UNIX Custom Application Migration Guide

Volume 1: Plan

Published: May 2006

Contents

About This Volume

Introduction

Volume 1, *Plan* of the *UNIX Custom Application Migration Guide* describes the features of the Microsoft® Windows® operating system and the operational differences between the UNIX and Windows environments. It also outlines the activities carried out in the Microsoft Solutions Framework (MSF) Envisioning and Planning Phases for a UNIX to Windows application migration. This volume includes the following topics:

- Introduction to UNIX to Windows migration.
- Envisioning Phase: Beginning your migration project.
- Planning Phase: Planning your migration project.
- Planning Phase: Setting up the development and test environments.

Intended Audience

This volume is designed for senior IT decision makers and managers, network managers, project managers, and operating system administrators. The following list describes the specific ways each audience can use the guide.

- **IT decision makers and managers**. High-level IT decision makers and managers can find information on the feasibility of migrating applications from UNIX to Windows. They can also find guidelines on assessing the UNIX application and choosing the appropriate path and strategy for migration.
- **Network managers**. Network managers can find help with assessing the current infrastructure in their UNIX environment and identify the high-level infrastructure requirements for migration.
- **Project managers**. With the help of this volume, project managers can assess the feasibility of migration, identify major areas that will be affected by the migration, and choose the appropriate environment and strategy for conducting the migration. The guidance discusses common issues involved in migrating applications from UNIX to Windows and provides resolutions for them. Project managers can use this volume to plan the development, budget, project plans, and delivery of the migration.
- **Operations managers**. Operations managers can use this guide to obtain information on the interoperability issues that must be addressed when deploying an application to the new environment. The guide also provides information on integration, deployment, support, and maintenance of the application.
- **Developers**. Developers can use this volume to learn about the various UNIX to Windows migration options and choose the best strategy to fit their environment and application types. This volume also helps in identifying and validating the technology and will also help in writing the functional specification documents.
- **Testers**. Testers can use this volume to identify the tools and integrated development environments (IDEs and to configure the test environments.

Knowledge Prerequisites

The readers of this volume should possess the following knowledge prerequisites:

- Basic knowledge of UNIX and Windows.
- Knowledge of software engineering methodologies and process models.
- Knowledge of project management.

Layout of the Guide: Volume 1

The following diagram depicts the layout of the guide and how the volumes of the guide correlate with the components of the MSF Process Model. The shaded portion indicates to the reader the position of the current volume in the layout of the entire guide.

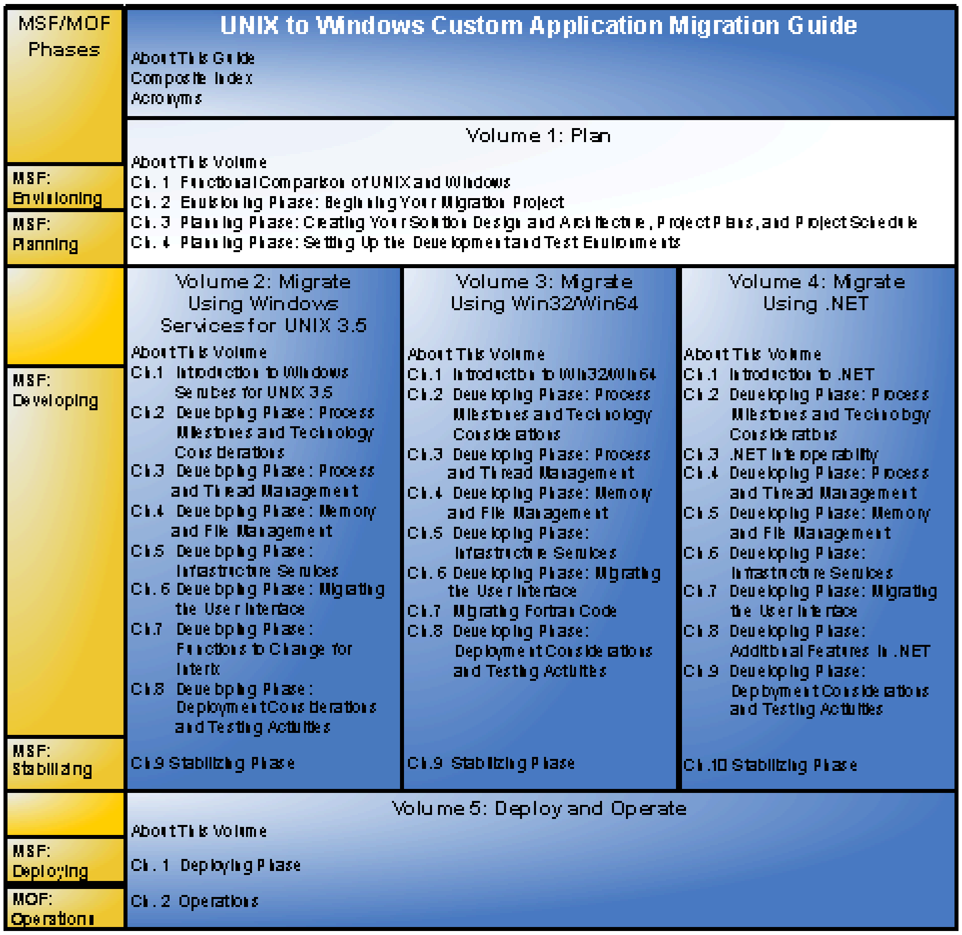

MSF/MOF Phases
UNIX to Windows Custom Application Migration Guide
About This Guide
Composite Index
Acronyms
MSF: Envisioning
MSF: Planning
Volume 1: Plan
About This Volume
Ch. 1 Functional Comparison of UNIX and Windows
Ch. 2 Envisioning Phase: Beginning Your Migration Project
Ch. 3 Planning Phase: Creating Your Solution Design and Architecture, Project Plans, and Project Schedule
Ch. 4 Planning Phase: Setting Up the Development and Test Environments
MSF: Developing
Volume 2: Migrate Using Windows Services for UNIX 3.5
About This Volume
Ch.1 Introduction to Windows Services for UNIX 3.5
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 Developing Phase: Process and Thread Management
Ch.4 Developing Phase: Memory and File Management
Ch.5 Developing Phase: Infrastructure Services
Ch. 6 Developing Phase: Migrating the User Interface
Ch.7 Developing Phase: Functions to Change for Interix
Ch.8 Developing Phase: Deployment Considerations and Testing Activities
Ch.9 Stabilizing Phase
Volume 3: Migrate Using Win32/Win64
About This Volume
Ch.1 Introduction to Win32/Win64
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 Developing Phase: Process and Thread Management
Ch.4 Developing Phase: Memory and File Management
Ch.5 Developing Phase: Infrastructure Services
Ch. 6 Developing Phase: Migrating the User Interface
Ch.7 Migrating Fortran Code
Ch.8 Developing Phase: Deployment Considerations and Testing Activities
Ch.9 Stabilizing Phase
Volume 4: Migrate Using .NET
About This Volume
Ch.1 Introduction to .NET
Ch.2 Developing Phase: Process Milestones and Technology Considerations
Ch.3 .NET Interoperability
Ch.4 Developing Phase: Process and Thread Management
Ch.5 Developing Phase: Memory and File Management
Ch.6 Developing Phase: Infrastructure Services
Ch.7 Developing Phase: Migrating the User Interface
Ch.8 Developing Phase: Additional Features in .NET
Ch.9 Developing Phase: Deployment Considerations and Testing Activities
Ch.10 Stabilizing Phase
MSF: Stabilizing
MSF: Deploying
MOF: Operations
Volume 5: Deploy and Operate
About This Volume
Ch. 1 Deploying Phase
Ch. 2 Operations

Figure 0.1. UCAMG organization

Organization of Content by Chapter

This volume consists of the following chapters:

- **About This Volume**. This chapter provides an overview of the organization, layout, and content of the volume. It describes the intended audience and the knowledge prerequisites required for this volume and provides resources for using this volume.
- **Chapter 1: Introduction to UNIX to Windows Migration**. This chapter provides the technical background behind the evolution of UNIX and Windows, followed by a brief introduction to Microsoft Windows Server™ 2003. This chapter also lists the business and technical reasons for using each of the migration alternatives in the Windows environment. It also provides examples of application architectures that exist on the UNIX and Windows platforms.
- **Chapter 2: Envisioning Phase: Beginning Your Migration Project**. This chapter provides information on the various activities carried out in the Microsoft Solutions Framework (MSF) Envisioning Phase and its major tasks and deliverables. The primary purpose of this chapter is to assess the applications in the UNIX environment in order to reduce the gap between the UNIX application and proposed Windows application. This chapter also provides information on defining the business goals, setting up the project team, defining high-level requirements, developing the solution concept, defining the solution scope and project scope, and performing the risk assessment.
- **Chapter 3: Planning Phase: Planning Your Migration Project**. This chapter explains the process of defining the solution design and architecture along with developing the functional specifications. It lists the inputs required to validate the technology and create the proof of concept. The chapter also provides guidance on creating the project plan and schedule.
- **Chapter 4: Planning Phase: Setting Up the Development and Test Environments**. This chapter discusses the process of establishing a development environment, which includes setting up the hardware and other infrastructure resources, as well as any project structure or policy elements required to develop the solution. This chapter also describes the process of creating a test environment according to the test plan and choosing an appropriate build process.

Resources

An acronyms list and job aids are additional resources provided with the guide that will be helpful for the planning work covered in this volume.

Job Aids

Job aids are included in a Tools and Templates folder that is available as part of the download version of this guide at http://go.microsoft.com/fwlink/?LinkId=30864 . The following job aids are relevant to Volume 1:

- Project Team Skills Job Aid (Microsoft Word template)
- Risk Assessment Tool (Excel tool)
- Migration Approach Assessment Tool (Excel tool)
- UCAMG Scripts Tool (Excel tool)

Acronyms

An Acronyms list identifies the acronyms used in the guide and shows their meanings.

Document Conventions

Table 0.1 provides details of the document conventions used in this guide.

Table 0.1. Document Conventions

Text Element	Meaning
Bold text	Used in the context of paragraphs for file names, commands, literal arguments to commands (including paths when they form part of the command), switches, and programming elements such as methods, functions, data types, and data structures. Also used to identify the UI elements.
Italic text	Used in the context of paragraphs for variables to be replaced by the user. Also used to emphasize important information.
Monospace font	Used for excerpts from configuration files, code examples, and terminal sessions.
Monospace bold font	Used to represent commands or other text that the user types.
Monospace italic font	Used to represent variables that the reader supplies in command-line examples and terminal sessions.
`Shell prompts and Code Snippets`	Used to represent shell scripts and code snippets.
Note	Represents a note.
`Code`	Represents code.

Chapter 1: Functional Comparison of UNIX and Windows

Organizations today are constantly focusing on improving the availability, manageability, performance, and serviceability of their services and products, while at the same time reducing the cost and complexity involved in providing them. As the IT decision maker, these requirements create a difficult situation for you. You can either choose to make no changes and risk your systems becoming obsolete, or you can make a change and risk joining a computing trend that turns out to be an evolutionary dead end.

Mainframe computers have been used by the industry for a long time. They often use any one of the various versions of UNIX provided by vendors. The various implementations of the UNIX operating system have served the industry well, as witnessed by the large base of installed systems and the number of large-scale applications installed on these systems. However, there are increasing signs of dissatisfaction with expensive, often proprietary solutions running on UNIX architecture. In many cases, the user experience of these systems involves text-based interaction with dumb terminals or a terminal-emulation session on a computer. Dissatisfied with these characteristics of UNIX, IT managers have often sought alternatives to the increasing expense of being tied to single-vendor software and hardware solutions. In addition, since the mid-1980s, corporate data centers have been moving away from mainframes, with their dedicated operating systems, to minicomputers.

This has prompted more organizations to evaluate migrating to Intel-based architecture as an option. One of the most extraordinary and unexpected successes of the Intel PC architecture is the flexibility to extend its framework to include very large server and data center environments. Large-scale hosting organizations are now offering enterprise-level services to multiple client organizations at an availability level of more than 99.99 percent on what are just racks of relatively inexpensive computers.

These catalysts have led more organizations to migrate their enterprise-level applications from UNIX to Windows. The key benefits (for example, reduction of costs and increased flexibility) of the Microsoft® Windows® operating system has fueled this interest, increasing the number of organizations that are considering migrating to alternative platforms.

Migration, or coexistence, with the Microsoft Windows operating system makes sense because it offers greater enterprise-readiness and a better-integrated future roadmap for application development and server support, along with quantifiable reductions in total cost of ownership (TCO).

Before you migrate from UNIX to Windows, it is recommended that you learn about the evolution and architecture of the two operating systems. The following section provides this information.

Evolution and Architecture

This section provides an overview of the development and production environments in both Windows and UNIX. Before you begin to plan your UNIX-to-Windows migration, however, it is important that you understand the Windows and UNIX operating systems, their terminologies, and the key differences between them.

Windows

This section outlines the evolution of the Windows family of operating systems. The section also discusses the architecture of Microsoft Windows Server™ 2003 and its salient features.

Evolution of the Windows Operating System

In the late 1980s, Microsoft began designing a new operating system that could take advantage of the advancements in processor design and software development. The new operating system was called Windows NT® (for new technology). (The Windows Server 2003 and Windows XP operating systems are based on Windows NT.)

Figure 1.1 illustrates the evolution of the Windows family of operating systems, culminating in Windows XP and Windows Server 2003. Windows XP is built on the Windows NT kernel. It incorporates many of the best features of all the earlier Windows platforms, including Plug and Play support, an intuitive user interface (UI), and many innovative support services. Windows Server 2003 is essentially Windows XP with added server features and enhancements to various services, such as IIS Web Server. In addition, Windows Server 2003 includes a 64-bit edition for computing.

Note Windows Server 2003 and Windows XP are the most recent operating systems from the Windows family and are the preferred operating systems when planning a migration from UNIX to Windows. This guide assumes that you are migrating to one of these two operating systems.

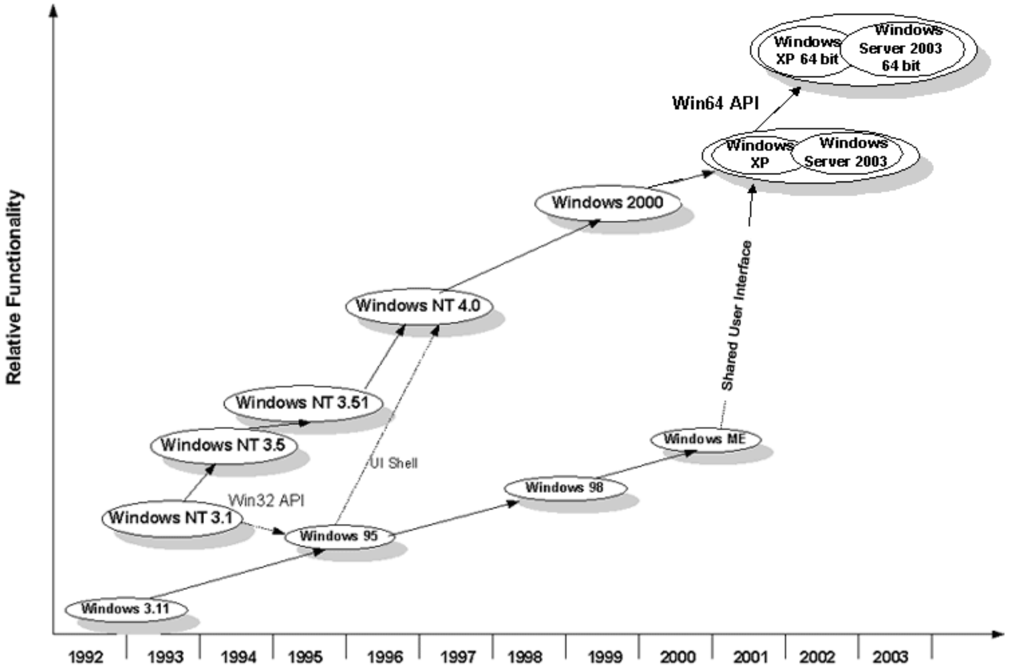


Figure 1.1. The evolution of the Windows family of operating systems

Windows Server 2003 Architecture

Windows Server 2003 architecture uses two processor access modes: the user mode and the kernel mode.

The user mode includes application processes, which are typically Windows programs and a set of protected subsystems. These subsystems are referred to as "protected" because each of these subsystems is a separate process with its own protected virtual address space. Of these, the most important subsystem is the Microsoft Win32® subsystem, which supplies much of the Windows functionality. The Windows application programming interface (API) is useful for developing both 32-bit and 64-bit Windows-based applications.

Another important subsystem, particularly with respect to migration of UNIX applications, is the Portable Operating System Interface (POSIX) for computing environments. This is a set of international standards for implementing UNIX-like interfaces. The POSIX subsystem implements these standards-based interfaces and allows application developers to easily port their applications to Windows from another operating system. The POSIX subsystem is not implemented on Windows Server 2003 but comes as a part of the Interix (Windows Services for UNIX 3.5) installation on Windows.

The kernel mode is a highly privileged mode of operation in which the program code has direct access to the virtual memory. This includes the address spaces of all user mode processes and applications and their hardware. The kernel mode is also known as the supervisor mode, protected mode, or Ring 0. The kernel mode of Windows Server 2003 contains the Windows NT executive as well as the system kernel. The Windows NT executive exports generic services that protected subsystems call to obtain basic operating system services, such as file operations, input/output (I/O), and synchronization services. Partitioning of the protected subsystems and the system kernel simplifies the base operating system design and makes it possible to extend the features of an individual protected subsystem without affecting the kernel. The kernel controls how the operating system uses the processors. Its operations include scheduling, multiprocessor synchronization, and providing objects that the executive can use or export to applications.

The Windows operating system supports the following features and capabilities:

- Multitasking.
- Flexibility to choose a programming interface (user and kernel APIs).
- A graphical user interface (GUI) and a command-line interface for users and administrators. (The default UI is graphical.)
- Built-in networking. (Transmission Control Protocol/Internet Protocol [TCP/IP] is standard.)
- Persistent system service processes called "Windows Services" and managed by the Windows Service Control Manager (SCM).
- Single compatible implementation irrespective of the vendor from whom it is purchased.

Figure 1.2 shows a high-level illustration of the Windows Server 2003 architecture.

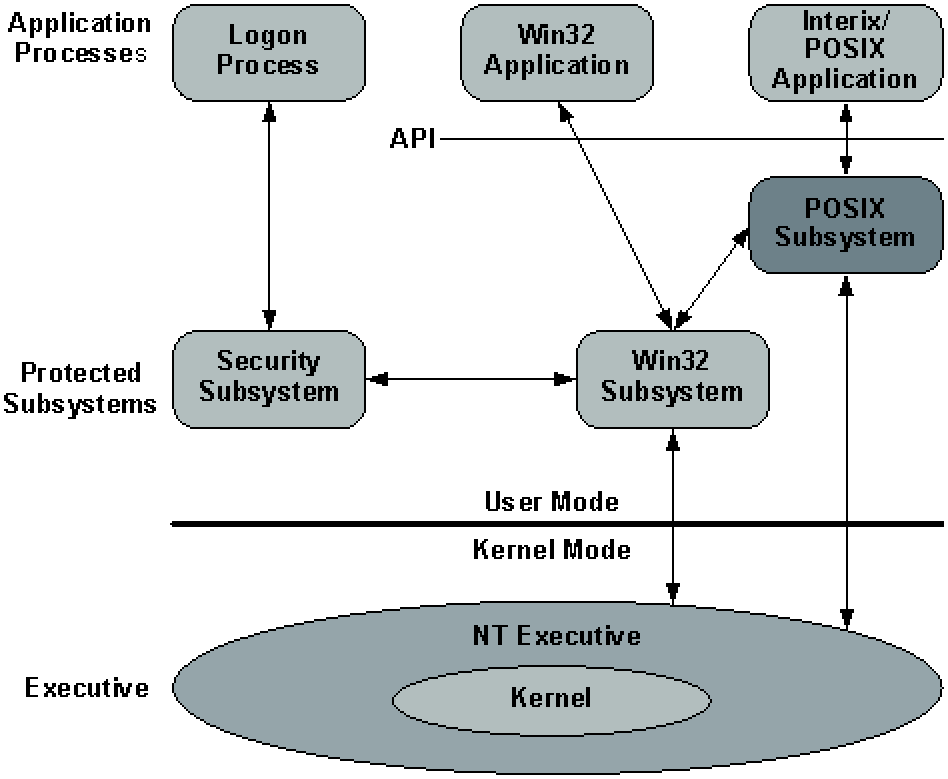


Figure 1.2. Windows Server 2003 architecture

UNIX

This section outlines the evolution of the UNIX operating system. The architecture of UNIX and its salient features are also discussed.

Evolution of the UNIX Operating System

In 1969, Bell Laboratories developed UNIX as a "timesharing" system, a term used to describe a multitasking operating system that supports multiple users at each terminal. Although the first implementation was written in assembly language, the designers always intended to write UNIX in a higher-level language. Therefore, Bell Labs invented the C language so that they could rewrite UNIX. UNIX has evolved into a popular operating system that runs on computers ranging in size from personal computers to mainframes.

Figure 1.3 depicts the evolution of UNIX from a single code base into the wide variety of UNIX systems available today. In fact, this is only a summary—there are more than 50 flavors of UNIX in use today. The codes in the diagram refer to the brands and versions of UNIX that are in common use, including:

- AIX from IBM.
- Solaris from SUN Microsystems.
- HP-UX and Tru64 from Hewlett Packard.

- UnixWare from Caldera.
- FreeBSD, which is an open source product.

