following *Port Settings* for the serial line:

Parameter	Value
Bits per second	19200
Data Bits	8
Parity	None
Stop Bits	1
Flow Control	None

For tlogin these parameters are non-configurable. 'tlogin' does not work with modems. A separate application has to be written to provide remote access over dialup serial lines.

This device starts two threads that perform read and write operation. These threads take care of detecting the carrier, setting

up the line parameters. There are no ioctl commands to take care of the line status.

A complete termios interface is provided. The tcgetattr()/tcsetattr() APIs are used to get/set the termios settings.

Appendix 1
Unsupported Features

It would be nice if the entire UNIX could be implemented with WIN32, but this is not the case. Some of the unsupported features have been discussed below.

One problem, which is an artifact of using the WIN32 API rather than the POSIX subsystem, is that there is no way to create or access a file whose name ends in '.' Even using the FILE_FLAG_POSIX_SEMANTICS flag with CreateFile()does not help.

A second problem is that the way authentication works in Windows NT differs from that on UNIX systems. On UNIX systems, the password is encrypted. On Windows NT, a function that takes the user name and password is called, and this function returns a token that can be used to define the access privileges of a process. Since there is no access to the encrypted passwords in WIN32, programs such as ftp that require authentication or programs such as telnetd that need to create processes on behalf of a user had to be given extra support in the form of UWIN Client Service for them to work properly.

A third problem is how to fchmod() a file whose handle is opened with read permission only. If the handle was opened by another process, the file name is not known due to which opening the file by another process is not possible. One solution is to try opening the file always with the permission to change the mode. The problem with this solution is that the open will fail for any file that is not owned by the user who tries to open it, and a second open attempt is required. This would practically double the time needed for opening a read-only file, which is unacceptable.

There are some other problems related to concurrency restrictions that do not occur on UNIX systems.

- If a file is a memory-mapped file, an attempt to open it for truncation will fail. There is a way around this problem – the vi editor has to be changed not to use memory mapping.
- Changing the access permissions of a dynamically linked library that is in use is not permitted.

- Renaming a directory which is in use by any other process or whose subdirectory is the current working directory of another process is not possible.
- In Windows 9x, it is not possible to move or rename open files, even when they are opened for maximum sharing.

Finally, there is a problem with the permission system. Adding a new group to an existing process is not possible. It is often difficult or impossible to map access control lists to UNIX permissions in a meaningful way.

Appendix 2
UWIN Specific API

There are some extra APIs supported by UWIN. These functions provide a bridge between the UWIN's Unix environment and the WIN32 environment enabling the use of native WIN32 calls from UWIN applications. Including the header <uwin.h> causes the header <windows.h> also to be included. The following types are also defined in <uwin.h>

```
typedef HANDLE Handle_t;
typedef STARTUPINO  Startupinfo_t;
```

The UWIN specific APIs are discussed in the following sections.

uwin_handle()

In Unix, there is no concept of a HANDLE but only file descriptors. In UWIN, the handle to an object is represented as a file descriptor conforming to Unix. For a given file descriptor, this API returns the corresponding win32 handle associated with it. Since a UWIN file descriptor may have two handles associated with it, use UWIN_PRIMARY flag to specify the primary handle.

uwin_mktoken()

This function returns an access token given a username and a password. If there is no account for the given user, or if the password is not correct, uwin_mktoken() reports failure.

Any combination of the following flags can be specified.

UWIN_TOKCLOSE:

Close the token before returning. The return value will still be non-zero if the name and passwd arguments were valid.

UWIN_TOKUSE:

The created token will become the effective user and group of the current process.

uwin_ntpid()

On Windows NT, the UWIN process id will be the NT process id unless the process id has been created by exec. For a given UWIN pid, this API returns the NT pid.

uwin_path()

For a given UNIX pathname, the corresponding Win32 pathname is stored in a buffer supplied to the API and the length of that path is returned.

uwin_spawn()

The uwin_spawn() function provides a UNIX style interface to the WIN32 CreateProcess() function. In addition to the argument list argv and environment list env, uwin_spawn() allows you to specify data as a pointer to the spawndata structure which contains following elements:

Element	Description
Handle_t tok	Create with this token
Unsigned long flags	Create process flags
Startupinfo_t start	Structure containing starup info
Id_t grp	The process group id
Int trace	Trace file descriptor

uwin_unpath()

The uwin_unpath() function converts a Win32 pathname into an equivalent UWIN pathname. The UWIN path name is returned into a buffer that is supplied as an argument to the API and the length of the UWIN path is returned by the function.