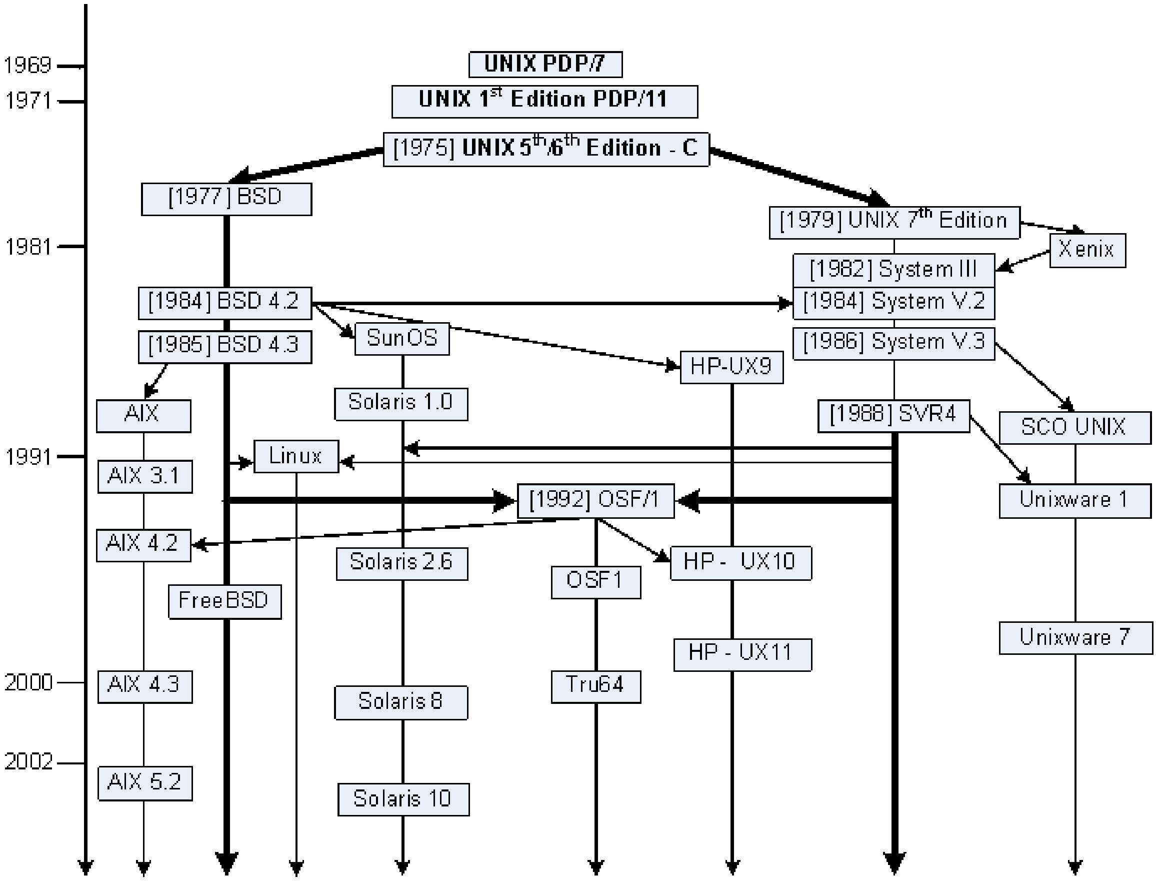


Figure 1.3. The evolution of the UNIX operating system

UNIX Architecture

The architecture of UNIX can be divided into three levels of functionality, as shown in Figure 1.4.

1. The lowest level is the kernel, which schedules tasks, manages resources, and controls security.
2. The next level is the shell, which acts as the UI, interpreting user commands and starting applications.

   Note For more information about the shell, refer to Chapter 2, "Developing Phase: Process Milestones and Technology Considerations" of Volume 3: *Migrate Using Win32/Win64* of this guide.

3. The tools are at the highest level. These provide utility functions such as **ls**, **vi**, and **cat**.

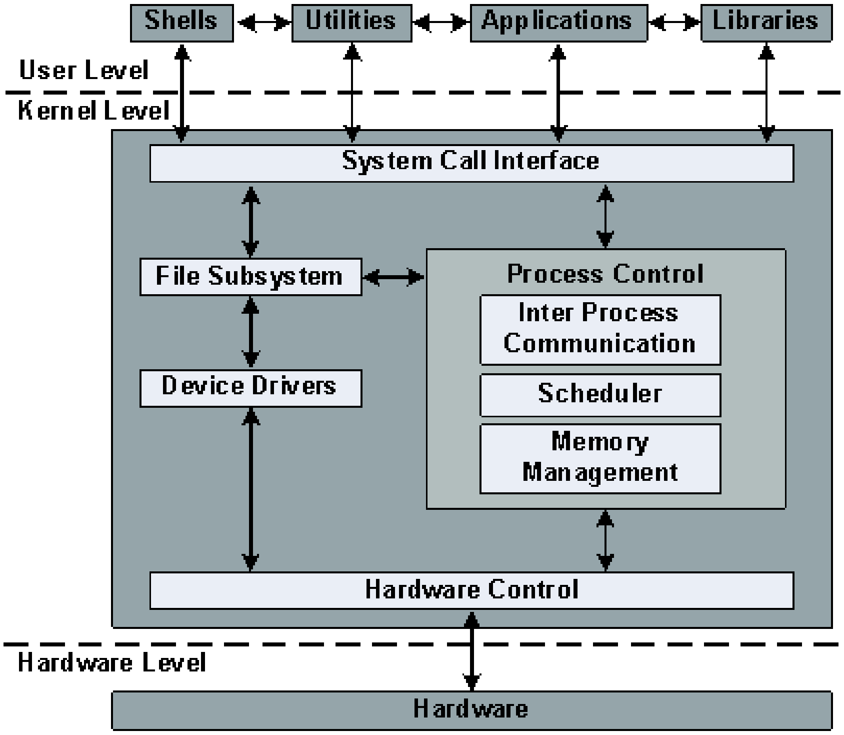


Figure 1.4. UNIX architecture

The UNIX operating system supports the following features and capabilities:

- Multitasking and multiuser.
- Kernel written in high-level language.
- Programming interface.
- Use of files as abstractions of devices and other objects.
- Character-based default UI.
- Built-in networking. (TCP/IP is standard.)
- Persistent system service processes called "daemons" and managed by **init** or **inetd**.

Comparison of Windows and UNIX Environments

This section compares the Windows and UNIX architectures, emphasizing the areas that directly affect software development in a migration project.

Kernels and APIs

As with most operating systems, Windows and UNIX both have kernels. The kernel provides the base functionality of the operating system. The major functionality of the kernel includes process management, memory management, thread management, scheduling, I/O management, and power management.

In UNIX, the API functions are called system calls. System calls are a programming interface common to all implementations of UNIX. The kernel is a set of functions that are used by processes through system calls.

Windows has an API for programming calls to the executive. In addition to this, each subsystem provides a higher-level API. This approach allows Windows operating systems to provide different APIs, some of which mimic the APIs provided by the kernels of other operating systems. The standard subsystem APIs include the Windows API (the Windows native API) and the POSIX API (the standards-based UNIX API).

Windows Subsystems

A subsystem is a portion of the Windows operating system that provides a service to application programs through a callable API.

Subsystems come in two varieties, depending upon the application program that finally handles the request:

- **Environment subsystems**. These subsystems run in a user mode and provide functions through a published API. The Windows API subsystem provides an API for operating system services, GUI capabilities, and functions to control all user input and output. The Win32 subsystem and POSIX subsystem are part of the environment subsystems and are described as follows:
    - **Win32 subsystem**. The Win32 environment subsystem allows applications to benefit from the complete power of the Windows family of operating systems. The Win32 subsystem has a vast collection of functions, including those required for advanced operating systems, such as security, synchronization, virtual memory management, and threads. You can use Windows APIs to write 32-bit and 64-bit applications that run on all versions of Windows.
    - **POSIX subsystem and Windows Services for UNIX**. To provide more comprehensive support for UNIX programs, Windows uses the Interix subsystem. Interix is a multiuser UNIX environment for a Windows-based computer. Interix conforms to the POSIX.1 and POSIX.2 standards. It provides all features of a traditional UNIX operating system, including pipes, hard links, symbolic links, UNIX networking, and UNIX graphical support through the X Windows system. It also includes case-sensitive file names, job control tools, compilation tools, and more than 300 UNIX commands and tools, such as KornShell, C Shell, **awk**, and **vi**.

      Because it is layered on top of the Windows kernel, the Interix subsystem is not an emulation. Instead, it is a native environment subsystem that integrates with the Windows kernel, just as the Win32 subsystem does. Shell scripts and other scripted applications that use UNIX and POSIX.2 tools run under Interix.
- **Integral subsystems**. These subsystems perform key operating system functions and run as a part of the executive or kernel. Examples are the user-mode subsystems, Local Security Authority subsystem (LSASS), and Remote Procedure Call subsystem (RPCSS).

Kernel Objects and Handles

Kernel objects are used to manage and manipulate resources—such as files, synchronization objects, and pipes—across the processes. These kernel objects are owned by the kernel and not by the process. Handles are the opaque numbers or the data type used to represent the objects and to uniquely identify the objects. To interact with an object, you must obtain a handle to the object.

In UNIX, the kernel object can be created using the system calls and it returns an unsigned integer. There is no exact equivalent of handles in UNIX.

In Windows, Windows APIs are used to create the kernel object and it returns a Windows-specific data type called HANDLE.

Hardware Drivers

Hardware drivers are system software used to interact the hardware devices with the operating system.

In UNIX, there are several different ways to manage drivers. Some UNIX implementations allow dynamic loading and unloading of drivers, whereas other implementations do not. The UNIX vendor usually provides drivers. The range of Intel hardware supported for UNIX is typically smaller than that for Windows.

In Windows, the Windows driver model provides a platform for developing drivers for industry-standard hardware devices attached to a Windows-based system. The key to developing a good driver package is to provide reliable setup and installation procedures and interactive GUI tools for configuring devices after the installation. In addition, the hardware must be compatible with Windows Plug and Play technology in order to provide a user-friendly hardware installation.

Process Management

A process is usually defined as the instance of the running program. Process management describes how the operating systems manage the multiple processes running at a particular instance of time. Multitasking operating systems such as Windows and UNIX must manage and control many processes simultaneously. Both Windows and UNIX operating systems support processes and threads.

The following sections provide more details on process management in both UNIX and Windows:

- **Multitasking**. UNIX is a multiprocessing, multiuser system. At any given point, you can have many processes running on UNIX. Consequently, UNIX is very efficient at creating processes.

  Windows has evolved greatly from its predecessors, such as Digital Equipment Corporation’s VAX/VMS. It is now a preemptive multitasking operating system. As a result, Windows relies more heavily on threads than processes. (A thread is a construct that enables parallel processing within a single process.) Creating a new process is a relatively expensive operation while creating a new thread is not as expensive in terms of system resources like memory and time. Hence, multiprocess-oriented applications on UNIX typically translate to multithreaded applications on the Windows platform, thus saving such system resources as memory and time.

- **Multiple users**. One key difference between UNIX and Windows is in the creation of multiple user accounts on a single computer.

  When you log on to a computer running UNIX, a shell process is started to service your commands. The UNIX operating system keeps track of the users and their processes and prevents processes from interfering with one another. The operating system does not come with any default interface for user interaction. However, the shell process on the computer running UNIX can connect to other computers to load third-party UIs.

  When a user logs on interactively to a computer running Windows, the Win32 subsystem’s Graphical Identification and Authentication dynamic-link library (GINA) creates the initial process for that user, which is known as the user desktop, where all user interaction or activity takes place. The desktop on the user's computer is loaded from the server. Only the user who is logged on has access to the desktop. Other users are not allowed to log on to that computer at the same time. However, if a user employs Terminal Services or Citrix, Windows can operate in a server-centric mode just as UNIX does.

The Windows operating system supports multiple users simultaneously through the command line and a GUI. The latter requires the use of Windows Terminal Services. The UNIX operating system supports multiple simultaneous users through the command line and through a GUI. The latter requires the use of X Windows. Windows comes with a default command shell (cmd.exe); UNIX typically includes several shells and encourages each user to choose a shell as the user's default shell. Both operating systems provide complete isolation between simultaneous users. There are some key differences between the two: Windows comes with a "single user" version that allows one user at a time (Windows XP) as well as a multiuser server version (Windows Server). It is rare for a Windows Server system to have multiple simultaneous command-line users.

- **Multithreading.** Most new UNIX kernels are multithreaded to take advantage of symmetric multiprocessing (SMP) computers. Initially, UNIX did not expose threads to programmers. However, POSIX has user-programmable threads. There is a POSIX standard for threads (called Pthreads) that all current versions of UNIX support.

  In Windows, creating a new thread is very efficient. Windows applications are capable of using threads to take advantage of SMP computers and to maintain interactive capabilities when some threads take a long time to execute.

- **Process hierarchy**. When a UNIX-based application creates a new process, the new process becomes a child of the process that created it. This process hierarchy is often important, and there are system calls for manipulating child processes.

  Unlike UNIX, Windows processes do not share a hierarchical relationship. The creating process receives the process handle and ID of the process it created, so a hierarchical relationship can be maintained or simulated if required by the application. However, the operating system treats all processes like they belong to the same generation. Windows provides a feature called Job Objects, which allows disparate processes to be grouped together and adhere to one set of rules.

Signals, Exceptions, and Events

In both operating systems, events are signaled by software interrupts. A signal is a notification mechanism used to notify a process that some event has taken place or to handle the interrupt information from the operating system. An event is used to communicate between the processes. Exceptions occur when a program executes abnormally because of conditions outside the program's control.

In UNIX, signals are used for typical events, simple interprocess communication (IPC), and abnormal conditions such as floating point exceptions.

Windows has the following mechanisms:

- Windows supports some UNIX signals and others can be implemented using Windows API and messages.
- An event mechanism that handles expected events, such as communication between two processes.
- An exception mechanism that handles nonstandard events, such as the termination of a process by the user, page faults, and other execution violations.

Daemons and Services

A daemon is a process that detaches itself from the terminal and runs disconnected in the background, waiting for requests and responding to them. A service is a special type of application that is available on Windows and runs in the background with special privileges.

In UNIX, a daemon is a process that the system starts to provide a service to other applications. Typically, the daemon does not interact with users. UNIX daemons are started at boot time from init or rc scripts. To modify such a script, it needs to be opened in a text editor and the values of the variables in the script need to be physically changed. On UNIX, a daemon runs with an appropriate user name for the service that it provides or as a root user.

A Windows service is the equivalent of a UNIX daemon. It is a process that provides one or more facilities to client processes. Typically, a service is a long-running, Windows-based application that does not interact with users and, consequently, does not include a UI. Services may start when the system restarts and then continue running across logon sessions. Windows has a registry that stores the values of the variables used in the services. Control Panel provides a UI that allows users to set the variables with the valid values in the registry. The security context of that user determines the capabilities of the service. Most services run as either Local Service or Network Service. The latter is required if the service needs to access network resources and must run as a domain user with enough privileges to perform the required tasks.

Virtual Memory Management

Virtual memory is a method of extending the available physical memory or RAM on a computer. In a virtual memory system, the operating system creates a pagefile, or swapfile, and divides memory into units called pages. Virtual memory management implements virtual addresses and each application is capable of referencing a physical chunk of memory, at a specific virtual address, throughout the life of the application.

Both UNIX and Windows use virtual memory to extend the memory available to an application beyond the actual physical memory installed on the computer. For both operating systems, on 32-bit architecture, each process gets a private 2 GB of virtual address space. This is called user address space or process address space. The operating system uses 2 GB of virtual address space, called system address space or operating system memory. On 64-bit architecture, each process gets 8 terabytes of user address space.

File Systems and Networked File Systems

This section describes the file system characteristics of UNIX and Windows. Both UNIX and Windows support many different types of file system implementations. Some UNIX implementations support Windows file system types, and there are products that provide Windows support for some UNIX file system types.

File system characteristics and interoperability of file names between UNIX and Windows are discussed as follows:

- **File names and path names**. Everything in the file system is either a file or a directory. UNIX and Windows file systems are both hierarchical, and both operating systems support long file names of up to 255 characters. Almost any character is valid in a file name, except the following:
    - In UNIX: ***/***
    - In Windows: **?, ", /, \, >, <, *, |,** and **:**

  UNIX file names are case sensitive while Windows file names are not.

  In UNIX, a single directory known as the root is at the top of the hierarchy. You locate all files by specifying a path from the root. UNIX makes no distinction between files on a local hard drive partition, CD-ROM, floppy disk, or network file system (NFS). All files appear in one tree under the same root.

  The Windows file system can have many hierarchies, for example, one for each partition and one for each network drive. A UNIX system provides a single file system tree, with a single root, to the applications it hosts. Mounted volumes (whether local devices or network shares) are "spliced" into that tree at locations determined by the system administrator. The Windows operating system exposes a forest of file system trees, each with its own root, to the applications it hosts. Mounted volumes appear as separate trees ("drive letters") as determined by the administrator or user. Both UNIX and Windows provide a tree view of all network-accessible shares. UNIX provides an administrator-defined view of these shares through an automounter mechanism, while Windows provides a full view through the Universal Naming Convention (UNC) pathname syntax.

- **Server message block (SMB) and Common Internet File System (CIFS)**. One of the earliest implementations of network resource sharing for the Microsoft MS-DOS® platform was network basic input/output system (NetBIOS). The features of NetBIOS allow it to accept disk I/O requests and direct them to file shares on other computers. The protocol used for this was named server message block (SMB). Later, additions were made to SMB to apply it to the Internet, and the protocol is now known as Common Internet File System (CIFS).

  UNIX supports this through a software option called Samba. Samba is an open-source, freeware, server-side implementation of a UNIX CIFS server.

  In Windows, the server shares a directory, and the client then connects to the Universal Naming Convention (UNC) to connect to the shared directory. Each network drive usually appears with its own drive letter, such as X.

- **Windows and UNIX NFS interoperability**. UNIX and Windows can interoperate using NFS on Windows or CIFS on UNIX. There are a number of commercial NFS products for Windows. For UNIX systems, Samba is an alternative to installing NFS client software on Windows-based computers for interoperability with UNIX-based computers. It also implements NetBIOS-style name resolution and browsing. Microsoft offers a freely downloadable NFS Server, Client, and Gateway as part of Windows Services for UNIX 3.5 installation. Windows Services for UNIX also provide a number of services for interoperability between Windows-based and UNIX-based computers.

Security

This section describes some of the security implementation details and differences between UNIX and Windows:

- **User authentication**. A user can log on to a computer running UNIX by entering a valid user name and password. A UNIX user can be local to the computer or known on a Network Information System (NIS) domain (a group of cooperating computers). In most cases, the NIS database contains little more than the user name, password, and group.

  A user can log on to a computer running Windows by entering a valid user name and password. A Windows user can be local to the computer, can be known on a Windows domain, or be known in the Microsoft Active Directory® directory service. The Windows domain contains only a user name, the associated password, and some user groups. Active Directory contains the same information as the Windows domain and may contain the contact information of the user, organizational data, and certificates.

- **UNIX versus Windows security**. UNIX uses a simple security model. The operating system applies security by assigning permissions to files. This model works because UNIX uses files to represent devices, memory, and even processes. When a user logs on to the system with a user name and a password, UNIX starts a shell with the UID and GID of that user. From then on, the permissions assigned to the UID and GID, or the process, control all access to files and other resources. Most UNIX vendors can provide Kerberos support, which raises their sophistication to about that of Windows.

  Windows uses a unified security model that protects all objects from unauthorized access. The system maintains security information for:

  - **Users.** System users are people who log on to the system, either interactively by entering a set of credentials (typically user name and password) or remotely through the network. Each user's security context is represented by a logon session.
  - **Objects**. These are the secured resources (for example, files, computers, synchronization objects, and applications) that a user can access.
- **Active Directory**. Windows Server 2003 uses the Active Directory directory service to store information about objects. These objects include users, computers, printers, and every domain on one or more wide area networks (WANs). Active Directory can scale from a single computer to many large computer networks. It provides a store for all the domain security policy and account information.

Networking

Networking basically provides the communication between two or more computers. It defines various sets of protocols, configures the domains, IP addresses, and ports, and communicates with the external devices like telephones or modems and data transfer methods. It also provides the standard set of API calls to allow applications to access the networking features.

The primary networking protocol for UNIX and Windows is TCP/IP. The standard programming API for TCP/IP is called sockets. The Windows implementation of sockets is known as Winsock (formally known as Windows Sockets). Winsock conforms well to the Berkeley implementation, even at the API level. The key difference between UNIX sockets and Winsock exists in asynchronous network event notification. There is also a difference in the remote procedure calls (RPC) implementation in UNIX and Windows.

User Interfaces

The user interface (UI) provides a flexible way of communicating between the users, applications, and the computer.

The UNIX UI was originally based on a character-oriented command line, whereas the Windows UI was based on GUI. UNIX originated when graphic terminals were not available. However, the current versions of UNIX support the graphical user interface using the X Windows system. Motif is the most common windowing system, library, and user-interface style built on X Windows. This allows the building of graphical user interface applications on UNIX.

The Windows user interface was designed to take advantage of advancements in the graphics capabilities of computers. It can be used by all applications—including both client side and server side—and can also be used for tasks such as service administration. Windows contains the Graphics Device Interface (GDI) engine to support the graphical user interface.

System Configuration

UNIX users generally configure a system by editing the configuration files with any of the available text editors. The advantage of this mechanism is that the user does not need to learn how to use a large set of configuration tools, but must only be familiar with an editor and possibly a scripting language. The disadvantage is that the information in the files comes in various formats; hence the user must learn the various formats in order to change the settings. UNIX users often employ scripts to reduce the possibility of repetition and error. In addition, they can also use NIS to centralize the management of many standard configuration files. Although different versions of UNIX have GUI management tools, such tools are usually specific to each version of UNIX.

Windows has GUI tools for configuring the system. The advantage of these tools is that they can offer capabilities depending on what is being configured. In recent years, Microsoft Management Console (MMC) has provided a common tool and UI for creating configuration tools. Windows provides a scripting interface for most configuration needs through the Windows Script Host (WSH). Windows provides WMI (Windows Management Instrumentation), which can be used from scripts. Windows also includes extensive command-line tools for controlling system configuration. In Windows Server 2003, anything that can be done to manage a system through a GUI can be done through a command-line tool as well.

Interprocess Communication

An operating system designed for multitasking or multiprocessing must provide mechanisms for communicating and sharing data between applications. These mechanisms are called interprocess communication (IPC).

UNIX has several IPC mechanisms that have different characteristics and which are appropriate for different situations. Shared memory, pipes, and message queues are all suitable for processes running on a single computer. Shared memory and message queues are suitable for communicating among unrelated processes. Pipes are usually chosen for communicating with a child process through standard input and output. For communications across the network, sockets are usually the chosen technique.

Windows also has many IPC mechanisms. Like UNIX, Windows has shared memory, pipes, and events (equivalent to signals). These are appropriate for processes that are local to a computer. In addition to these mechanisms, Windows supports clipboard/Dynamic Data Exchange (DDE), and Component Object Model (COM). Winsock and Microsoft Message Queuing are good choices for cross-network tasks. Other cross-network IPC mechanisms for Windows include remote procedure calls (RPCs) and mail slots. RPC has several specifications and implementations, developed by third-party vendors, which support client server applications in distributed environments. The most prevalent RPC specifications are Open Network Computing (ONC) by Sun Microsystems and Distributed Computing Environment (DCE) by Open Software Foundation. UNIX systems support interoperability with Windows RPC. UNIX does not implement mailslots.

Synchronization

In a multithreaded environment, threads may need to share data between them and perform various actions. These operations require a mechanism to synchronize the activity of the threads. These synchronization techniques are used to avoid race conditions and to wait for signals when resources are available.

UNIX environments use several techniques in the Pthreads implementation to achieve synchronization. They are:

- Mutexes
- Condition variables
- Semaphores
- Interlocked exchange

Similarly, the synchronization techniques available in the Windows environment are:

- Spinlocks
- Events
- Critical sections
- Semaphores
- Mutexes
- Interlocked exchange

DLLs and Shared Libraries

Windows and UNIX both have a facility that allows the application developer to put common functionality in a separate code module. This feature is called a shared library in UNIX and a dynamic-link library (DLL) in Windows. Both allow application developers to link together object files from different compilations and to specify which symbols will be exported from the library for use by external programs. The result is the capability to reuse code across applications. The Windows operating system and most Windows programs use many DLLs.

Windows DLLs do not need to be compiled to position-independent code (PIC), while UNIX shared objects must be compiled to PIC. However, the exact UNIX behavior can be emulated in Windows by pre-mapping different DLLs at different fixed addresses.

Component-based Development

Component-based development is an extension to the conventional software development where the software is assembled by integrating several components. The components themselves may be written in different technologies and languages, but each has a unique identity, and each of them exposes common interfaces so that they can interact with other components.

UNIX supports CORBA as the main component-based development tool. However, it is not a standard component of the UNIX system; you have to obtain a CORBA implementation from another source.

On the other hand, the Windows environment offers a wide range of component-based development tools and technologies. This includes:

- COM
- COM+
- Distributed COM (DCOM)
- .NET components

Middleware

This section compares the various middleware solutions available for UNIX-based and Windows-based applications. Middleware technologies are mostly used to connect the presentation layer with the back-end business layers or data sources. One of the most prominent middleware technologies used in applications is a message queuing system.

Message queuing is provided as a feature in AT&T System V UNIX and can be achieved through sockets in Berkeley UNIX versions. These types of memory queues are most often used for IPC and do not meet the requirements for persistent store and forward messaging.

To meet these requirements, versions of the IBM MQSeries and the BEA Systems MessageQ (formally the DEC MessageQ) are available for UNIX. Microsoft provides similar functionality with Message Queuing for Windows.

Note More information about Windows messages and message queues is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/messagesandmessagequeues.asp.

IBM and BEA Systems also provide versions of their queuing systems for Windows.

Transaction Processing Monitors

A transaction processing monitor (TP monitor) is a subsystem that groups sets of related database updates and submits them to a database as a transaction. These transactions, often monitored and implemented by the TP monitors, are known as online transaction processing (OLTP). OLTP is a group of programs that allow real-time updating, recording, and retrieval of data to and from a networked system. OLTP systems have been implemented in the UNIX environments for many years. These systems perform such functions as resource management, threading, and distributed transaction management.

Although OLTP was originally developed for UNIX, many OLTP systems have Windows versions. Windows also ships with its own transaction manager. In addition, gateways exist to integrate systems that use different transaction monitors.

Shells and Scripting

A shell is a command-line interpreter that accepts typed commands from a user and executes the resulting request. Every shell has its own set of programming features known as scripting languages. Programs written through the programming features of a shell are called shell scripts. As with shell scripts, scripting languages are interpreted.

Windows and UNIX support a number of shells and scripting languages, some of which are common to both operating systems. Examples are: Rexx, Python, and Tcl/Tk.

Command-Line Shells

A number of standard shells are provided in the UNIX environment as part of the standard installation. They are:

- Bourne shell (sh)
- C shell (csh)
- Korn shell (ksh)
- Bourne Again Shell (bash)

On the Windows platform, Cmd.exe is the command prompt or the shell.

Windows versions of the Korn shell and the C shell are delivered with the Windows Services for UNIX 3.5, MKS, and Cygwin products.

Scripting Languages

In UNIX, there are three main scripting languages that correspond to the three main shells: the Bourne shell, the C shell, and the Korn shell. Although all the scripting languages are developed from a common core, they have certain shell-specific features to make them slightly different from each other. These scripting languages mainly use a group of UNIX commands for execution without the need for prior compilation. Some of the external scripting languages that are also supported on the UNIX environment are Perl, Rexx, and Python.

On the Windows environment, WSH is a language-independent environment for running scripts and is compatible with the standard command shell. It is often used to automate administrative tasks and logon scripts. WSH provides objects and services for scripts, establishes security, and invokes the appropriate script engine, depending on the script language. Objects and services supplied allow the script to perform such tasks as displaying messages on the screen, creating objects, accessing network resources, and modifying environment variables and registry keys. WSH natively supports VBScript and JScript. Other languages that are available for this environment are Perl, Rexx, and Python. WSH is built-in to all current versions of Windows and can be downloaded or upgraded from Microsoft.

Development Environments

The generic UNIX development environment uses a set of command-line tools. In addition, there are many third- party integrated development environments (IDEs) available for UNIX. Most of the character-based and visual IDEs provide the necessary tools, libraries, and headers needed for application development

The Windows development environment can be of two types: a standard Windows development environment or a UNIX-like development environment such as Windows Services for UNIX.

The standard Windows development environment uses the Microsoft Platform Software Development Kit (SDK) and Microsoft Visual Studio® .NET. The platform SDK delivers documentation for developing Windows-based applications, libraries, headers, and definitions needed by language compilers. It also includes a rich set of command-line and stand-alone tools for building, debugging, and testing applications. It is available at no cost from the MSDN Web site.

The Microsoft Visual Studio .NET environment delivers a complete set of tools for application development, including the development of multitier components, user interface design, and database programming and design. It also provides several language tools, editing tools, debugging tools, performance analysis tools, and application installation tools.

The development environment of Windows Services for UNIX contains documentation, tools, API libraries, and headers needed by language compilers. It also comes with a UNIX development environment, with tools such as GNU gcc, g++, g77 compilers, and a gdb debugger, which makes compilation of UNIX applications possible on the Windows environment.

Application Architectures

The following sections introduce and discuss different application architectures and how these applications are implemented on UNIX and Windows platforms.

Distributed Applications

Distributed applications are logically partitioned into multiple tiers: the view or the presentation tier, the business tier, and the data storage and access tier. A simple distributed application model consists of a client that communicates with the middle tier, which consists of the application server and an application containing the business logic. The application, in turn, communicates with a database that stores and supplies the data.

The most commonly used databases in UNIX applications are Oracle and DB2. You can either run the application’s current database (for example, Oracle) on Windows (that is, migrate the existing database from UNIX to Windows) or migrate the database to Microsoft SQL Server™. In some cases, the best migration decision is a conversion to SQL Server.

Another option available is Oracle. It offers a range of features available to both Windows Server 2003 and UNIX. You may choose to use these features with the current Oracle database. By separating the platform migration from the database migration, you have greater flexibility in migrating database applications.

Next is the presentation tier, which provides either a thick or a thin client interface to the application. UNIX applications may use XMotif to provide a thick client interface. In Windows, a thick client can be developed using the Win32 API, GDI+. The .NET Framework provides Windows Forms that help in rapid development of thick clients. Either one of these two methods can be used while migrating the thick client from UNIX to Windows.

Workstation Applications

Workstation-based applications run at the UNIX workstation (desktop) computer and access data that resides on network file shares or database servers. Workstation-based applications have the following architectural characteristics:

- They can be single-process applications or multiple-process applications.
- They use character-based or GUI-based (for example, X Windows or OpenGL) UIs.
- They access a file server (through NFS) or a database server for data resources.
- They access a computer server for computer-intensive services (for example, finite element models for structural analysis).

The Windows environment supports a similar workstation application using client/server technology.

Web Applications

Web applications from a UNIX Web server are normally one of the following types:

- **Common Gateway Interface (CGI)**. CGI is a standard for connecting external applications with information servers, such as Hypertext Transfer Protocol (HTTP) or Web servers. CGI has long been out of favor because of performance problems.
- **Java Server Page (JSP)**. Java Server Page (JSP) technology also allows for the development of dynamic Web pages. JSP technology uses Extensible Markup Language (XML)-like tags and scriptlets written in the Java programming language to encapsulate the logic that generates the content for the page. The application logic can reside in server-based resources (for example, the JavaBean component architecture) that the page accesses by using these tags and scriptlets. All HTML or XML formatting tags are passed directly back to the response page.
- **HTML**. Hypertext Markup Language is the authoring language used to create documents on the World Wide Web.

- **PHP**. PHP is a widely used, general-purpose scripting language that is especially suited for Web development and can be embedded into HTML.
- **JavaScript**. JavaScript is an extension of HTML. JavaScript is a script language (a system of programming codes) created by Netscape that can be embedded into the HTML of a Web page to add functionality.

Web applications on UNIX can be used on Windows with minor configuration changes on the Web server. You can also migrate Java-based Web applications on UNIX to Microsoft Web technologies on Windows.

Graphics-Intensive Applications

Graphics-intensive applications may support additional UI standards, such as OpenGL. OpenGL has become a widely used and supported two-dimensional and three-dimensional graphics API. OpenGL fosters innovation and provides for faster application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can take advantage of the capabilities of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

OpenGL runs on every major operating system, including Macintosh OS, OS/2, UNIX, Windows Server 2003, Windows XP, Linux, OPENStep, and BeOS. OpenGL also works with every major windowing system, including Win32, Macintosh OS, Presentation Manager, and X-Windows. OpenGL includes a full complement of graphics functions. The OpenGL standard has language bindings for C, C++, Fortran, Ada, and Java; therefore, applications that use OpenGL functions are typically portable across a wide array of platforms.

System-Level Programs or Device Drivers

The existence of an appropriate device driver on the Windows platform is often a gating factor for migrating applications that make use of nonstandard device drivers. Typically, customers do not have access to device-driver source code and are therefore unable to migrate a UNIX device driver to Windows. The topic of migrating device drivers is highly specialized and is not covered in this guide. The Windows Driver Development Kit (DDK) can be used to develop drivers on the Windows environment.

Chapter 2: Envisioning Phase: Beginning Your Migration Project

This chapter provides the background and technical information required to complete the first phase of a migration project, the Envisioning Phase.

Introduction to the Envisioning Phase

The Envisioning Phase represents an early form of planning. The purpose of this phase is to get the project started by achieving a basic agreement between the business and IT on the goals of the project, as well as its constraints. This chapter provides guidance on ways to break down the various decisions and analyses that must be made into manageable steps. It also offers guidance on practical tasks such as organizing the team. The decision-making activities are designed to minimize project risk by helping to ensure that decisions are aligned with broader organizational goals and are based on a careful assessment of the current environment and input from all key stakeholders.

This guide makes the following basic assumptions about the state of the organization at the beginning of the Envisioning Phase:

- The IT manager has decided to investigate the project and has sufficient budgetary authority to fund the investigation (Envisioning and Planning Phases). However, the IT manager may or may not have sufficient budgetary authority to approve the entire project.
- Rough estimates indicate that the project's benefits will exceed its costs, but detailed return on investment (ROI) calculations cannot be made until the project scope is defined.

Envisioning Phase Tasks

The major migration tasks conducted during the Envisioning Phase are summarized in the following list. They will be described in more detail in the following sections.

1. **Set up the team**. The Program Manager assembles a core project team, representing all six MSF Team Model roles.
2. **Define the project structure**. The Program Manager creates the project structure, which describes the administrative structure for the project team going into the Planning Phase. It also establishes the standards the team will use to manage and support the project.
3. **Define the business goals**. The team identifies the business problems or opportunities in order to determine the objectives for the solution.
4. **Assess the current situation**. The team assesses the current state of the business and performs a gap analysis to help identify a path toward the desired state of the business.

5. **Create the vision and define the scope for the project**. This includes creating a vision statement to guide the team toward its business goals and identifying a project scope to define what will and will not be included in the project.
6. **Define high-level requirements**. This includes determining the needs of each key stakeholder, sponsor, and end user in order to provide input regarding the formation of the solution concept, to provide criteria for evaluating the vision/scope document, and to provide a basis for detailed requirements in the Planning Phase.
7. **Develop the solution concept**. This includes creating an initial concept of the migration approach that will best meet business goals and technical requirements. This serves as a baseline and sets the stage for the more formal design of the solution.
8. **Assess risk**. The team identifies and qualifies any high-level risks that might affect the project.
9. **Close the Envisioning Phase**. The team agrees that they have achieved the Vision/Scope Approved Milestone by formally signing off on the documentation that records the results of the Envisioning Phase tasks.

Note Although listed sequentially, many of these activities can be performed concurrently.

Envisioning Phase Deliverables

The Envisioning Phase activities culminate in a major milestone, the Vision/Scope Approved Milestone. By the end of the Envisioning Phase, the project team and all major stakeholders (other members of the organization who will be affected by the project) should have agreed upon the following conceptual areas and documented their understanding in the appropriate documents. This includes:

- A project structure document that:
  - Describes and maps all MSF team roles.
  - Defines a project structure and hierarchy for the team to follow.
  - Defines the process standards for the team to follow.
- A vision statement that:
  - Defines the business opportunities or problems that the solution is expected to address, including major business and user requirements.
  - Describes the current state of the customer's environment at a high level, with respect to the technology under consideration.
  - Describes the desired future state of the organization's environment from an unbounded viewpoint.
- A scope definition that:
  - Defines the projects to be executed as part of the solution, based on business implications and technologies involved.
  - Lists the new features and functions that the project provides: the in-scope and the out-of-scope items.
  - Distinguishes between the scope of the solution and the scope of the project in cases where the solution will be implemented in stages or will require discrete projects to be run in parallel.
  - Provides a rough estimate of the project time frame and duration.

- A solution concept definition that:
  - Identifies the appropriate migration method and technology for the business problem like Microsoft Windows® Services for UNIX, Win32®/Win64, or .NET.
  - Describes at a high level how the solution will solve the business problem in terms of the approaches that will be followed to build and deliver the solution.
  - Documents business goals, design goals, and the required functionality of the solution.
  - Calls out assumptions and constraints that apply to the solution.
  - Defines the success criteria for the project.
- A risk assessment document that:
  - Assesses and identifies project risks.
  - Attempts to qualify and prioritize these risks.

Together, these deliverables (documents) comprise a high-level description of the project that forms the basis for further planning. They represent a baseline and may need to be revised as the planning progresses and new information dictates. The vision/scope document, especially, must be viewed as a living document that keep changing, subject to change control.

Note For more detailed discussions on how these tasks and deliverables can be approached and the responsibilities assigned for them, refer to the *Unix Migration Project Guide* (UMPG) at http://www.microsoft.com/technet/itsolutions/cits/interopmigration/unix/umpg/default.mspx.

Envisioning Phase Activities In-Depth

The following sections describe in depth the various activities involved in the Envisioning Phase of the MSF Process Model and how these activities specifically relate to a migration project.

Setting Up a Team

You need to assemble a core team first, based on the six roles defined in the MSF Process Model and with defined skill sets appropriate to the project. Once assembled, the team will be involved in defining the vision and scope that together provide a clear direction for the project and set the expectations for the project within the organization. Although the team as a whole is responsible for the project's success, each of the roles focuses on a distinct quality goal to ensure that responsibility is taken for implementing different aspects of the project.

Each role may be filled by one or several persons, depending on the size of the project. Some roles, such as Development and Test, are likely to be filled by entire subteams. In this case, the core team member acts as the lead for the subteam, and the guide frequently refers to the role as a team (for example, the Development team).

See the "Project Team Skills" job aid", a Word template that helps with assembling the project team by identifying the particular skills that are needed in each role (or subteam) for a UNIX application migration project.

The six roles are:

- **Product Management Role.** The Product Management Role acts as the interface between the customer and the project team by ensuring that the team addresses the customer's requirements and concerns.

- **Program Management Role.** The Program Management Role is responsible for leading the team in making decisions and driving the project schedule and project plan. Program Management owns responsibility for all internal team communication and ensuring adherence to corporate and project standards.
- **Development Role.** The Development Role is responsible for designing, building, and unit testing the baseline code. Development drives the architecture and physical design of the solution, estimates the effort required, and then builds the solution.
- **Test Role.** The Test Role is responsible for testing the components, architecture, and associated documentation against the functional specifications and ensuring that they meet requirements.
- **User Experience Role.** The User Experience Role advocates for the end user and defines the requirements for functionality from the end-user perspective. They are also involved in other critical activities such as development of help and support documentation and training.
- **Release Management Role.** The Release Management Role participates in the design process to ensure that the completed solution is deployable, manageable, and supportable. This role interacts with the product support, help desk, and other delivery channels with a focus on smooth deployment and ongoing management.

Note Depending on your business goals, some of these roles may not be necessary in the context of your organization. To understand these roles and their responsibilities, refer to the Project Team Skills Job Aid which is provided along with this guide.

Defining the Project Structure

The project structure defines how the team manages and supports the project and describes the administrative structure for the project team going into the subsequent project phases. The main function of the project structure is to define standards that the team will use during the project. Program Management takes the lead in defining the project structure. The major standards that are defined as part of this process are:

- **Defining project communications.** A communication standard should be defined as part of the project structure for the team members to follow. Among these standards can be a definition of the hierarchy under which team members operate, procedures to escalate project issues, regularly mandated status meetings, and any other project-specific communication standard that needs to be defined at the beginning of the project.
- **Defining documentation standards.** The project structure also defines standards on how to create training materials and documentation related to the project, including standards that prescribe the templates and applications the team should use to create documents.
- **Defining change control.** The project structure should include change control standards. These standards may apply to two distinct areas: the solution content and the documents that describe the content, such as the functional specification. For solution content, change control standards can include prescribed daily builds and the use of source control and versioning software.

Interim Milestone: Core Team Organized

This milestone marks the point at which the key team members have been assigned to specific roles in the team and the team has been organized.

Defining Business Goals

Business goals are established either to take advantage of an opportunity or to solve a problem in the current business. The problem/opportunities statement for the project leads to the creation of a team vision for the project. It also clearly articulates the objectives for the project and ensures that the solution supports the business requirements.

For migration projects, it would be helpful to consider the goals satisfied by the existing application or infrastructure as a way of identifying new or extended goals. However, justification for the migration project should exceed the justification for the initial work. Common business goals for migrating existing UNIX applications include:

- Reducing the total cost of ownership (TCO).
- Creating an environment that integrates all organizational processes.
- Increasing the scalability, manageability, performance, and flexibility of all organizational applications.
- Reducing the development time for the applications.
- Expanding the existing application capabilities.
- Replacing the existing UNIX applications.
- Internationalizing or localizing the agility of products.
- Extending the availability of the application to special devices (for example, Tablet PCs).

Assessing the Current Situation

After setting up a team to migrate your applications and defining the business goals, the next step is to assess your current situation. At this stage, you must measure the gap between where you are today and where your application will be after your vision is realized. This step is very critical because it reveals the effort that each migration approach will require, which can ultimately determine the direction your migration project will take.

This guide provides two tools that help you decide upon a migration approach by using information about your current situation and goals. The primary tool, the Migration Approach Assessment Tool, is a Microsoft Excel®-based tool that lists certain assessment questions, which are divided into three levels based on the business, architecture, and technology requirements of the current situation. Answer the relevant questions that are applicable to your environment with appropriate answers. At the end of the assessment, the tool will evaluate the requirements and their weights and will provide the suggested migration approach that is most suitable for your migration.

A supporting tool, the UNIX Application Source Code API Analysis Tool, also written in Excel, allows you to identify various UNIX APIs used in the source code of a UNIX application. The tool reports the level of use of each API as a percentage of the source code. For example, you might discover that SNMP APIs are used in 20 percent of your code. This is important because it will help you identify the effort that the migration process might require based on the APIs used in your application.

These two tools are both available in the Tools and Templates folder in the download version of this guide.

The following sources in your organization can help you answer these questions:

- Source code of the application

- Application documentation
- Stakeholders and business partners
- Current infrastructure
- Development team
- User base

Creating the Vision and Defining the Scope

Delineating the project vision and identifying the scope are distinct activities: Vision is an unbounded view of what a solution may be, whereas solution scope identifies the part or parts of the vision to be provided in a version of the solution. These are critical for migration projects because migration projects involve replacing an existing system, and the fastest path to success is achieved by tightly restricting the scope of change to minimize the addition of features beyond those delivered solely through the migration itself.

Creating the Project Vision

Before you proceed, it is important to define a vision for the solution that addresses your business goals. The vision describes the desired future state of your application environment. It may be crafted by a project team or be handed down by the business and negotiated as needed, and it is documented in the vision/scope document. The vision that guided the development of the existing implementation of an application or infrastructure component should serve as a major source in developing the vision statement for a migration project. A vision statement is always from an unbounded viewpoint, that is, an ideal world, what one strives to achieve given all necessary resources. Examples of vision statements are:

- We will increase the market share of our product by making it available on Windows.
- All the applications in the organization should run on both UNIX and Windows platforms.

Defining the Solution Scope

The solution scope places a boundary around the migration process by defining the range of features and functions mapping to your business goals that can be achieved. You can do this by defining what is out of scope and cannot be achieved, and by discussing the criteria by which the end users will accept the migration. Define the solution scope as part of your migration; it will be the basis for defining the project scope and for performing many types of project and operations planning.

Defining the solution scope may involve some of the following activities:

- Determining whether the entire application will be migrated or only some specific modules of the application.
- Identifying the third-party libraries or tools to be used during migration.
- Identifying the modules that will need to be recompiled during migration.
- Identifying the modules that need to be rewritten or redesigned during migration.
- Defining the acceptance criteria for performance of the migrated application on Windows.
- Determining the business goals that can be achieved, along with those that cannot be achieved.

Defining the Project Scope

As part of defining the solution scope for migration, you must define the different projects that need to be undertaken. The project scope defines what work the team will perform on the project—and what work it will not perform. The project scope applies the constraints imposed on the project (by resources, time, and other limiting factors) to the overall solution vision expressed in the vision statement. The solution scope and project scope are interdependent activities.

When defining your project scope you need to:

- Explain and clarify the planned intentions and deliverables of the project so that you can manage the expectations of the project stakeholders.
- Ensure that the resources and schedules for the projects are well defined and bounded.
- Confirm objectives and timescales for the migration so that you can manage expectations, recruit appropriate resources, and allocate tasks as necessary.

It is quite possible that multiple projects are identified—for example, one for back-end databases and one for specific user-facing applications. In this case, the project scope enables each project to be defined and any dependencies to be identified so that the migrations can take place as independently as possible.

The end of the Envisioning Phase is marked by the completion of the first milestone where the migration team agrees on the overall vision and scope of the project. The vision is the complete view of what the solution can be, based on refining the original idea. On the other hand, the scope establishes restrictions on the vision to establish what can be accomplished within the constraints of the project.

Note The project scope is the definition of success. It tells you when you are done. If you do not reach a true "meeting of the minds" between the stakeholders and project staff, significant time or cost overruns, or even project failure, will occur. For more information, see the UMPG.

Interim Milestone: Vision/Scope Document Baselined

At this stage of your migration, you ought to have outlined your vision/scope document, which also includes business goals, your high-level requirements, and the Windows technology to which you will be migrating. You should have analyzed and decided whether your application will be migrated in parts or in entirety. This decision needs to be made after repeated discussions with your team and feedback from the application customers and stakeholders has been considered.

The vision/scope document represents the result of the team's work in defining the vision for the solution and project scope. The process of working on this document as a team ensures that everyone on the team shares and understands the goals of the project. This helps each team member to work effectively toward the successful completion of the project. You are now ready to start defining the high-level requirements for your migration project.

Defining High–Level Requirements

After creating the vision statement, your core team must define a set of high-level requirements and features that map to achieving the business goals driving your vision. It is recommended that you adopt the use of versioned releases while defining the high-level requirements. Versioned releases help the project team respond to ongoing changes in scope, schedule, and project risk. In a migration project, the first release might be to migrate core functionality. Subsequent releases would migrate additional functionality or extend the migrated component to meet new customer needs.

The high-level requirements can fall into any of the following categories:

- Business or functional requirements
- Operating environment requirements
- Performance requirements
- Standards requirements

For example, if your vision is to improve the performance efficiency of the existing UNIX applications on Windows, then some of the high-level requirements that need to be defined are:

- What is the performance efficiency (in percent) you want to achieve?
- Should this performance efficiency be constant at all times?
- What is the maximum deviation from this efficiency that you can tolerate?

You need to perform the following activities as part of your requirements definition:

- Documenting requirements for the migrated application.
- Preparing the acceptance criteria.
- Reviewing the requirements and acceptance criteria.
- Obtaining a sign-off on the requirements from application users, such as customers and stake holders.

Another critical factor to be considered when defining high-level requirements is the environmental profile. Environment includes everything that is outside the application but affects the application in one way or another. It is important to consider these external factors too, which either impose restrictions on or anticipate certain behavior from the application.

Environmental Profile

These external factors should be captured as high-level requirements to be tested in the migrated application. Examples include:

- Software systems that interact with the migrated application and impose such constraints as:
    - Time limitations (for example, batch jobs or data transfer time).
    - Communication requirements (for example, protocol support).
    - Performance metrics (processing per time unit).
- Privacy policies (for example, affects use of the software clipboard).
- Security requirements.
- IT operations, including:
    - Backup.
    - Maintenance windows.
    - Management and deployment.
- Legal guidelines for data retention. Legal requirements must be clearly understood and stated, including what needs to be done to adhere to these requirements. According to Sarbanes-Oxley records and data retention mandates, digital

information should be available and protected from loss, compromise, or corruption. Corporate records managers are responsible for ensuring that electronic business records are properly retained to be available for audits, litigation, or compliance investigations.

- Robustness criteria of availability.

Developing the Solution Concept

The solution concept outlines the path for achieving your business goals and provides the basis for moving into the Planning Phase. After your team has identified the business problem and defined the vision and scope, it creates the solution concept. It is important to understand the target environment for the application migration while coming up with the solution concept. Examples include:

- Migration to a Portable Operating System Interface for UNIX (POSIX) environment on Windows, such as Windows Services for UNIX 3.5.
- Migration to Microsoft Windows Server™ 2003 using Microsoft Win32/Win64.
- Migration of UNIX application to .NET Framework.
- Continued development on UNIX with Windows as a cross-platform migration.

When developing the solution concept, it can be helpful to concentrate less on specific technologies and more on the ways the technology can deliver the business value requirements. When determining whether a solution concept meets a team’s needs, some important criteria to consider are:

- Project success factors.
- Operational concept.
- Acceptance criteria.

The purpose of the solution concept is to:

- Provide teams with limited but sufficient detail to prove the solution to be complete and correct.
- Perform several types of analysis including feasibility study, risk analysis, usability study, and performance analysis.
- Communicate the proposed solution to the customer and other key stakeholders.
- Identify the most suitable migration method.

Comparing Migration Methods

Depending on your goals and how you envision your application to evolve, it may be necessary for you to retain a large code base of installed UNIX applications. There are migration methods that will allow you to preserve your investment in UNIX applications while developing under, or moving to, Windows over a longer period. These methods are:

- **Using a quick port.** A quick port involves migrating your applications to Windows with only the changes that are necessary to allow the applications to run smoothly. One of the simplest possible migration paths is to port the code directly to Windows Services for UNIX 3.5. Windows Services for UNIX 3.5 includes Microsoft Interix, which provides a UNIX environment that runs on top of the Windows kernel, allowing your native UNIX applications and scripts to work alongside the Windows-based applications.

- **Completing a full port**. Unlike the quick port, a full port entails significant alterations to the source code, although the number of modifications depends on the level of standards compliance within the original application.
- **Redesigning and rewriting the application**. Redesigning and rewriting the application is the ideal approach if you want to make full use of all the benefits of migrating to the Windows platform. While initially requiring the greatest amount of work, this course provides benefits in the long run.
- **Allowing two versions to coexist**. With coexistence, you retain the original application alongside the new version while porting or rewriting the application. This method involves significantly reduced risk because it allows you to use the original system if you face unexpected issues with the new application. However, you will need to employ a cross-platform source-code control system to allow concurrent development on both UNIX and Windows.

Comparing Windows Technologies

There are three Windows technologies that can be used to migrate UNIX applications. These technologies are:

- Windows Services for UNIX 3.5.
- Win32/Win64.
- .NET.

Windows Services for UNIX 3.5

Windows Services for UNIX 3.5 provides a full range of cross-platform services for blending Windows and UNIX-based environments. With Windows Services for UNIX 3.5, you can transfer and recompile UNIX-based tools to the Windows platform and extend the value of your UNIX applications. The benefits of moving to Windows Services for UNIX 3.5 include:

- **Maximum return on application investments**. Windows Services for UNIX 3.5 can optimize your investments in UNIX applications by providing the capability to reuse code to run on Windows. It also provides enhanced value to your existing UNIX applications by updating the UNIX code with .NET technology.
- **Reduced development time**. Windows Services for UNIX 3.5 removes the overhead cost of rewriting applications from scratch. Interix subsystem technology saves development time by making it easy to transfer existing UNIX applications to Windows.
- **Reusability of UNIX scripts**. Windows Services for UNIX 3.5 allows you to get the most out of existing UNIX network administrator tools and skill sets. Windows Services for UNIX 3.5 provides more than 300 UNIX tools and shells that enable you to run existing shell scripts on Windows with little or no change.
- **Streamlined network management**. Windows Services for UNIX 3.5 reduces system administration time with tools, such as two-way password synchronization and the Server for NIS components, which centralize network management across the UNIX and Windows platforms.
- **Powerful software development kit (SDK)**. Windows Services for UNIX 3.5 comes with an SDK that supports more than 1,900 UNIX APIs and migration tools (conforming to the IEEE 1003.1-1990 standard), such as make, rcs, yacc, lex, cc, c89, nm, strip, gbd, as well as the gcc, g++, and g77 compilers.

Win32/Win64

Win32 is the 32-bit API for the Windows operating system. Win64 is the 64-bit version of Win32, which includes extension functions for use on 64-bit computers. The Win64 API is found only on 64-bit versions of Windows XP and Windows Server 2003. The benefits of moving to Win32/Win64 include:

- **Expanding application capabilities**. Win32/Win64 provides a tremendous scope to expand the existing capabilities of your applications. Windows APIs provide building blocks that can be used by applications to provide various critical functionalities such as GUI for your application, display graphics and formatted text, and manage system objects such as memory, files, and processes.
- **Powerful application frameworks**. Windows provides powerful frameworks, such as Microsoft Foundation Class (MFC), Active Template Library (ATL), and Graphics Device Interface (GDI+), for application development. These frameworks provide a set of wizards and APIs that reduce the time and effort needed to redevelop existing applications on Windows platforms. Internally, these frameworks communicate with the Win32 subsystem to exploit the capability of low-level Win32/Win64 APIs.
- **Support for 64-bit computing**. For the migration of UNIX 64-bit applications, Windows provides a 64-bit Windows Server 2003 operating system that supports far more physical memory than a 32-bit operating system. Increased physical memory allows more applications to run simultaneously and remain completely resident in the main memory of the system. This reduces or eliminates the performance penalty of swapping pages to and from the disk. A 64-bit operating system also provides a larger virtual memory space for applications, especially database applications that require a larger addressing space.

.NET Framework

The .NET Framework is an integral Windows component that supports building and running the next generation of applications with a consistent, object-oriented programming environment. It supports safe execution of code, eliminates performance problems, and builds such widely varying applications as Windows applications, Web applications, Web services, and deployment applications. It consists of two main parts: the Common Language Runtime and the .NET Framework class library. The benefits of moving to the .NET Framework include:

- **New value from existing assets**. .NET-connected services can help extract new value from existing assets. Organizations can build new and flexible solutions by building on existing technology and exposing existing functionality through Web services.
- **Increased efficiency of business processes and business users**. .NET-connected services can streamline existing business processes and improve efficiency. One way to streamline processes is to link systems so that they can interoperate without manual intervention. This enables the streamlining of processes that once involved multiple systems and eliminates the manual reentry of data and duplication of efforts, thus saving time and money.
- **Language interoperability**. The .NET language allows interoperability that enables developers to make good use of specific features of a language and integrate it easily into your application. For example, to obtain a very good performance, developers may choose to work with C++ with managed extensions.
- **Common development environment and easier debugging**. The IDE Visual Studio® .NET (VS.NET) will be the same for the languages supported by .NET. Application debugging is much easier, and one of the goals of VS.NET is to provide end-to-end debugging from the ASP.NET page to the business components to the database stored procedure.

- **Unified programming class**. Prior to the development of .NET, Microsoft Visual C++® developers had their own library MFC, which was not much help to them in the event of a shift to Microsoft Visual Basic® at a later point. In .NET, the base library will remain the same, irrespective of the language used. Therefore, knowledge once gained can be reused across languages.

Choosing the Appropriate Windows Technology

Migration of a UNIX application to Windows may be as simple as recompiling, or it may require significant redesign and redevelopment. The amount of effort required depends on how portable the application source code is to begin with, whether compatibility tools (such as Windows Services for UNIX) are being used, and the intended use of the application on the Windows platform. The Microsoft recommended order of priority for migrating UNIX applications is .NET, Win32/Win64, and Windows Services for UNIX 3.5.

During the assessment conducted earlier in the current situation assessment process, you answered questions in the assessment tool about your application. You need to now use the results provided by the assessment tool to select the appropriate migration approach. The suggested migration approach (the one with the highest score) is generally the best fit for the migration scenario under consideration and involves the least amount of effort, risks, and deployment issues. The other migration approaches (with lower scores) might present more challenges and issues in the migration process but can act as backup choices in case the best approach becomes difficult to adopt.

Some of the assessment questions, especially the business-level and technology-level questions, may be common to all layers and modules. You will observe at the end of this decision-making exercise that these common questions are the critical driving factors for deciding the appropriate Windows technology to which to migrate.

Note Before you proceed further, you need to have calculated the percentage of recommendations for each migration approach and analyzed the suggested migration options.

Analyzing the various migration options may lead to one of the following situations:

- High recommendations for a single Windows technology.
- High recommendations shared between two Windows technologies.
- High recommendations shared equally among the three Windows technologies.

Also, in the preceding three migration approaches, the customer must evaluate if particular components of the application have an affinity toward a specific Windows technology and, if so, the customer should consider the liability of having one component on a different platform while others are migrated to a new platform.

Note For more information on Windows technologies described previously, refer to the following volumes of this guide:

Volume 2: *Migrate Using Windows Services for UNIX 3.5*

Volume 3: *Migrate Using Win32/Win64*

Volume 4: *Migrate Using .NET*

High Recommendations for a Single Windows Technology

This is a straightforward situation where one particular Windows technology has been given the highest recommendation (highest score). It can be one of the following:

- Quick or a complete port using Windows Services for UNIX 3.5.
- Port or rewrite to Win32/Win64 platform.
- Port or rewrite using .NET.

Several critical factors, as mentioned in the previous section, should be considered when finalizing the Windows technology for the migration process. In specific cases, there may be an equal distribution of high recommendation votes among the critical driving factors. For example, your budget may be low for a static module and, at the same time, you may need better performance than UNIX can provide and need to use a Windows feature like Plug and Play. These are your four critical factors, and all your other noncritical factors suggest moving to Win32/Win64. In this case, two of your critical factors favor the use of Windows Services for UNIX 3.5, while two other critical factors point to Win32/Win64.

The recommended way to resolve such a tie situation is to go back to your core group and identify which of these two sides, taken independently, can override all the other factors in the list put together. In the example, if you absolutely need better performance than UNIX and must use a Windows feature like Plug and Play, then Win32/Win64 is the recommended path. But if your budget is limited and you cannot afford to move to Win32/Win64, then it is recommended that you do not carry out the migration until you have sufficient funds. This is because the majority of your requirements will not be satisfied if you choose to migrate using Windows Services for UNIX 3.5.

High Recommendations Shared Between Two Windows Technologies

This is again a situation where you need to fall back on the recommendations for your critical factors. If the recommendations for these critical factors are shared between two Windows technologies, it is recommended that you use a combination of the two technologies for your migration.

For example, if you have 50 percent of your high recommendations to .NET and 50 percent to Windows Service for UNIX 3.5, then the critical factors are also equally distributed between .NET and Windows Services for UNIX 3.5. In this case, it is recommended that you choose a combination of .NET and Windows Services for UNIX 3.5 to carry out your migration.

High Recommendations Shared Equally Among Three Windows Technologies

When the high recommendations are equally shared between the three Windows technologies (for example, .NET–30 percent, Windows Services for UNIX 3.5–35 percent, and Win32/Win64–30 percent), check for the two technologies that cumulatively have the maximum high recommendations for your critical factors.

For example, if .NET and Windows Services for UNIX 3.5 together satisfy all or most of your critical factors, it is recommended that you choose a combination of Windows Services for UNIX 3.5 and .NET to carry out your migration.

After you have completed the evaluation of all the modules, you need to superimpose all your individual assessments and check for the most highly recommended Windows technology across all modules. If there is a clear indicator, then that is the technology to which you need to migrate. In the absence of a clear indication, it is recommended that you use a combination of the two most recommended Windows technologies (for example, .NET and Windows Services for UNIX 3.5 in this case). Microsoft recommends that all customers move their applications to the .NET Framework. If that's not feasible, then Win32 should be the next choice. If neither .NET nor Win32 is appropriate, then Windows Services for UNIX or Services for UNIX Administration is recommended. If none of these three is possible, then MKS Toolkit or Cygwin are suggested. The following sections describe MKS and Cygwin.

MKS Toolkit

MKS Toolkit for Enterprise Developers (formerly NutCracker) is a UNIX and Windows interoperability suite. This suite allows remote access, interconnectivity, remote system administration, file sharing, and full automation and scripting capabilities.

MKS Toolkit helps in UNIX-to-Windows interoperability by providing:

- A set of UNIX-named tools that work under Windows. Complete shells help in doing shell scripting.
- Interoperability tools, which allow switching from UNIX to Windows environments easily.
- Numerous APIs for porting to the Windows development platform from a number of UNIX versions.
- A set of conversion tools that help update UNIX applications to take advantage of current operating system features.

MKS Toolkit supports synchronous and asynchronous UNIX signals. Job-control signals are not supported but thread priorities are properly supported. MKS Toolkit provides the capability of working with portable code. Therefore, nonportable applications, such as device drivers, do not benefit from the MKS Toolkit.

Note More information on the MKS Toolkit is available at http://www.mkssoftware.com/products/tk.

Cygwin

Cygwin is a Linux-like environment for Windows. The Cygwin tools are part of the popular GNU development tools that have been ported to Windows. They run using the Cygwin library, which provides the UNIX system calls and environment.

With the Cygwin environment, it is possible to write Win32 applications that make use of the standard Win32 API and the Cygwin API. Therefore, it is possible to port many UNIX programs without the need for extensive changes to the source code.

Note The Cygwin system is available at http://cygwin.com/.

Sample Scenarios for Developing a Solution Concept

The following sections describe six example scenarios that illustrate some possible combinations of existing situations and requirements along with the recommended path of migration for the situation/requirements. A sample solution concept is also provided for each example.

Example 1

You have a monolithic, stand-alone application running on a single-processor Sun Solaris platform. The application is written using native UNIX APIs and is not GUI-oriented. Your application is static with no change requests or scalability requirements. The budget for this migration is low, and you want to complete the migration as quickly as possible. Your driver, such as performance requirements, on the Windows platform is the same as UNIX. You do not have anyone in your organization with adequate knowledge of programming in Windows and you are not willing to invest too much money in training your development team on Windows programming.

Solution Concept for Example 1

In this situation, it is recommended that you perform a quick port using Windows Services for UNIX 3.5 if all of your code is compatible with Interix, or perform a complete port using

Services for UNIX 3.5 if the code requires minor changes to make it compatible with Interix. Even a quick port using Windows Services for UNIX 3.5 typically requires some code changes, but it's substantially less than for a full port, which usually means moving to Win32. Because a quick port is defined as "do just enough to get it running under Windows Services for UNIX 3.5," then by those definitions example 1 requires a quick port regardless of how much effort is involved. It is only if you decide to move the application to Win32 without doing a redesign that you really need a full port.

Example 2

You have a decoupled distributed application built using MQ Series running on the IBM AIX platform. Your application is dynamic and needs to be scalable over a period of time. The budget for this migration is high, and you are willing to use your existing application on UNIX for a long duration before migrating it to Windows. The developers on your project have Windows programming knowledge, or you are willing to train them in Windows programming. The application uses UNIX APIs that have equivalents in Windows—for example, file handling commands like open() and close().Your performance requirements on the Windows platform are such that they need to be better than UNIX. You also have an additional requirement of adding Web services to your application.

Solution Concept for Example 2

In this situation, it is recommended that you migrate to .NET by rearchitecting your application and rewriting it in C# primarily because of the Web services technology that .NET offers. If, on the other hand, your requirement for adding Web services is a long-term plan and you want to retain your existing application architecture, then it is recommended that you move your application to Win32/Win64 because the time required to rewrite your application is faster. The task of rewriting will consist of replacing the existing UNIX commands with Windows equivalents—for example, OpenFile() and CloseHandle(). You have the option of running MQ Series on Windows or rewriting the application to use the Windows Message Queuing API.

Example 3

The application code has less than 5000 lines written using native UNIX APIs.

Solution Concept for Example 3

In this situation, it is recommended that you rewrite the entire application using C or C++ on Windows. If you do not have very stringent performance requirements, but you need extensibility for your code because your code requirements are constantly changing, then it is recommended that you use C# as the programming language.

You may have observed that in this particular situation, we have not gone into the details of the other requirements because of the size of your application code. It is recommended that you rewrite the application on Windows, based on the time and effort required to rewrite such a small-scale application. In addition, if your application does evolve over a long period, it helps that the code is available on Windows.

Example 4

You have a client/server application running on the SGI IRIX platform. Your server side is static while the client is dynamic and needs to evolve over a period of time. The budget for this migration is low, and you are doing a pilot project to see how the application runs on Windows. You have developers with Windows programming knowledge within your organization, or you are willing to train them in Windows programming. The third-party tools that you have used to develop your GUI code in UNIX are not X Windows and do not have a Windows equivalent, or you want to develop a GUI for your application based on Windows forms to get the Windows look and feel.

Solution Concept for Example 4

In this scenario, it is recommended that you perform a complete port of your server side using Windows Services for UNIX 3.5 by rewriting the UNIX code so that it is compatible with POSIX or by creating Interix wrappers on the UNIX code that is not compatible with POSIX. The client part of your code can be written using GDI+ on Windows. Depending on whether the GUI is a thick client or thin client, you will also need to write wrappers in Interix around your server code to talk to the Windows-based GUI. In case your existing GUI code on UNIX is written using X Windows, it is recommended that you rewrite the X Windows code to be compatible with Windows because some of your existing code may be reusable.

Example 5

You have a highly dynamic Web application that is bound to evolve over a period of time. You want to move away from your existing environment, so there is no requirement to maintain your code on several platforms concurrently. Your budget for migration is high, and you expect a better performance on Windows platforms. You have a high requirement for a Windows look and feel in the Web pages and such features as transaction handling, messaging, and security. There is a lot of messaging and data exchange between the modules in your application. The application is written using C++ and Perl. You have developers with Windows programming knowledge within your organization, or you are willing to train them in Windows programming.

Solution Concept for Example 5

In this scenario, it is recommended that you migrate to .NET by rearchitecting your application and rewriting it in C# primarily because of the extensibility and scalability that .NET offers to you in the long run.

Example 6

You have a 64-bit application.

Solution Concept for Example 6

In this scenario, it is recommended that you rewrite the application on Win32/Win64 because .NET and Windows Services for UNIX 3.5 do not currently support 64-bit applications. If your application is written using native UNIX calls, it is recommended that you rewrite them with Windows calls. If your application is using third-party tools, it is recommended that you use their Windows equivalent, if they have one, or redesign and redevelop the application on Windows.

Assessing Risk

A risk is an event that can jeopardize the successful completion of a project or the ongoing success of the deployed application. Any condition that may create an unfavorable outcome and whose occurrence is uncertain is a risk. The objective of risk management is to identify these conditions; prioritize them based on their impact on project goals and the probability of the occurrence of these conditions; identify, plan, and execute mitigation steps; and track the risks during the entire project life. During project execution, additional risks may appear and these must be addressed similarly.

Risk is also the possibility of suffering loss. This loss may be in the form of diminished quality of the migrated application, increased development and support costs, missed deadlines, or complete failure to achieve the mission and purpose of the project. In other words, a risk is a problem waiting to happen. It is recommended that the Microsoft Solutions Framework (MSF) approach to risk management be adopted, as illustrated in Figure 2.1. This approach allows you to retire risks through a process of identification, analysis, mitigation planning, tracking, and control.

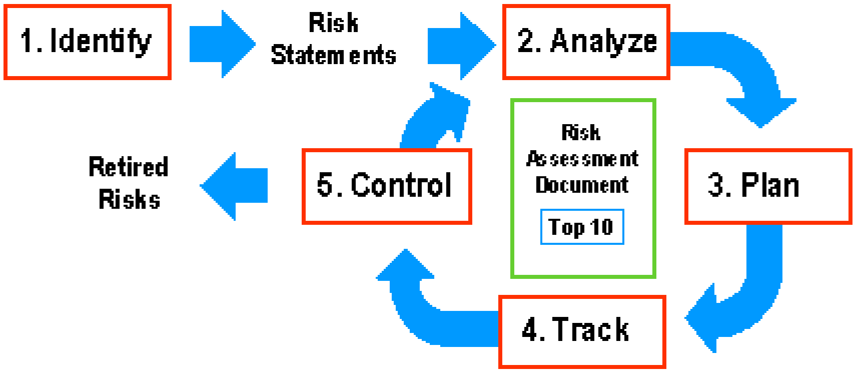


Figure 2.1. The MSF approach to risk management

Identifying Risks

Throughout the Envisioning Phase, you will identify and document issues surrounding the migration. These issues may be business-, technical-, application-, or infrastructure-specific. Before you can create a realistic project plan, you must determine whether any of these issues pose a serious risk to the project and what should be done to minimize the probability of things going wrong.

In general, the more potential stumbling blocks—attitudinal, structural, or otherwise—you identify ahead of time, the easier it will be for you to minimize the amount of time spent trying to resolve them during migration.

Risk Analysis and Mitigation

To help you with risk analysis and mitigation as well as risk identification, a Risk Assessment Tool is included in the Tools and Templates folder with the downloadable version of this guide. This job aid lists some of the critical risks that may be encountered in a UNIX-to-Windows migration project and the plans for mitigating them. The risks are categorized into generic, business, architectural, and technical risks. Any additional risks that you have identified during the course of the Envisioning Phase needs to be added to that list and a suitable risk mitigation strategy should be planned.

Closing the Envisioning Phase

The Envisioning Phase ends when the team, the customer (business sponsor), and major project stakeholders review the vision/scope document and formally signify their agreement with its terms. Securing this agreement indicates achievement of a major project milestone, Vision/Scope Approved, which is required to proceed to the next phase, the Planning Phase. At this point, the business and IT should have a clear idea of the goals for the project (and all its associated risks), and the project team can now begin to make specific plans for how to achieve them.

Key Deliverables from the Envisioning Phase

The key deliverables that should be produced by the end of the Envisioning Phase are:

- Project structure document
- Vision/scope document
- Risk assessment document

Key Milestone: Vision/Scope Approved

Reaching the Vision/Scope Approved Milestone concludes the Envisioning Phase. At this point, the project team and the customer have agreed on the overall direction for the project, including the features to be included in the project scope within a specified time of delivery.

Project teams usually mark the completion of a milestone with a formal sign-off. The sign-off document becomes a project deliverable and is archived for future reference.

Chapter 3: Planning Phase: Planning Your Migration Project

This chapter provides the background and technical information required to complete the Planning Phase of a migration project.

Introduction to the Planning Phase

The Planning Phase is the time when the project team translates the initial vision/scope from the Envisioning Phase into practical plans on how to achieve it. The purpose of the Planning Phase is to define the solution in detail along with the approved project plan and schedule. This work includes creating a functional specification, developing the solution architecture and design, and preparing cost estimates. Team members draw upon their expertise to create detailed individual plans, such as the development plan, test plan, and deployment plan, as well as schedules for all aspects of the project. Program Management combines these individual plans and schedules and synchronizes them to create the master project plan and schedules. The Planning Phase culminates in the Project Plans Approved Milestone. Passing this milestone indicates that the customer, the project team, and all stakeholders agree on the details of the plans, including what will be built, how it will be built, when it will be delivered, and what it will cost.

Planning Phase Tasks

The major migration tasks conducted during the Planning Phase are summarized in the following list. They will be described in more detail in the subsequent sections.

1. **Developing the solution design and architecture**. The development team begins the design process with the solution design and architecture and culminates it with a design document that becomes part of the functional specification.
2. **Validating the technology**. The development team also validates technologies to ensure that they meet the business needs for the specific solution.
3. **Creating the functional specification.** The project team and Program Management Role create a functional specification that describes the solution requirements, the architecture, and the detailed design for all the features. This represents the contract between the project team and customer.
4. **Developing the project plans**. The Program Management Role and the various teams that make up the project team develop a collection of plans to define the tasks for all six MSF team roles, and Program Management consolidates them into a master project plan.
5. **Creating the project schedules**. The Program Management Role and the various teams create milestone-driven schedules for each individual team role, and Program Management consolidates them into the master project schedule.

6. **Setting up the development and test environment**. The development and test teams create development and testing environments that are independent of the production environment to develop and test the solution.
7. **Close the Planning Phase**. The project team completes the Planning Phase with the approval process for the Project Plans Approved Milestone.

Note Although listed sequentially, many of these activities can be performed concurrently.

Planning Phase Deliverables

The Planning Phase activities culminate in a major milestone, the Project Plans Approved Milestone. By the end of the Planning Phase, the project team and all major stakeholders (other members of the organization who will be affected by the project) should have agreed upon the functional specification, technology for the solution, and project plans and schedule. These deliverables include:

- A technology validation-complete document that:
    - Documents the current environment.
    - Develops and tests the key features in the solution with a proof of concept.
    - Documents all the potential issues and their resolution.
- A functional specification document that:
    - Describes requirements of all the components of the solution.
    - Describes solution design and architecture of the components.
    - Provides quantitative specifications about performance measures, database capacity, and concurrent usage and security measures.
- A master project plan of the solution that:
    - Contains individual plans for various roles.
    - Guides the project completion as per the functional specification.
- A master project schedule that:
    - Integrates all the schedules along with release dates.
    - Creates awareness about project priorities.
- A risk management document that:
    - Identifies the potential risks and mitigation strategies.
- An instruction document on setting up the development and test environments that:
    - Creates a proper development and testing environment for the solution without affecting the production system.
    - Identifies the hardware and infrastructure requirements for the environment.

Together, these deliverables comprise a high-level design description and plan of the project that form the basis for the subsequent phases. Therefore, the functional specification document, especially, must be viewed as a living document that keeps changing, subject to change control. The deliverables may undergo numerous iterations before the project team, the customers, and the stakeholders reach a final consensus.

Note For more detailed discussions on how these tasks and deliverables can be approached and the responsibilities assigned for them, refer to the *Unix Migration Project Guide* (UMPG) at http://www.microsoft.com/technet/itsolutions/cits/interopmigration/unix/umpg/default.mspx.

Instructions for setting up the development and test environments are explained in Chapter 4, "Planning: Setting Up the Development and Test Environments" of this volume.

Planning Phase Activities

The following sections detail the various activities involved in the Planning Phase of the MSF Process Model and how these activities specifically relate to a migration project.

Developing the Solution Design and Architecture

The development of the solution design and architecture begins with a design process, the results of which become the functional specification. The design process helps identify the project team structure and the team's responsibilities for the upcoming Developing Phase. The foundation of the design process is the vision that the team developed and the business goals that were gathered during the Envisioning Phase. The architectural design describes how features and functions operate together to form the solution. It identifies the specific components of the solution and their relationships.

The design document contains details of the architecture and the components that go into building the solution. For UNIX-to-Windows migration projects, the solution architecture remains the same; however, to include it in the design document ensures completeness. This helps the team to work in a systematic way—from abstract concepts in the vision/scope document down to specific technical details in the design process. It also helps to maintain the correlation between the requirements and the solution features.

Conceptual Design

The conceptual design stage includes the process of analyzing and prioritizing business and user perspectives of the problem and the solution, and then creating a high-level representation of the solution. This stage helps in mapping the functionality associated with each of the requirements.

A conceptual design is a means to understand the business expectations and application requirements that include both technical and infrastructure requirements in terms of business, user, system, and operational requirements. The design formulates the solution to be developed, keeping in mind the end users and the business requirements. It is therefore essential to answer all questions in the assessment checklist provided with this guide. This helps to assess the current situation and further define the project scope developed during the Envisioning Phase in order to obtain a clear understanding of the functionality required to build the solution.

The conceptual design lays the foundation for developing the solution and addresses the requirements by describing the design and architecture of its components.

For migration projects, the conceptual design is generally identical to the original functionality of the current application or infrastructure component. It is nonetheless important to articulate the existing design in the functional specification for the migration project because the actual concept for the current component may have drifted from its initial conception. Even if that conceptual design has remained constant, it serves as a touchstone for subsequent design phases.

The following example demonstrates what is meant by conceptual design.

Example of a Conceptual Design

Consider an engineering graphics application developed in UNIX that is used by the design staff within an organization. Because of evolving business requirements and a globalized environment, the organization now wants to make use of the capability of its external partners to provide design solutions using the same application.

To achieve this, the application needs to be available on the Microsoft® Windows® platform, which most of its suppliers have already standardized on.

Migrating this application to the Windows platform enables it to be shared with the partners, resulting in an increased degree of collaboration between users who use the application, both within the organization and outside.

The conceptual design should document any unique requirements that would arise in this new environment and verify that the proposed solution architecture caters to these requirements.

Logical Design

During the logical design stage, each part of the conceptual design is assigned to a specific role within the architecture of the solution. It provides a clear view of the solution from the functional perspective. A logical design identifies and defines all the objects and their behaviors, attributes, and relationships within the scope of the solution. The application design is split into three levels: presentation, business, and the data layer. For a migration project, you must document the existing logical design as well as the logical design of the migrated application or infrastructure component and emphasize the areas of change if applicable. It is also important to show how the migration project affects the other components outside the scope of the project.

Example of Logical Design

Continuing with the same engineering application example, the logical design documents the existing as well as the new architectural components required to realize the conceptual design. The presentation layer can be achieved by a Windows user interface (UI) instead of the existing X-Motif–based user interface. The communication layer can be achieved by Winsock or .NET messaging in place of the existing UNIX sockets calls.

It may also be necessary to show how the migrated applications interact with other components outside the scope of the migration project. For example, it is possible for applications on the partner’s side to exchange data with the migrated application on Windows.

Physical Design

The physical design of the solution identifies the pieces from the logical design that must fit into the physical architecture. The physical design identifies the physical infrastructure architecture and topology. It creates a set of physical design models, including the component’s design, UI design, and a physical database design for the applications. The physical design should include anticipated metrics to assess performance goals, uptime goals, and milestones for writing the solution code. For example, the physical design might include metrics for transaction time and performance requirements for the transactions before deployment. Production metrics for the particular deployment scenarios must also be established.

The physical design is a complete implementation design, in the form of a technical specification, that the development team uses to build the solution. For a migration project, the physical design should also include the process of implementing the infrastructure and the step-by-step details of how to deploy the migrated application, keeping in mind the current milestones of the application. It must also cover how the new implementation satisfies the business requirements without violating ongoing service level agreements (SLAs).

Example of Physical Design

The physical design of the application might describe which components in each of the layers (presentation, business, and data) need to be changed in one of the following ways:

- Ported to recompile and fix problems that arise.
- Rewritten if there is no corresponding library or component available on Windows.
- Replaced if an equivalent library is available in Windows.
- Purchased if the library or component is to be bought from third-party vendors.

It might also provide a detailed mapping of the source UNIX architecture to the target Windows architecture, where each component/library of the UNIX application is mapped to one that provides equivalent functionality on Windows.

Validating the Technology

Often, in tandem with the design process, your team can also validate the technologies being used in the solution. In the technology validation process, the team evaluates the products or technologies to ensure that they work according to the specifications provided by their vendors and that they address the business needs for the specific solution scenario. Validating the technology is an essential step in a UNIX-to-Windows migration project because the tools, software, and hardware in the new Windows environment must work together to produce the same or better effect than that produced in the UNIX environment. For example, if the UNIX application uses a third-party library and if a Windows version of it is also available, it would be a good idea to check if the Windows equivalent of the library works according to the required specifications.

Technical Proof of Concept

After validating the technologies, the team makes an initial attempt at creating a model of the technology that will be implemented. This produces a proof of concept. The initial proof-of-concept model often produces both answers and additional questions for the team about the issues that might arise with the technology during the Developing Phase. This information helps in the risk management process and identifies overall design changes that must be incorporated into the specifications.

The various libraries or modules in the UNIX application can be listed in the functional specification document, as shown in the following proof-of-concept identification table for a sample banking solution application.

Table 3.1. Sample Proof-of-Concept Identification

Name of Library or Module	Area or Layer of the Application	Functionality Covered	Prototype Required (Y/N)?
LoanInterestCalLib	Loans–Business Layer	Interest calculating routines	N
AuthenticationLib	User Authentication–Business Layer	Secured login routines	N

Name of Library or Module	Area or Layer of the Application	Functionality Covered	Prototype Required (Y/N)?
Super-annuityLib	Retirements–Business Layer	Retirements routines	Y
DataAccessLib	DataAccess Layer	Database routines	Y

In a UNIX-to-Windows migration project, the proof of concept typically addresses such areas as the compatibility and suitability of certain third-party libraries in the Windows environment, the manner in which certain UNIX-environment–specific features are handled in Windows, and how the performance of the application in the Windows environment will change. It may even involve the porting of a small portion of the application to confirm the migration methodology. If possible, steps should be taken to create a reusable prototype, which can be used again in the actual migration phase.

The following are examples of proof of concepts:

- Porting of a small portion of the communications library in the UNIX application to Windows in order to establish the communication model between the migrated Windows client and server.
- Porting of a text-based library of the UNIX application, which uses GLX (OpenGL Extensions) for character rendering to Windows and uses WGL (Wiggle) to compare the rendering speed of characters.

Baselining the Environment

The team must also conduct an audit of the as-is production environment in which the solution is now operating. Information is collected on server configurations, network, desktop software, and all relevant hardware. This baseline allows for the team to account for any changes that might be required or design issues that might cause the solution to be at risk. This baselining step is critical to the success of a migration project.

Interim Milestone: Technology Validation Complete

At this point, the team has confirmed the technologies to be used and should be well versed with the technical issues in order to move forward with creating the functional specification. The team also validates the technology, considering the implementation of the features from the business perspective. When you reach this milestone, you are ready to create the functional specification for the solution.

Creating the Functional Specification for the Solution

The functional specification document is a technical description of the solution and represents the contract between the customer and the project team. It is the basis for building project plans and schedules for the migration project. The program manager takes the lead in creating the functional specifications, with input from the role leads regarding their areas of responsibility. During the Planning Phase, the functional specification document is baselined and, after a formal review from the project manager, is taken as the final document. Otherwise, a change document is prepared for the functional specification.

The Solution Feature Set

A basic functional specification document must include the following:

- A summary of the vision/scope document as agreed upon and refined, including background information to place the solution in a business context.
- Any additional user and customer requirements beyond those already identified in the vision/scope document.
- The solution design (including conceptual, logical, and physical designs).
- Specifications of the components that will be a part of the solution.

Interim Milestone: Functional Specification Baselined

The functional specification is maintained as a detailed description of the various solution components and how the solution will look and operate. The team baselines this functional specification and formally tracks the changes based on the customer's approval of the changes. The functional specification is the basis for building the master project plan and schedule. When you reach this milestone, you are ready to develop plans to execute the project.

Developing the Project Plans

During the Planning Phase, you must map the requirements to the conceptual, logical, and physical designs, as well as plan for the future phases of the project. Plans need to be developed for developing or migrating, stabilizing, deploying, and operating the system, and also for other aspects of the project such as the budget, team communication, facilities, and purchasing. Depending on the size and complexity of the project, the number of plans can vary. The essential plans that are a part of the master project plan are discussed in this section. The program manager is responsible for creating the master project plan components in consultation with various teams. The master project plan consolidates the feature team and role plans and guides the project to completion, as defined by the functional specification. It is a key Planning Phase deliverable.

The benefit of having a plan that breaks into smaller plans is that it:

- Facilitates concurrent planning by various team roles.
- Keeps accountability clear because specific roles are responsible for various plans.

The benefit of presenting these plans as one Master Project Plan is that it:

- Facilitates synchronization into a single schedule.
- Facilitates reviews and approvals.
- Helps identify gaps and inconsistencies.

Note Sample MSF deliverables, templates, and white papers are available at http://www.microsoft.com/technet/itsolutions/msf/default.mspx.

The following table lists the various essential plans that are part of a master project plan, describes the purpose of each plan, and names who is responsible for creating the corresponding plan.

Table 3.2. Preparing the Master Project Plan

Name of Plan	Purpose of Plan	Responsibility
Budget and Purchasing Plan	To establish the estimated cost of migration and determine what resources and infrastructure requirements will be required for the migration.	Program Management
Solution Development Plan	To validate the feasibility of the migration strategy and identify requirements for environment readiness.	Development
Solution Testing Plan	To ensure the quality of the migrated solution, test plan, and test methodology.	Test
Pilot Deployment Plan	To ensure smooth deployment at production.	Release Management
Production Deployment Plan	To specify the deployment instructions on the production computer configuration.	Release Management
User Operations Training Plan	To ensure that the UNIX users are comfortable with the migrated Windows applications.	Release Management
Security Plan	To make the migrated application as secure as it was on UNIX.	Development and Release Management
Communications Plan	To define the communication strategy with the customer.	Product Management

Budget and Purchasing Plan

Adequate financing is crucial for a project to survive, and therefore a budget plan is important. You may get into a situation where you need resources that your project budget cannot accommodate. The needs and demands of all the phases of the project must be taken into account when planning the budget.

The budget plan must include the following information:

- Software costs, including those of the operating system, development tools, and management tools.
- Hardware costs, including those of computers, power supplies, racks, and cables.
- Personnel costs, including those for executing the migration and maintaining the system post-migration.
- Cost of training resources, including those associated with developers, administrators, and end users.
- Cost of vendor support for software, hardware, and network.
- Miscellaneous costs, such as those for travel and shipping.

The purchasing plan includes (but is not limited to) the following information:

- Equipment required for the setup, such as racks, power points, and uninterruptible power supply (UPS).
- Test setup resources.
- Systems required for the setup.

Solution Development Plan

The development plan describes the solution development process for the project, in addition to the tasks to create and assemble the components of the solution. A development plan for a UNIX-to-Windows migration project should include (but not be limited to) the following information:

- How to set up an Interix or Win32/Win64 or .NET development environment for the migration.
- How to set up the test environment to migrate the application code.
- The different components of the application and an indication of the components that will be migrated and the ones that will be replaced.
- After the various scripts, modules, and tools of the UNIX system have been successfully migrated to Windows, they need to be integrated into the Windows environment. They must work together with other dependent applications. This development must precede the development of the application. Integration activities must be added to the existing development plan.
- A list of the porting methodologies that can be used for the application. (Porting methodologies are discussed in detail in the build volumes [Volumes 2, 3, and 4] of the guide.)
- Configuration management is the process used to control, coordinate, and track code, requirements, documentation, issues, change requests, tools, changes affected, and the people making the changes. Configuration management activities must be added to the existing development plan.

Solution Testing Plan

The testing team tests the migrated application. Testing can begin before the entire development is complete. It must include tests for security, scalability, and performance of the application, along with the functionality tests of the migrated application.

A testing plan describes the strategy and approach that is used to plan, organize, and manage the testing activities in a project. It identifies the testing objectives, methodologies, tools, expected results, responsibilities, and resource requirements. The Test Role is responsible for creating the test plan. The test plan must include (but is not limited to) the following information:

- Manual procedures or script to test if the setup works.
- Testing procedures (testing methodology).
- Test cases to test the functionality of the application.
- Test cases to test the interaction with other applications.
- Expected results compared with the existing UNIX application functionality.
- Assumptions made prior to testing.
- Bug reporting and tracking mechanisms.
- Plans to improve the performance of the application and the application infrastructure.

Pilot Deployment Plan

The pilot deployment plan addresses the initial deployment of the solution into the production environment. The pilot deployment plan includes (but is not limited to) the following information:

- Pilot participant profiles and their selection process.
- The number of pilot builds and the number of participants in them.
- The procedure for the pilot deployment.
- Backup and recovery mechanisms in case of deployment failures.
- Mechanisms to gather feedback on the pilot application.

Production Deployment Plan

The production deployment plan addresses the deployment of the migrated application based on various scenarios that you have encountered and the best practices that you have drawn up for deploying the pilot application. The production deployment plan must include (but is not limited to) the following information:

- Details of the deployment process for the operating system.
- The modules of the migrated application to be deployed.
- The tools to be used for the deployment of the migrated application and other software.
- The deployment procedure.
- Backup and recovery mechanisms in case of deployment failures.

User and Operations Training Plan

The user and operations training plan focuses on the process that the users and their operations should follow to make a successful transition from the UNIX environment to the new Windows environment. It includes details on the process that the Release Management Role should follow to coordinate with the current operations team to ensure a smooth migration and rollover. The user and operations training plan includes (but is not limited to) the following information:

- Details of the training needs of the existing users, which will enable them to work on the migrated application and the Windows environment.
- Details on making software and hardware inventories.
- Details on network administration and security administration in the new environment.
- Backup procedure details for the new system.
- Details on monitoring the new system's performance on a daily basis.

Security Plan

Security is often overlooked during the Planning Phase. The importance of security is only realized when you begin working on the system. A security plan includes (but is not limited to) the following information:

- Details of various roles and users who may access the application.
- Types of access rights given to various user groups.

- Response measures for a possible security breach.
- Physical security plans.
- Details on how users can access the setup location.
- Details on the roles that can access the application for infrastructure maintenance.

Communications Plan

Communication between various roles and with the customer is very important for the success of a project. Most delays and wrong executions are usually due to a communication gap. A communications plan should contain (but is not limited to) the following information:

- Procedures for escalating issues.
- Procedures for managing status updates.
- Types of issues and the roles that will handle communication regarding the issues.

A service level agreement (SLA) can be drawn up at the start of the project. This lists the communication channels and the levels of escalation for each.

Interim Milestone: Master Project Plan Baselined

At this point, the team has rolled up all initial drafts of the plans required to create estimates for the time required to fulfill these project plans.

Creating the Project Schedules

The *UNIX Migration Project Guide* (UMPG), a companion to this guide, contains most of the information that you will need to develop a schedule for your migration project. Some important points to consider while creating the project schedules are as follows:

- The individual schedules must fit into the overall migration project schedule.
- When drawing up project schedules, it is best to define the milestones early in order to establish the proof of concept. In a UNIX-to-Windows migration project, the proof of concept provides valuable inputs for overall project estimations by giving a realistic picture of the expertise and the time required to complete the application by extrapolating on the proof-of-concept time and the overall cost of the project.
- You may want to treat the application infrastructure migration separately from application migration. If you do, you may have parallel milestones. In a UNIX-to-Windows migration project, the effort required to procure the hardware and software is often underestimated. Effort estimations are often based solely on the size of the application, which results in an inaccurate estimate. A project schedule must therefore include the hardware and software procurement time, the time required for the initial setup of the infrastructure, and the time needed to establish the proof of concept.
- The project schedule must be drawn up by the Program Management Role in consultation with representatives of all the roles because this will provide estimates for all areas of the project.

Interim Milestone: Master Project Schedule Baselined

At this point, the goal for the schedule should be set. It creates awareness about the project’s priorities and contributes to a sense of ownership by all the project’s contributors.

Interim Milestones

This chapter discussed the major tasks of the Planning Phase (with the exception of setting up the development and test environments and closing the Planning Phase). The key interim milestones that accompany the activities performed in this chapter are:

- Technology Validation Complete
- Functional Specification Baselined
- Master Project Plan Baselined
- Master Project Schedule Baselined

The next chapter discusses setting up the developing and test environments as part of the Planning Phase and their importance in improving the productivity of the development team.

Chapter 4: Planning Phase: Setting Up the Development and Test Environments

After finalizing the project plans and project schedules as part of the Planning Phase, you need to focus on setting up the development, test, and staging environments. The setup and management of a dedicated development, test, and staging environment is required in the Planning Phase to ensure smooth migration in the later phases. This is done after defining the solution design and architecture as discussed in Chapter 3 of this volume. The development and test environments allow proper development and testing of the solution so that it will have no negative impact on production systems. They must be properly set up before moving into the Developing Phase in order to maximize development team productivity and to mitigate risks. To avoid delay, the development and testing environments should be set up even as plans are being finalized and reviewed. The Microsoft® Solutions Framework (MSF) Release Management Role is typically responsible for performing these tasks for the project team. Setting up the development and test environments in the Planning Phase helps you to identify the infrastructure requirements and tools required for the development and testing activities. It also gives you instructions about the configuration of the integrated development environment (IDE) for development activities.

Setting Up the Development Environment

The development environment consists of the hardware, software, and tools that enable development activities in the migration project. The development environment includes the compilers, libraries, tools for debugging, source code management, and the IDE. This section provides an overview of the Microsoft Windows® operating system development environment. In addition, this section discusses the development environments for Windows Services for UNIX 3.5, Win32®/Win64, and .NET.

The Windows operating system development environment includes the following:

- SDKs such as Interix SDK for Windows Services for UNIX, Platform SDK for Win32/Win64, and .NET Framework SDK for .NET (which include compilers, editors, and debuggers) provide the basis for creating, building, and debugging the application.
- IDEs, such as Visual Studio .NET 2003, provide uniform and coherent access to tools.
- Software management tools such as software for source code management and problem tracking are not critical, but they add significant value to larger migration projects.
- Analysis tools, such as performance analysis and testing tools, can be used to improve the performance of the application.

- Cross-platform tools and libraries, including third-party packages such as Rogue Wave SourcePro, can be integrated with the application.

The facilities provided in the development environment, such as compilers, source control system, IDEs, and tools, are used by both the developers and the testers of the system. The decisions made while developing the migration strategy will help define the development environment. This environment is either native Microsoft Win32/Win64 application programming interface (API), Interix, or Microsoft .NET.

The solutions recommended in this chapter use the Microsoft Visual Studio® .NET 2003 integrated development environment (IDE) for Win32/Win64 source editing, builds, and makefile-based batch builds. Interix solutions use the Microsoft Visual Studio for source editing and the included Interix SDK (software development kit) tools for compiling, linking, and debugging (**gcc** and **gdb**). .NET solutions use Visual Studio .NET 2003 IDE for source editing, compiling, linking, and debugging. Win32/Win64 solutions and .NET solutions are integrated with Microsoft Visual SourceSafe® as version control software for source code control. Interix solutions use out-of-the box Revision Control System (RCS) for source code control.

This section describes how to set up a development environment for migrating applications and the major tasks involved in setting up the development environment. These tasks include:

- Identifying the tools and IDEs.
- Migrating source under source control.
- Configuring the development environment.
- Creating projects.
- Populating the development environment.
- Migrating the build environment.

The following sections describe each of these topics in detail.

Identifying the Tools and IDEs

The following sections identify the tools and IDEs required for developing the applications using the following Windows technologies:

- Windows Services for UNIX 3.5
- Win32/Win64
- .NET

Each section describes the necessary tools required for the development environment and is organized as follows.

- **Software development kit (SDK)**. The SDK provides extensible options for developing the application using its code files, import libraries, and the application programming interfaces.
- **Compiling tools**. A compiler is a specialized computer program that converts source code written in one programming language into another intermediate language or computer language so that it can be understood by computer processors.
- **Debugging tools**. Debugging is the technique of determining the cause of the symptoms of malfunctions in the source code. The debugging tools allow you to trace the source code by adding break points and viewing the data at that point during runtime.

- **Application build tools**. Application build tools help you in developing and designing the application by reducing the development time and improving the reusability.
- **Third-party extension tools**. Third-party tools work as add-ins to the existing IDE or as stand-alone applications to obtain the additional functionality.
- **Source code management tools**. Source code management tools help you manage multiple versions of source files in a space-efficient manner by storing only the differences between the versions and configuration management.

Note Some of the topics described previously are not applicable for Windows Services for UNIX 3.5.

Windows Services for UNIX 3.5

The following tools and IDEs are useful for developing applications using Windows Services for UNIX 3.5.

Interix SDK

The Interix SDK for Windows Services for UNIX 3.5 contains documentation, tools, API libraries, and headers that the language compilers require to port UNIX applications to Windows. With the Interix SDK, you can host your own tools and applications alongside Windows Services for UNIX tools and applications.

The Interix SDK includes a UNIX development environment, with tools such as the GNU **gcc**, **g++**, and **g77** compilers, and the **gdb** debugger. The Interix SDK also provides UNIX-style command-line interfaces through the **cc** and **c89** compiler drivers for Visual Studio .NET 2003. You can compile C programs to use the benefits of the native compiler for Windows. The **cc** and **c89** tools work only with the Visual C++® compiler. These tools do not work with the **gcc** compiler. You cannot compile C++ code with the **cc and c89** interfaces. You must use **g++** for a C++ code.

The Interix SDK documentation includes developer guides and references for all POSIX.1 system interfaces and headers, Interix extensions to POSIX.1 and POSIX.2 interfaces, and the International Organization for Standardization/American National Standards Institute (ISO/ANSI) C libraries. The Interix SDK documentation also provides information on designing and building UNIX daemons as services, curses, and X Windows-based applications and porting UNIX code along with various categories of APIs and services.

Interix provides a rich set of command-line and stand-alone tools for building, debugging, and testing applications. The following tool categories are also delivered with the SDK:

- Compiling (**cc**, **c89**, **gcc**, **g++**, and **g77**)
- Linking (**ld**)
- Debugging (**gdb**)
- File management
- Performance monitoring tools

Integrated Development Environments

For creating applications using Windows Services for UNIX 3.5, you can use Visual Studio or Visual Studio .NET 2003 as integrated development environments.

For more information on this, refer to ".NET" section later in this chapter.

Compiling Tools

The Interix SDK provides compilers to run C, C++, and Fortran programs using the Interix SDK API. It includes compliers like **cc**, **c89**, **gcc**, **g++,** and **g77**. You can compile your programs from the command line using the makefiles. You can also specify appropriate complier options for debugging, preprocessing, and optimizing the programs. The compilers **cc** and **c89** work as an interface to the system C compiler. If Microsoft Visual C++ is installed, Windows Services for UNIX setup configures **cc** and **c89** complier interfaces to a Visual C++ compiler. Both **cc** and **c89** invoke the Microsoft Visual C++ tools **Cl.exe** for compiling and **Link.exe** for linking.

For more information on compiler options, refer to help documentation for Services for UNIX 3.5.

Debugging Tools

The Interix SDK contains the GNU debugger (**gdb**). The debugger attaches to the virtual file system. It provides information only for object files compiled with **gcc** and **g++,** not for files compiled with **cc** or **c89** compilers. The Interix implementation of the **gdb** debugger works with shared libraries.

For more information, refer to refer to help documentation for Services for UNIX 3.5 or the **gdb** help function.

Source Code Management Tools

Table 4.1 describes the source code management tools that can be used by Windows Services for UINX 3.5 to manage source code.

Table 4.1. Source Code Management Tools for Windows Services for UNIX 3.5

Tools	**Description**
RCS	The Revision Control System (RCS) manages multiple revisions of software files. RCS can store, retrieve, log, identify, and merge file revision data. The RCS tools provided by the Interix SDK are based on the RCS 5.6 tools.
CVS	The Concurrent Versions System (CVS) is the dominant open-source, network-transparent version control system. This tool does not include the Interix SDK by default. You can download it at http://www.interopsystems.com/tools/warehouse.htm.

Win32/Win64

This section discusses the tools and IDEs that are useful for developing applications using Win32/Win64.

Windows Platform SDK

A Windows Platform SDK provides tools to help build, debug, test, and deliver applications. It also provides all the definitions (include files) and libraries that are needed to compile programs. In general, SDK tools are run from the command line, just as applications are run from a shell prompt in UNIX. Output from SDK tools can be files created on the disk, text output to the console (like **stdout** in UNIX), or graphical output to one or more dialog boxes.

The Windows Platform SDK is a set of tools and API definitions that help create applications for the Windows platform. The SDK includes files and documentation for all subsystems and APIs in Windows. It also contains definitions and documentation for the Microsoft .NET Enterprise servers, including Microsoft BizTalk® Server, Microsoft Commerce Server, and Microsoft SQL Server™. Installation of the .NET server components is optional.

ThePlatform SDK is available on CD and as a free download on the Web. It is also available with Visual Studio .NET 2003 or with an MSDN Professional or Universal subscription. You can order or download the SDK at http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en.

Integrated Development Environments

For creating Win32/Win64 applications, you can use Visual Studio 6.0 or Visual Studio .NET 2003 as integrated development environments. This guide recommends using Visual Studio .NET 2003 as the IDE for developing Win32/Win64 applications.

Visual Studio 6.0

The Visual Studio 6.0 integrated development environment can be used as the IDE for developing applications in Win32/Win64 with the Windows Platform SDK.

Visual Studio .NET 2003

The Visual Studio .NET 2003 IDE can also be used for developing applications using Win32/Win64. For more information, refer to ".NET" section later in this chapter.

Compiling Tools

The Platform SDK provides compilers to run C or C++ programs using the Windows API, targeting 32-bit and 64-bit platforms. The IDE can be used to compile applications using the compiler with the Platform SDK by making the appropriate settings for 64-bit applications. For more information on this, refer to Chapter 2, “Developing Phase: Process Milestones and Technology Considerations” of Volume 3, *Migrate Using Win32/Win64* of this guide.

You can also compile your programs from the command line instead of using the IDE. You can also specify several appropriate options for debugging, preprocessing, and optimizing the programs, both from the IDE and the command line.

Note More information on C++ compiler options is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/vcrefcompileroptionslistedalphabetically.asp

Debugging Tools

The Visual Studio 6.0 or Visual Studio .NET 2003 built-in debugger can be used for debugging Win32/Win64 applications. The debuggers can set breakpoints, execute the program or step through it one line at a time, and examine memory and registers. The debuggers understand the debugging information in the application and can display the source with the executing program. The symbolic debuggers and the associated tools are:

- Microsoft Windows NT® Symbolic Debugger (NTSD) debugs user mode programs.
- Debug Monitor (DbMon) runs in its own console window and displays messages sent by the application.

- MAPSYM converts the contents of your application's symbol map (.map) file into a symbol.
- WinDBG is used for debugging Windows-based programs, services, and kernel-mode drivers, which provides source-level debugging through a graphical user interface (GUI).

WinDbg uses the Microsoft Visual Studio® debug symbol formats for source-level debugging. It can access any public function names and variables that are exposed by the modules compiled with Codeview (.pdb) symbol files. WinDbg can view source code, set breakpoints, and view variables (including C++ objects), stack traces, and memory. It includes a command window for issuing a wide variety of commands not available through the drop-down menus. WinDbg also allows remote debugging of user-mode code.

Note More information on debugging tools for Windows 32-bit is available at http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx.

More information on debugging tools for Windows 64-bit is available at http://www.microsoft.com/whdc/devtools/debugging/64bit-home.mspx.

Application Build Tools

Most migration projects rely heavily on the text files tools. WinDiff, for example, is a graphical tool for comparing files and directories, recursively, if required. Files with differences are highlighted with colors for your convenience. Another useful tool is Where, which finds a file by name or pattern in a directory or subdirectories.

The Platform SDK also contains the Windows Script Technologies, which include Microsoft JScript® and Visual Basic® Scripting Edition (VBScript) engines. JScript is the Windows implementation of ECMAScript or JavaScript. VBScript is based on Visual Basic. Both the JScript and VBScript engines run in the context of the Windows Script Host (WSH), which is a language-independent environment for scripting engines. Windows Script Host allows scripts to be run from the command line or from the Windows environment. Scripts can run in batch mode and do not require user input. Scripts can also use the objects provided by WSH to access operating system resources.

In addition, Visual Studio 6.0 also provides tools that help in the design and development of the software. Some of the Visual Studio tools are:

- Visual Modeler assists object-oriented programming to create applications quickly and easily and contains components that can be reused in other applications. Visual Modeler enhances the capability to develop scalable applications supporting reverse engineering and round-trip engineering.
- Application Performance Explorer monitors the performance of applications.
- Remote Automation Connection Manager sets or revokes the permissions on the COM components.

Third-Party Extension Tools

You can use third-party tools that integrate with Visual Studio in the development environment for Win 32/Win64. Add-ins can also be used. For more information on this, refer to “Third-Party Extension Tools” in the next section.

Source Code Management Tools

Source code management tools can be integrated into the Visual Studio .NET 2003 IDE to perform an automatic checkout when a developer modifies a file. For more information on this, refer to “Source Code Management Tools” in the next section.

.NET

The following tools and IDEs are useful for developing applications using .NET.

.NET Framework SDK

In addition to .NET Framework, the Microsoft .NET Framework SDK includes everything you need to write, build, test, and deploy .NET Framework applications like command-line tools and compilers. The SDK also contains excellent documentation and Quick Start tutorials that help you learn .NET technologies with ease. It is available free of cost and you can download the entire Framework SDK from the MSDN Web site at http://www.microsoft.com/downloads/details.aspx?familyid=9B3A2CA6-3647-4070-9F41-A333C6B9181D&displaylang=en.

Note that when you install Visual Studio .NET 2003, the .NET Framework is automatically installed on your computer.

.NET Redistributable

To run applications developed using .NET Framework, the computer must have certain run-time files installed. These files are collectively called .NET redistributable. The .NET redistributable provides one redistributable installer that contains the common language runtime (CLR) and .NET Framework components that are necessary to run .NET Framework applications. The redistributable is available as a stand-alone executable and can be installed manually or as a part of your application setup.

For more information on CLR, refer to the .NET build volume (Volume 4) of this guide.

You can download .NET redistributable from http://www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5&DisplayLang=en.

Technical information on .NET redistributable is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetdep/html/dotnetfxref.asp.

Note If you have installed the .NET Framework SDK, you do not need to install .NET redistributable separately. Note that there is a difference between the .NET Framework SDK and .NET redistributable in terms of purpose, tools, and documentation supplied. The .NET Framework SDK is intended to "develop and execute" applications, whereas .NET redistributable is intended to "run" .NET applications.

Integrated Development Environments

Unlike the SDK command-line tools, IDEs offer users an integrated set of tools such as editors, compilers, and debuggers. Visual Studio .NET 2003, the IDE discussed here, comes with a choice of .NET languages (such as C#, VB.NET, and Managed C++), compilers and debuggers, and a large set of integrated tools that do everything from creating resources to packaging applications for installation.

The Visual Studio .NET 2003 IDE is an integrated visual development environment that allows you to program on .NET Framework on various languages. Visual Studio .NET 2003 is available in the following editions.

You can select the edition that is appropriate for the kind of development you are doing:

- Professional
- Enterprise Developer
- Enterprise Architect

The Visual Studio .NET 2003 Professional edition offers a development tool for creating the various types of applications mentioned previously.

The Visual Studio .NET 2003 Enterprise Developer (VSED) edition provides all features of the Professional edition, along with additional capabilities for enterprise development. These additional features include such functionality as collaborative team development, third-party tool integration for building XML Web services, and built-in project templates with architectural guidelines, spanning a comprehensive project life cycle.

The Visual Studio .NET 2003 Enterprise Architect (VSEA) edition provides all features of the Enterprise Developer edition, along with capabilities for designing, specifying, and communicating application architecture and functionality. These additional features include Visual Designer for XML Web services, Unified Modeling Language (UML) support, and enterprise templates for development guidelines and policies. A complete comparison of these editions is available at http://msdn.microsoft.com/vstudio/products/compare/default.aspx.

In addition to these editions, special language-specific editions are available with Visual Studio .NET 2003. A complete comparison of the standard editions listed here with the professional edition of Visual Studio .NET 2003 is available at http://msdn.microsoft.com/vstudio/previous/2003/.

This guide considers Visual Studio .NET 2003 as the integrated development environment (IDE) for developing .NET applications. Although newer technologies exist, the guide is based on best practices developed by partners and customers. As new practices establish, they will be incorporated into future releases of the guide. These latest technologies and their features are briefly described in “Roadmap for Future Migrations” of Volume 5: Chapter 2, “Operations” of this guide.

Compiling Tools

Visual Studio .NET 2003 provides compilers for the common .NET-supported languages like Visual C#.NET, VB.NET, Managed C++.NET, Visual J#.NET, and JScript.NET. These compilers are used to produce a highly optimized Intermediate Language (IL) code. Later, when the code is executed for the first time, a JIT (Just In Time) compiler translates this IL code to the computer code.

You can also compile your programs from the command line. For example, the following command compiles a C# program named Test.cs:

```
csc Test.cs
```

You can also specify several appropriate options for each of the language compilers, both from the IDE and the command line.

More information on compiler options is available at C++ Compiler options at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/vcrefcompileroptionslistedalphabetically.asp.

More information on compiler options is available at C# Compiler options at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cscomp/html/vcrefcsharpcompileroptionslistedalphabetically.asp.

Debugging Tools

The process of debugging applications essentially goes in the reverse direction from the process of compiling the code. As you execute the code, the debugger first maps the native code back to MSIL, and then maps the MSIL code back to the source code using programmer database (PDB) files. Therefore, to enable debugging, it is necessary to generate the required mapping information at each stage in the compilation process.

To debug .NET Framework applications, Microsoft provides a debugger integrated with Visual Studio .NET 2003. When you run your code in debug mode, you can watch how it executes each line of code. To run your project in the debug mode, click **Start** on the **Debug** menu. Alternatively, press F5 on your keyboard.

Apart from the debugger in Visual Studio .NET 2003, you can use the Microsoft CLR debugger (DbgClr.exe) and the Runtime Debugger (Cordbg.exe), which are included with the .NET Framework SDK, to debug your applications. Table 4.2 explains the usage of these tools.

Table 4.2. Debugging Tools

Tool	Description	Purpose
Visual Studio .NET Debugger	Provides debugging for various sets of applications to inspect and modify the state of the programs.	Allows debugging of managed code, Win32 native code, COM, ActiveX controls, script, Web applications, and SQL stored procedures and Just-In-Time debugging.
Microsoft CLR Debugger (DbgClr.exe)	Provides debugging services with a graphical interface to help application developers find and fix bugs in programs that target the runtime.	Allows debugging of applications written in Visual Basic .NET or C# or managed C++ and compiled only for common language runtime. Does not support debugging of Win32 native code, unmanaged C++ code, and remote debugging.
Runtime Debugger (Cordbg.exe)	Provides command-line debugging services using the CLR Debug API. Used to find and fix bugs in programs that target the runtime.	Allows command-line debugging with the compiled managed code. Does not support compiling the code and debugging unmanaged code.

Application Build Tools

In addition to well-known tools such as Visual Studio .NET 2003, there are a multitude of small but effective tools that are available free of cost from the .NET community. These tools make the process of building .NET applications faster and easier. Some of these tools are:

- NUnit to write unit tests.
- NDoc to create code documentation.
- NAnt to build your solutions.
- CodeSmith to generate code.
- FxCop to police your code.
- Snippet compiler to compile small bits of code.
- Two different switcher tools: the ASP.NET Version Switcher and the Visual Studio .NET Project Converter.
- Regulator to build regular expressions.
- .NET Reflector to examine assemblies.

More information on these tools is available at http://msdn.microsoft.com/msdnmag/issues/04/07/MustHaveTools/default.aspx.

Third-Party Extension Tools

Third-party tools that integrate with Visual Studio .NET 2003 to enable detection of run-time errors include:

- BoundsChecker from Compuware.
- PurifyPlus from IBM.
- MKS Software.

BoundsChecker, a run-time error detection tool, was originally a single product. It is now part of the DevPartner Studio suite, which contains performance as well as debugging tools. BoundsChecker can diagnose programming problems unique to C++, such as stack and heap memory errors, and leaks of memory and resources. BoundsChecker also eliminates common Windows mismatched argument problems by checking API calls for Win32, ODBC, and Internet, among others.

PurifyPlus contains the Purify error and memory problem detection tool as well as performance tools to analyze bottlenecks and to detect untested code. PurifyPlus also checks for errors in API calls, including Win32, ActiveX® controls, COM objects, and DLLs. PurifyPlus detects C++ memory problems such as heap and stack errors, pointer errors, memory usage errors, and handle errors. PurifyPlus also works outside the Visual Studio IDE for stand-alone debugging.

MKS software offers an extension to Visual Studio .NET 2003 to support the UNIX **lex** and **yacc** tools. The **lex** scanning tool analyzes input text files for the occurrence of predefined text patterns. The **yacc** (another compiler tool) is a parser that takes a setup input token and applies a set of grammar rules to the token. A combination of **lex** and **yacc** provides developers with a powerful method to generate compilers, front-end processors for compilers, text processors, and language translators. The **lex** tool creates tokens based on regular expressions, and **yacc** performs grammar rules on the tokens. The MKS **lex** and **yacc** tools work as add-ins to Visual Studio .NET 2003 that allow developers to create scanners and parsers within the Visual Studio .NET 2003 IDE.

Source Code Management Tools

Source code management tools can be integrated into the Visual Studio .NET 2003 IDE to perform an automatic checkout when a developer modifies a file. Source code management tools that integrate with Visual Studio .NET 2003 have point-and-click interfaces, which may be new to UNIX developers.

Visual SourceSafe is the source code management tool for Win32/Win64 and .NET applications. It creates projects in a tree view structure and provides complete source control for checking files in and out and tracking differences between files. Visual SourceSafe uses COM automation for customization.

The automation model can help automate the migration of source code from UNIX to Windows by automating repetitive tasks. For example, in a batch build process, you can use the automation features of Visual SourceSafe to retrieve the source code automatically before it is compiled.

Like other source code control systems, Visual SourceSafe maintains a library of projects from which users check out files for exclusive or nonexclusive access. Users check in the files after they are modified and tested. If nonexclusive checkout is allowed, the files are interactively merged at check-in time, if necessary. You can recall each incremental checked-in version if functionality is inadvertently deleted. Visual SourceSafe can also create releases by labeling the files. Visual SourceSafe handles text as well as binary files.

.NET Tools

The .NET Framework SDK also includes several tools for examining assemblies and working with the system assembly cache. Table 4.3 lists the packaging and deploying tools.

Table 4.3. .NET Tools

Tool	Purpose
Assembly Binding Log Viewer (Fuslogvw.exe)	Windows-based tool for examining assembly and resource bind requests.
Assembly Linker (Al.exe)	Assembly linker for creating assembly manifests, satellite assemblies, and working with the GAC.
Global Assembly Cache Tool (Gacutil.exe)	Console tool that manages the GAC and download caches.
MSIL Disassembler (Ildasm.exe)	Windows-based tool for examining the manifest (containing metadata) and MSIL code inside assemblies.
Strong Name Tool (Sn.exe)	Console tool to help generate strongly named assemblies.
File Signing Tool (SignCode.exe)	Wizard that helps to sign a portable executable (PE) file (.dll, .exe, .ocx, or .cab file) with an Authenticode digital signature.

More information on these packaging and deploying tools is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptutorials/html/appendix_b___packaging_and_deployment_tools.asp.

Note For information on tools related to testing and on tuning of applications, refer to Chapter 9, "Stabilizing Phase" of the build volumes (Volume 2, Volume 3, and Volume 4) of this guide.

Migrating Source Under Source Control

Source control systems impose additional requirements on moving the code. First, if the source control system allows both UNIX and Windows clients, then creating the project environment is the main step. In this case, after the Windows project environment is created, the source is moved by just checking out the source to the appropriate project area.

However, if the source control system does not allow both UNIX and Windows clients, perform the following steps:

1. Check out the project source from the source control system on UNIX.
2. Package the source on UNIX.
3. Copy the source to Windows.
4. Unpack the source on Windows.
5. Check the source into source control on Windows.

The remainder of this section describes these steps by using tools that are available on both UNIX and Windows.

First, the user checks out the source from the UNIX source control system into a local working directory on UNIX. Most code management tools provide facilities for exporting an entire code hierarchy and creating a tar file directly. If this is not the case, the hierarchy must be checked out and then packaged. The steps for packaging, copying, and unpacking the source take place in much the same way as already described in this section.

After it is moved and unpacked, the source needs to be checked into source control in the Windows environment. From the command line, the administrator can create a project, add the files to the project, and check the files into Visual SourceSafe by using the following commands:

```
ss Create $/MySystem
```

```
ss Add <working directory path>\MySource.C
```

or

```
ss Checkin MySource.C
```

The following command recursively retrieves all files associated with the MySystem project in Visual SourceSafe:

```
ss Get $/MySystem –R
```

Configuring the Development Environment

Configuring the development environment involves setting up the environment variables, logon scripts, computer configuration, application configuration, and security attributes. The following sections describe the instructions for configuring the development environment for Windows Services for UNIX 3.5, Win32/Win64, and .NET applications.

Configuration for Windows Services for UNIX 3.5

The Interix environment must be configured directly after its installation. As part of its configuration, administrators can add directories to the PATH variable in startup files. For example, the administrator may want PATH to point to programs and shell scripts in a directory named *bin* under the home directory. For examples of how to modify PATH, under /etc, see the files profile, profile.lcl, and profile.usr.

To manage the logon environment, create logon scripts in the home directory. The Korn shell uses the logon script .profile to manage the user's profile. Interix provides the /etc/profile.usr file as a template for creating .profile.

To use the Interix template

1. Copy /etc/profile.usr to the home directory and rename it .profile.
2. Edit .profile to set the initial environment to the desired requirements.

To customize the environment in the C shell

- Create the files $HOME/.cshrc, $HOME/.history, $HOME/.login, and $HOME/.cshdirs. When a user logs on, the C shell runs all of these files.

Interix maps the root directory (/) to the Windows Services for UNIX installation directory, which is C:\SFU by default. Under the Interix root directory, there are subdirectories that usually exist in UNIX, such as /usr and /etc. There are also virtual directories, such as /net for network resources and /dev for devices. In addition to entries for the usual devices, the /dev directory includes entries that correspond to Windows drive letters. For example, /dev/fs/A and /dev/fs/C correspond to drives A and C respectively.

Note If you change this default installation path, you should avoid spaces in the path name to minimize the confusion for UNIX programs and tools.

By default, Interix stores system binaries in one of the following three directories:

- /bin
- /usr/contrib.
- /usr/contrib/bin

Interix stores the X11 binaries in /usr/X11R6/bin and the Win32 binaries in /usr/contrib/win32/bin. Some symbolic links are added, for example, so that /usr/bin maps to /bin. Because of this, the file system tree's configuration supports most UNIX application and script ports to the Interix environment.

The creation and management of a development project in Interix proceeds just as in any UNIX environment. First, identify the location and format of the application to be prototyped (ported), and then determine its format (such as compressed tar file). Decide the location to which you want to load the source tree (and verify that there is enough disk space). Be sure that the development platform has the tools required for this application and for any other applications that will be ported to or developed in the Interix environment.

Configuration for Win32/Win64

Visual Studio .NET 2003 or Visual Studio 6.0 are configured to work with Win32 applications at the time of installation. You can configure to work with 64-bit applications using the /useenv option.

To load the 64-bit tool chain, execute the following statements:

```
call "C:\Program Files\SDK\SetEnv.Bat" /AMD64 /RETAIL

start devenv /useenv
```

To resume working with the 32-bit tool chain, quit VS6 and then use the following statements to relaunch the IDE:

```
call "C:\Program Files\Microsoft Visual Studio .NET 2003\VC7\Bin\VCVARS32.BAT"

start devenv /useenv
```

Note For more information on this, refer to "Using the Development Environment" section in the Win32/Win64 build volume (Volume 3).

Configuration for .NET

The .NET Framework provides to developers and administrators control and flexibility over the way applications run. An administrator can control which protected resources an application can access, which versions of assemblies an application will use, and where remote applications and objects will be located. Developers can add settings in the configuration files, eliminating the need to recompile an application every time a setting changes. Configuration files are XML files that can be changed as needed. Developers can use configuration files to change settings without recompiling applications. Administrators can use configuration files to set policies that affect how applications run on their computers.

There are three types of configuration files:

- Computer configuration files (Machine.config), which contain settings that apply to the entire computer.
- Application configuration files, which contain settings specific to an application.
- Security configuration files (Security.config), which contain settings specific to the computer security.

More information on configuring applications in .NET is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconconfiguringnetframeworkapplications.asp.

Creating Projects

To make all the individual files or components into an application, a project needs to be created from those files. A project consists of separate components that are stored as individual files in a solution. A simple project might consist of a form or HTML document, a database, source code files, and a project file. All projects are contained within an application, and every project contains a unique project file. The project file is a list of the project's files and objects and is used to store information about the project-specific environment options. The following sections describe the instructions for creating the projects for Interix, Win32/Win64, and .NET applications.

Creating a Project for Windows Services for UNIX 3.5

To use Visual Studio .NET 2003 to compile your Windows Services for UNIX application

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **Project Types** list on the **New Project** dialog box, click **Visual C++ Projects**.
3. Click **Win32**, and select **Win32 Console Project** as the Template.
4. Before adding a new project to this solution, set the name of the project and the path where this project will reside on the hard drive of your computer.
5. In the **Location** text box, enter the path where you want this project to reside. Visual Studio .NET creates the necessary path and will create a folder name with the same name as the project. For example, if you fill in the name CreateThread and set the path to D:\MySamples, the solution will be created in D:\MySamples\ CreateThread \ CreateThread.sln.

To compile your application

1. Add the C source file to the Source Files folder shown on the right side of the Solution Explorer.
2. On the **Project** menu, click **Settings**, select **Configuration Properties** folder and then **Build Events** folder, and then select **Configuration Type** as **Utility**.
3. Go to **Pre-Build Event** in the **Build Events** tab, and enter the following as **Command Line**:

   ```
   POSIX.EXE /c %SFUDIR%\bin\ksh -c "/usr/bin/cross_build.sh
   \"`pwd`\"   releasebuild"
   ```

Note that you need to have a script file cross_build.sh in the %SFUDIR%\bin directory. Copy the following as the contents of this file:

```
echo "Starting gcc build in directory: " $1
. /etc/profile
cd `winpath2unix ‘$1’`
make $2
```

You should keep your makefile in the project folder, which is D:\MySamples\ CreateThread in this case.

Now you can build your solution and the executable is put in the folder specified in the makefile.

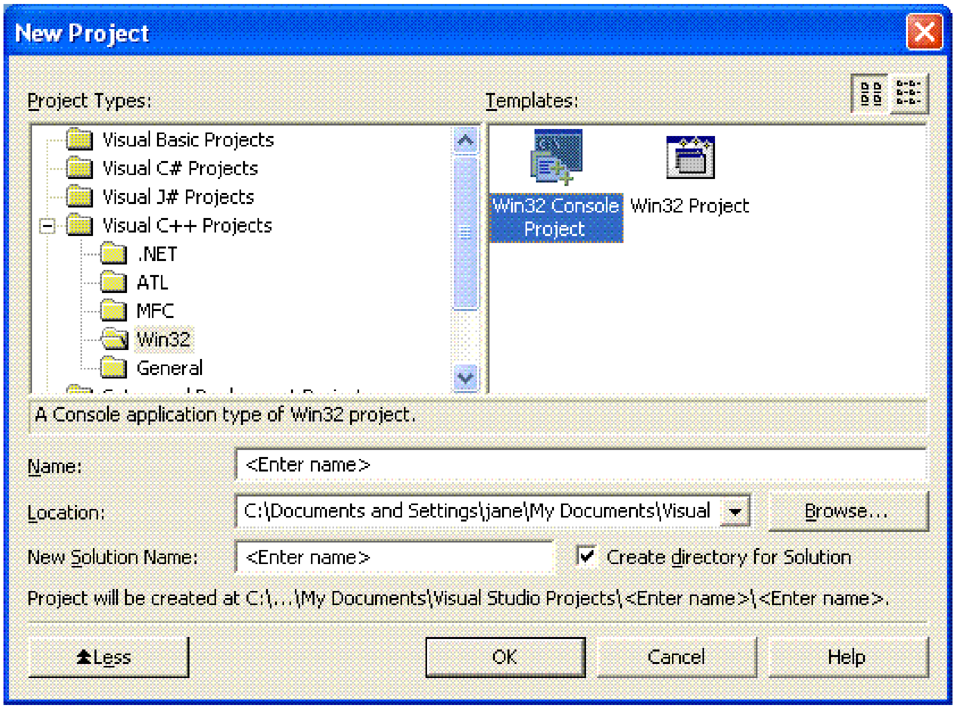
To run the application, the usual UNIX method is used, for example:

```
./CreateThread
```

Creating a Project for Win32/Win64 and .NET

To use Visual Studio .NET 2003 to compile your Win32/Win64 or .NET application

- In Visual Studio .NET 2003, on the **File** menu, point to **New**, and then click **Project**. The dialog box shown in Figure 4.1 appears.



A Console application type of Win32 project.

Name: <Enter name>

Location: C:\Documents and Settings\jane\My Documents\Visual

New Solution Name: <Enter name>

☑ Create directory for Solution

Project will be created at C:\...\My Documents\Visual Studio Projects\<Enter name>\<Enter name>.

Figure 4.1. The New Project dialog box allows you to create a new solution of a particular project type.

When putting together an application in Visual Studio .NET 2003, you may have multiple projects. The set of projects together make up a solution.

The **New Project** dialog box allows you to create a new solution of a particular project type.

In the **Project Types** list in this dialog box, you can choose the type of project that you want to create. Depending on the options you select when you installed your Visual Studio environment, you can choose from a C#, C++, or Visual Basic .NET project; or possibly, a project of other programming languages. Not all of these languages for Visual Studio come from Microsoft. There are other organizations that develop applications that use .NET Framework.

In the **Templates** list on this screen, you can choose a default template for the type of project you want to create. There are many different templates to choose from. The project type for creating a Win32/Win64 console application is shown in the figure. Other options to create MFC or ATL applications are also available.

Before adding a new project to this solution, set the name of the project and the path where this project will reside on the hard drive of your computer. In the **Location** text box, enter the path where you want this project to reside. Visual Studio .NET 2003 creates the necessary path and will create a folder name with the same name as the project. The default path for Visual Studio .NET 2003 is My Documents\Visual Studio Projects. For example, if you fill in the name **LoginTest** and set the path to D:\MySamples, this solution will be created in D:\MySamples\LoginTest\LoginTest.sln.

Visual Studio .NET 2003 supports a variety of project template types that you can choose from based on the application requirements.

More information on Visual Studio .NET 2003 project template types is available at C# and Visual Basic .NET project template types at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/vxoridebuggingcvisualbasicprojects.asp.

More information on Visual Studio .NET 2003 project template types is available at Visual C++ project template types at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/vxoriDebuggingUnmanagedCCProjects.asp.

Populating the Development Environment

One problem you need to solve early in the Planning Phase is how to transfer files from one system to another. A UNIX-to-Windows migration first requires a plan for moving UNIX code to a Windows-based computer. Then you must develop a plan that identifies the portions of the code you will be interoperating and the portions of code you will be redeveloping.

This section discusses the first part—standard considerations for moving code between systems in general—and then provides guidance on populating the Visual Studio .NET 2003 environment with the source.

After obtaining the original source from the source system—as a code hierarchy, as an export from a source control system, or as an archive—the build process must also be created and configured. Some testing of the build process is possible, but testing whether the code compiles and links is part of the port or the rewrite itself.

Standard Considerations for Moving Code Between Systems

When moving source code from one platform to another, you need to first look at the general structure and location of the various files that need to be moved.

UNIX applications typically go in the /usr directory. Many UNIX administrators further segregate system programs from added user programs. Commonly, user programs are added to /usr/local, and system programs go in the /usr root.

Developers of nonsystem programs ordinarily find source code files in /usr/local on the UNIX platform. Within this directory, common directories include:

- /usr/local/bin for executable files.
- /usr/local/include for header files.
- /usr/local/lib for library files.

Source code for user projects is usually added in a hierarchy from a directory based in /usr/local. For example, the source files for the myapp program, which is part of the mysystem application, are in /usr/local/mysystem/myapp.

On UNIX, a developer's working source files may be in a structure similar to that described for myapp, but in the developer's home directory instead. If the developer accesses these files from net shares by using network file system (NFS), the files may be found in a structure such as /export/home/<user name>/src/mysystem/myapp.

Windows uses different directory structures for project files. First, applications install their shared and read-only components to <drive letter>\Program Files. Windows uses a drive letter instead of the single-rooted UNIX file structure. The actual drive letter used varies according to the system configuration. A directory for the primary development tools and several directories for additional systems are usually found under C:\Program Files.

For example, if the developer needs to use development resources from the base system and an add-on service called mysystem, the developer can expect to find the base development files in C:\Program Files\mysystem and specific source files in C:\Program Files\mysystem\myapp. Under each top-level directory, the developer can expect to find a directory for binaries, header files, and libraries, such as:

- C:\Program Files\mysystem\bin for binaries.
- C:\Program Files\mysystem\include for header files.
- C:\Program Files\mysystem\lib for libraries.

For the released (permanent) version of the source for myapp, look in C:\My Programs\mysystem\myapp. The name of the top-level directory can vary because most installation programs allow the user to customize the name of the top-level directory. However, the directory structure under the top-level installation directory may be the same.

Finally, the developer needs a space for the working myapp source files. Unlike UNIX, in Windows these files do not go under the same directory tree as the rest of the program files. In .NET, the project files belong in the area allocated for the developer's documents, for example, My Documents\Visual Studio Projects. By default, this area is C:\Documents and Settings\<user name>\My Documents\Visual Studio Projects. Therefore, look for the myapp sources under C:\Documents and Settings\<user name>\My Documents\Visual Studio Projects\mysystem\myapp.

Note When migrating files, remember that the UNIX text file format is different from the Windows text file format. In UNIX, the line-feed character (hex 0A) is used to indicate a new line. In Windows, the combination is a carriage return (hex 0D) and line feed. To determine whether this conversion is needed, open a source file in Notepad. If the source lacks the proper line feeds, a conversion is needed. A number of tools can be used for this. One of them, **flip**, is delivered with Interix. Depending on the target environment, Windows files may need to be converted to UNIX, or vice versa. Interix provides the **flip** tool for these conversions. Also if you “**flip**” the files, some tools may break.

For example, the following command changes a UNIX text file to a Windows text file:

```
% flip –m MySource.C
```

The following command converts startup files created using Windows Notepad to the UNIX format:

```
% flip -u filename
```

The **flip** tool also allows the use of wildcards.

Exporting Files from the UNIX Environment

After placing the source files in a working directory, they can be packaged together using the **tar** (Tape Archive) UNIX tool. The tool was named in the period when it was used for packing files together on tape libraries. Now, this tool can create a single file that represents a number of files from a common source, which makes **tar** useful for packing together the files of a project. Use the **–c** option with **tar** to create the archive and the **-r** option to add files to the archive.

A file that represents the output of **tar** is sometimes called a “tar ball.” After the tar ball is created, the developer can compress it using **gz** (GNU Zip).

Now the project is ready to copy to Windows. To do this, use one of the following methods:

- Use file transfer protocol (FTP) from an FTP server.

  For an FTP file transfer, be sure the session is in binary (image) transfer mode before getting the compressed archive file.
- Use file copy from a file server such as Web, NFS, or server message block (SMB).
- Copy from a CD-ROM drive.

A copy can start from the UNIX side or the Windows side. To copy files from one system to another, the administrator needs to allow access to both the source and target systems from the computer issuing the copy commands. When using NFS or CIFS, the remote share needs to be mounted.

Note The .gz file type is not recognized on a standard Windows installation. UNIX versions of the ZIP compression algorithm are widely available, and the .zip file type is natively recognized by Windows XP and Windows Server 2003.

Populating an Interix Environment

Importing an application's configuration settings, build, and source files into the Interix development environment is similar to moving the application between two UNIX systems. Applications are typically compressed or uncompressed **tar** archive files. **tar** is most commonly used in tandem with an external compression tool such as **gzip** or **bzip2**.

Interix provides a program group that can start either a **ksh** or a **csh** shell because Interix integrates with the Windows environment.

To start a ksh or csh shell

- Click **Start**, point to **Programs**, point to **Windows Services for UNIX**, and click **Korn Shell**.

  Alternatively, point to **Windows Services for UNIX**, and click **C Shell**.

To import the application configuration settings and build and source files

1. Change the working directory to /usr/examples with the **cd** command, and then copy the compressed archive by typing the following:

   ```
   % cd /usr/examples
   % cp /dev/fs/<Drive Letter>/filename.tar.gz.
   ```

   (The period at the end of the last line indicates the current directory.)
2. If the archive currently exists on a network share and the network share has been mounted to a drive letter, move the archive into the environment by typing:

   ```
   % cp /dev/fs/<drive letter>/<source directory>/filename.tar.gz.
   ```
3. Uncompress and extract the files from the archive by typing the following:

   ```
   % gunzip < filename.tar.gz | tar xf –
   ```

After the files are extracted from the archive, populating the Interix environment is complete.

Populating for Win32/Win64 and .NET

Using the Platform SDK or Visual Studio .NET 2003, the procedure for populating the Windows environment is almost the same as that described earlier in "Populating an Interix Environment" in this chapter. However, the intended location of the source files is similar to C:\My Programs\mysystem\myapp, depending on the name of the system and application subsystem that you are importing.

After the tar file or compressed tar file is in the Windows environment, it must be unpacked into the local Windows project area using the Windows-based version of **tar**. Systems that include **tar** include Interix, MKS NutCracker, and Cygwin. A gz archive can be opened on Windows using the WinZip archive tool.

After unpacking the source files, the text for the source file may need a minor conversion. (For more information on **flip**, refer to the "Standard Considerations for Moving Code Between Systems" section earlier in this chapter.)

Note In a case where large amounts of data need to be communicated over time from a UNIX environment or version control system to Windows, it will often be helpful to install Windows Services for UNIX 3.5 because it includes most of the necessary tools and is capable of storing files in the NTFS file system for subsequent use by any Windows program.

Migrating the Build Environment

This section discusses the build environments in Windows and how build environments can be migrated from UNIX to Windows. As part of setting up the development environment, the build environment is also migrated from UNIX to Windows to build the migrated applications.

Planning for migration of the build environment in the Planning Phase helps to assess and understand the existing UNIX environment. This assists in identifying the requirements and appropriate tools for building the project on the target Windows environment.

Migrating a Build Environment to Windows SDK

The Platform SDK can be used to compile and link C and C++ programs that do not include Visual Studio .NET projects. The **nmake** tool can be used to build make-based applications in Visual Studio. This tool is very similar to the UNIX **make** tool in behaviors and capabilities, with minor variations in the command-line options.

More information on the **nmake** tool is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcepb40/html/_wcepb_nmake_tool.asp.

Also refer to Chapter 3, “Planning Your Build Migration” in the *Solution Guide for Migrating UNIX Build Environment*.

Migrating a Build Environment to Visual Studio .NET 2003

To maintain UNIX-style code on the Windows platform, you must take into account the nature of UNIX applications and the build procedures. In UNIX, programs are compiled into object modules. The object modules can be linked into executable files or they can be used to create libraries. Libraries can either be static archives (created with **ar**) used to add code to executable files, or they can be shared objects. Often, the UNIX C and C++ compilers invoke the linking tools implicitly.

Windows programs are compiled with the compiler. They are linked as executable files, static libraries, or DLLs. UNIX and Windows compilation and linking are almost the same except in certain cases. The compiling and linking by cl.exe in the Windows environment and by gcc in the UNIX environment is performed in the same sequence, that is, precompilation with optimization, object generation, and creation of executables or libraries. Table 4.4 compares the UNIX and Windows methods for creating executable files, static libraries, and shared libraries.

Table 4.4. UNIX and Windows File Creation Comparison

File Type	Tool to Create	Command Options
UNIX executable file	**Linker (ld)**	The -o option can be used to specify the output file name from the input object files. Refer to man pages of the corresponding UNIX vendor for the command options.
UNIX static library	**ar**	The -r option is used to create an archive library from the given input object files. Refer to man pages of the corresponding UNIX vendor for the command options.
UNIX shared library	**Linker (ld)**	**-shared**
Windows executable file or Windows dynamic library	**Linker**	Link.exe can be used with the command-line option –OUT:*filename*, where *filename* is the name of the output executable. This overrides the default name of output file. More information on Link.exe command-line options is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcepbguide5/html/wce50conCompilerSetupMechanisms.asp.
Windows static library	**LIB.exe**	Lib.exe can be used with the command-line option –OUT:*filename*, where *filename* is the name of the output library. This overrides the default name of the output file. More information on Lib.exe command-line options is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vcce4/html/evgrfRunningLIB.asp.

In .NET Framework, the source code is compiled and run using a two-step process. In the first step, the source code is compiled to Microsoft intermediate language (MSIL) code using a .NET Framework-compatible compiler, such as Visual C# .NET or Visual C++.NET. In the second step, the MSIL code is compiled to native code.

To migrate the build environment, use one of the following methods:

- Manually recreate the build structure. (Refer to the "Recreating the Build Structure" section later in this chapter.)
- Use batch builds in conjunction with Visual Studio .NET 2003. (Refer to the "Using Batch Builds with Visual Studio .NET 2003" later in this chapter.)
- Create a custom tool to manage projects and builds. (Refer to the "Creating a Custom Tool" section later in this chapter.)

Migrating the build environment essentially includes migrating the makefile from the UNIX environment to the Windows environment. Project makefiles consist of a set of commands to build the project executable or library from the input source file with the given compiler options and the dependent libraries. Each method starts with the UNIX makefile, so the first step is to look at the project makefiles to see what information needs to be migrated. You can use the **make** with **-n** switch to run the makefile without actually executing the commands in the makefile. This helps in finding the issues with actual files and sequences of events and tools involved in the makefile commands.

Look at the setting and configurations first. To do so, use a makefile to create dependencies and rules that generate the makefile target, usually the linked binary, such as an executable file. In addition to rules and dependencies, macros can be created to represent variables and options for each function and the corresponding directory locations. Table 4.5 lists common examples of macros in a makefile.

Table 4.5. Makefile Macro Examples

Macro	Used For
CC	An alias for the C and C++ compiler.
CFLAGS	The C compiler option setting.
CPPFLAGS	The C++ compiler settings.
LDFLAGS	Linker option flags.
INCLUDE	The directory path for header files.
LIB	Libraries to include at link time.

The UNIX variables, environment variables, or the variables defined in the makefiles CFLAGS, LDFLAGS, LIB, and INCLUDE yield the main information required to create a Visual Studio project on Windows or to create an nmake file on Windows for a batch build. Makefiles can be viewed by using a UNIX text editor, such as **vi**. Options can also be extracted by using UNIX text tools, such as **grep** o**r awk**. In the following example**, grep** finds the text pattern CFLAGS in the file mymake:

```
% grep CFLAGS mymake
```

Dependency rules can also be expressed in the makefile. Table 4.6 lists these dependency rules.

Table 4.6. Makefile Dependency Rules

Symbol	Action
:	Target is out of date, depending on the source.
!	Target is repeated as sources are examined.
::	Target is accumulated.

A special type of dependency can be expressed by inference rules. Using these rules, target dependencies can be grouped together by common attributes. For example, a .SUFFIXES rule allows files with a common file extension to have the same dependencies.

Recreating the Build Structure

Often, for small to medium-sized applications, the easiest way to migrate the project is to create the Visual Studio project manually. Because of their smaller size, this technique is useful for prototype applications. It is especially suited to applications that have dependencies on only a single file tree, with all the sources under a single starting directory.

Larger projects may require another scheme because they have a greater number of source files and the UNIX build procedure is more complex. A long-term migration goal may be to standardize on the Visual Studio .NET tools. When all applications, including the largest ones, are considered, manually creating and maintaining the Visual Studio projects is the best solution.

Converting an Application Build from a Makefile-based to a Visual Studio-based Project

Rogue Wave SourcePro, a third-party product that provides cross-platform C++ development components, uses a makefile-based build model on all platforms. Therefore, it provides a good example of manually migrating a makefile-based project to a Visual Studio project.

To convert an application from a makefile-based build to a Visual Studio-based project, you must ensure that the flags passed to the compiler are the same, the necessary **#defines** are the same, and the required header file and library file paths are properly set in the Visual Studio .NET 2003 environment. Because the UNIX makefiles use shell scripting, all the shell commands need translation to the nmake syntax.

Note Additional information on NMAKE references is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcug98/html/_asug_overview.3a_.nmake_reference.asp.

One way to obtain this information is to run **make** from a command line on the original UNIX environment and redirect the output to a file, for example, the following command from the command line:

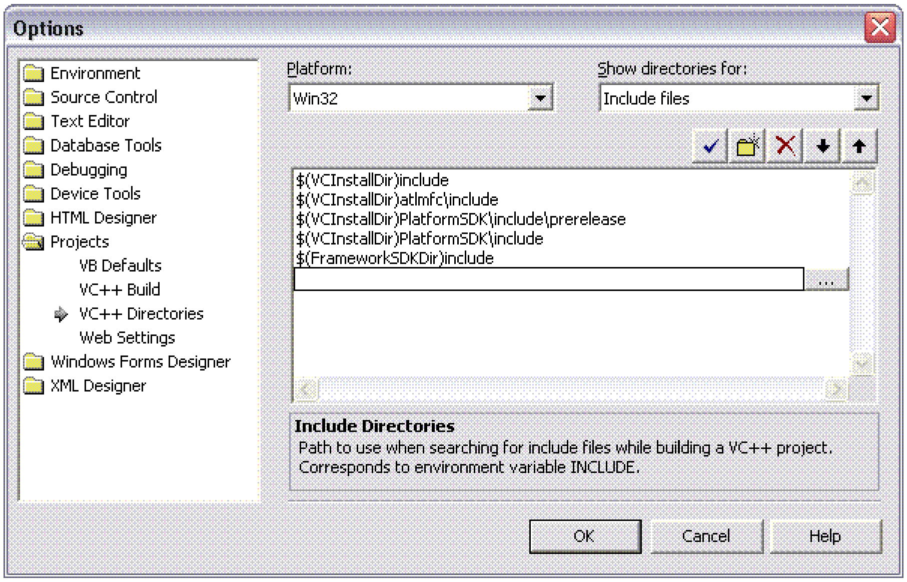
```
$ make –f project.mak > exampleBuild.txt
```

The preceding **make** command places the output of the compile and links it to the exampleBuild.txt file; then you need to move the makefile to the Windows environment and start making the necessary changes in that makefile.

After you have addressed these issues, you can continue development in the Visual Studio environment. The following is a general step-by-step guide to the process.

To convert a UNIX-based application to Microsoft Visual Studio .NET 2003 project

1. Copy the source and header files from UNIX to the Windows folder.
2. Create the Win32 empty console project with the same application name as on the UNIX folder specified previously.
3. Add the source code files to the project.
4. Add all the include directories from the INCLUDE environment variable or the makefile variable or **–I** option in the UNIX makefile to the Project include directories. To add the new directory to the Visual Studio .NET 2003 environment, on the **Tools** menu, click **Options**, select **Projects** folder, then click **VC++ Directories**, select the **Platform** and **Show Directories for** as Include files. Then click in the last row in the following list box, and edit or browse the new directory of include files. The dialog box shown in Figure 4.2 appears.



Platform: Win32

Show directories for: Include files

```
$(VCInstallDir)include
$(VCInstallDir)atlmfc\include
$(VCInstallDir)PlatformSDK\include\prerelease
$(VCInstallDir)PlatformSDK\include
$(FrameworkSDKDir)include
```

Include Directories
Path to use when searching for include files while building a VC++ project.
Corresponds to environment variable INCLUDE.

Figure 4.2. The Options dialog box allows you to set the Include files path.

5. Add all the Library directories from LD_LIBRARY_PATH environment or LIB or LIBRARY variables on UNIX makefiles to the project library directories To do this, on the **Tools** menu, click **Options**, select **Projects** folder, then click **VC++ Directories**, select the **Platform** and **Show Directories for** as Library files. Then click in the last row in the following list box, and edit or browse the new directory of Library files.

6. Add the preprocessor definitions added to the UNIX makefile to Visual Studio project. To do this, on the **Project** menu, click **Settings**, select **Configuration Properties** folder and then **C++** folder, select the **Configuration** as **All Configurations**, and add to **Preprocessor definitions**.

7. If any other compilation options are specified in the makefile, add them to the project. To do this, on the **Project** menu, click **Settings**, select **Configuration Properties** folder and then **C++** folder, select the **Configuration** as **All Configurations**, and add the appropriate compiler options to the additional option in the **Command line** tab.

8. To add additional libraries, identify the library names from the UNIX makefile and add them to Visual Studio .NET 2003 environment. To do this, on the **Project** menu, click **Settings**, select **Configuration Properties** folder and then **Linker** folder, and then click **Input** and add the library names to **Additional Dependencies**. You can also update the **Output file** in **General** tab.

9. To add any pre-build and post-build script execution, on the **Project** menu, click **Settings**, select **Configuration Properties** folder and then **Build Events** folder. Select the **Configuration** as **All Configurations**, and add the corresponding commands to execute in select **Pre-build step**, **Pre-link step**, or **Post-build step** of **Build Events** folder. This is generally used to copy the include files, library files, or output files to the specified folder after the build.

10. To specify the command-line arguments for your program, on the **Project** menu, click **Settings**, select **Configuration Properties** folder, and select the **Configuration** as **All Configurations**, select the **Debugging** folder, and add command-line arguments to **Command Arguments**.

11. Compile the project. If you receive error messages about conflicting compiler options, it means that the options cited in the error message were not found in the application. Remove any options that are not found and compile again.

The following steps describe an example application for converting to Microsoft Visual Studio project.

To convert a console-based UNIX example application to a Microsoft Visual Studio .NET 2003 project

1. Start Visual Studio .NET 2003. On the **File** menu, point to **New**, and then click **Project**.
2. On the **Project Types** list in the **New Project** dialog box, click **Visual C++ Projects**. Click **Win32**, and select **Win32 Console Project** as the **Template**.
3. Copy all of the source code files from the example application folder to the project folder.
4. Under **FileView** (on the left side of the screen), right-click **Source Files**, click **Add files to folder**, and select all of the .cpp files copied to the project folder.
5. Repeat step 4 for **Header Files** and add all the header files in the project folder.
6. On the **Project** menu, click **Properties**, select **Configuration Properties** folder, and select the **Configuration** as **All Configurations**, and then click the **C++** folder. At the bottom of the dialog box, you will find a text field for **Project Options**. Find and copy the compile options that were used in the application example. The following steps will guide you through the process:
   - Convert **-I**<*anything*> to **/I** "<*anything*>"**.** If a relative path is used in the application, you need to convert it to an absolute path.
   - Convert **-D**<*anything*> to **/D** "<*anything*>".
   - Convert **-GX** to **/GX**.
   - For other compiler options, identify the equivalent compiler options in Visual Studio .NET 2003 IDE. On the **Project** menu, click **Settings**, select **Configuration Properties** folder and then **C/C++** folder and select the equivalent options.

   Note More information on C/C++ compiler options in Visual Studio .NET 2003 IDE is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_asug_specifying_project_configuration_settings.asp.
7. On the **Project** menu, click **Properties**, select **Configuration Properties** folder and then **Linker** folder, and then click **Input** and add the library names. Use the following guidelines:
   - Find an **-L**<*anything*> from the example application, and copy it to **Additional library path** by entering <*anything*>*.*
   - Find the list of all libraries used in the example application, and copy the library names under **Additional Dependencies**. Remove any preexisting library names.
8. Compile the project. If you receive error messages about conflicting compiler options, it means that the options cited in the error message were not found in the example application. Remove any options that are not found.

Using Batch Builds with Visual Studio .NET 2003

Creating a Windows nmake file is similar to creating a Visual Studio project. However, creating a Windows makefile is different. When the Visual Studio .NET 2003 IDE creates a project, it automatically generates contents required to build an application at runtime by reading the project file. Although a batch build system can manage both the build process and the dependencies for a source code base with hundreds or thousands of files, in such a system it is often difficult to debug a single suspected problem file.

Managing Cross-Platform Builds

During a migration, some applications may still need to be maintained and built on UNIX. These cases require a cross-platform development and build strategy.

When building an application on multiple platforms, follow these guidelines:

- Choose one platform as the main development platform.
- Identify and segregate platform-specific code that cannot be easily maintained with compiler directives (#ifdef). It is easier to maintain such code separately for each platform if it is separated into callable modules.
- Keep platform-specific code local to its target platform.
- Look for libraries that already offer cross-platform support, such as most C run-time libraries and OpenGL cross-platform libraries.
- Migrate source to multiple platforms with each release instead of synchronizing source across multiple platforms, that is, migrate the entire code to multiple platforms with each release, instead of synchronizing the changes in the source code from one platform to another platform.
- Use batch builds on each platform. An automated process can usually be used to move a batch build from one platform to another.
- Consider using a cross-platform build tool, such as GNU **make**.

Ultimately, the effort exerted to create and maintain a cross-platform build environment depends on how often the changes to an application in one platform need to be migrated to another. A quicker migration process may take less effort than a cross-platform build process.

Setting Up the Test Environment

Testing is the process of determining the differences between the expected results and the observed results of the migrated application. The output of the parent application and the output of the migrated application should be compared to ensure the success of the migrated application. A working test environment allows the solution to be properly tested so that it will have no negative impact on production systems.

Note It is generally a good idea to set up separate test servers that the testers can use. The entire team should be informed that anything on such servers could become unstable and require reinstallation.

When you set up the test environment, you replicate the live environment of the application on the test system. The actual setting up the test environment depends upon the application type. Setting up the test environment involves the following steps:

- **Gathering testing requirements**. This stage is essentially part of planning the migration project. In this stage, the user gathers the testing requirement with the following considerations:
  - Scope of testing.
  - Entry and exit criteria.
  - Success criteria.
  - Key stakeholders in testing phase.
  - Test requirements from requirements specifications and design documents.
  - Planned number of test cycles.

  - Planned test coverage.
  - Testing tools and automation strategy.
- **Test case planning and execution**. This stage defines the types of test cases for different types of testing, such as unit testing, integration testing, functionality testing, system testing, and user acceptance testing. This step also involves logging defects and usage of defect tracking tools.
- **Infrastructure**. This stage defines a strategy for infrastructure planning like network configuration, installation of the necessary software, setting up hardware requirements, and installation of the application-specific test data.
- **User environment**. This stage defines a strategy to understand the user-specific application configuration settings and migration of the settings to the test environment. This involves language support and browser setting in the case of Web applications.
- **Interoperability**. This stage defines the minimal changes required to use the application-specific data files and the configuration file across the UNIX and Windows platforms. It ensures interoperability and decreases the complication of files and data conversion across the platform.
- **Functionality**. This stage defines a strategy for the functional testing of the migrated applications, executing the test cases in the test environment, and evaluating the test results.
- **Security**. This stage defines a strategy to allow authorized users to use the applicable features of the application. This includes defining the user-specific security settings, application-specific security settings, computer-specific security settings, and network and other common resource-level security settings.
- **Performance**. This stage defines benchmarks for performance of the migrated applications using the existing benchmarks on the UNIX environment.

Setting Up the Staging Environment

The staging environment is where the team tests content and code being changed or deployed to ensure that they function as expected. The content and code are moved from the test environment to the staging environment before they are published to the production environment. Solution components successfully tested in the development/test environment may not necessarily work after all elements are in place and have been integrated. The staging environment provides a place to test code with all solution elements to make sure it works before it is put into production.

Hardware and software installed in the staging environment should mimic the production environment as closely as possible. Depending upon the availability of hardware resources, an individual server can perform several different server roles in the staging environment. However, it should be recognized that all deviations from the production environment detract from the representative value of testing and should therefore be clearly known, understood, and agreed upon and tracked. Ideally, the staging servers are built according to the documented server build procedures that have been developed. This ensures that testing accurately portrays what will be seen in production.

The staging environment attempts to simulate the target environment. The simulation may be achieved through the use of synthetic load generation or other custom or commercial tools. When the staging environment is different from the target environment to a significant degree (typically, in terms of size or capacity), or when project constraints do not support the construction of a staging environment, some testing must occur in the deployment environment. Such testing must be closely coordinated with the operations staff for that environment.

The instructions for setting up the staging environment is similar to the instruction provided for setting the up the test environment in the earlier section.

Interim Milestone: Development and Test Environment Setup

At this point, the team has the necessary settings in place to enable the work of building and stabilizing the solution. With the environment setup completed, the teams can now begin shifting their focus entirely to development work involved in migrating.

Key Milestone: Project Plans Approved

At the Project Plans Approved Milestone, the project team and key project stakeholders agree that interim milestones have been met, that due dates are realistic, that project roles and responsibilities are well defined, and that mechanisms are in place for addressing areas of project risk. The functional specifications, master project plan, and master project schedule provide the basis for making future trade-off decisions.

After the team approves the specifications, plans, and schedules, the documents become the project baseline. The baseline takes into account the various decisions that are reached by consensus by applying the three project planning variables: resources, schedule, and features. After the baseline is completed and approved, the team moves to the Developing Phase.

After the team defines a baseline, it is placed under change control. This does not mean that all decisions reached in the Planning Phase are final. But it does mean that as work progresses in the Developing Phase, the team should review and approve any suggested changes to the baseline.

After the closure of the Planning Phase, the team shifts its focus to the build solution components, following instructions to migrate the UNIX code to the Windows environment. The details of the development activities are described in the next build volumes (Volume 2, Volume 3, and Volume 4) of this guide.

Index